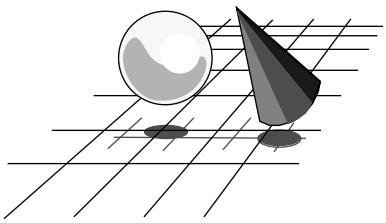
Programming with OpenGL: An Introduction

A TECHNICAL OVERVIEW OF THE OpenGL® GRAPHICS API

Course Notes for SIGGRAPH '96

Course Organizer

Tom McReynolds
Silicon Graphics Computer Systems



Course Speakers

Tom McReynolds
Silicon Graphics Computer Systems
Kathleen Danielson
Silicon Graphics Computer Systems

Abstract

OpenGL, the standard software interface for graphics hardware, allows programmers to create interactive 2D and 3D graphics applications on a variety of systems. With OpenGL you can create high-quality color images. OpenGL makes it easy to build geometric models, chang the viewing position, control the color and lighting of geometric primitives, and manipulate pixel and texture map images.

This course will cover an immediately applicable subset of OpenGL, so that you can write a simple graphics program, using shading, lighting, texturing and hidden surface removal. We will also discuss the new features in the most recent versions of OpenGL and GLX: OpenGL 1.1 and GLX 1.2.

About the Speakers

Tom McReynolds

Tom McReynolds is a software engineer in the Performer group at Silicon Graphics. Before that, he worked in the OpenGL group where he's implemented OpenGL extensions and done OpenGL performance work. Prior to SGI, he worked at Sun Microsystems, where he developed graphics hardware support software and graphics libraries, including XGL.

Tom is also an adjunct professor at Santa Clara University, where he teaches courses in computer graphics using the OpenGL library. Address: 2011 N. Shoreline Boulevard, Mountain View, CA 94043, E-mail: tomcat@asd.sgi.com, Phone: 415-933-5144, Fax: 415-965-2658

Kathleen Danielson

Kathleen Danielson is a software engineer in the OpenGL group at Silicon Graphics. At SGI she has implemented OpenGL extensions, improved OpenGL performance, and implemented Ada and Java bindings for OpenGL. Previous to SGI, she worked at Kubota Graphics in the graphics system software group, where she developed OpenGL libraries for several graphics hardware systems. Address: 2011 N. Shoreline Boulevard, Mountain View, CA 94043, E-mail: kat@asd.sgi.com, Phone: 415-933-1239, Fax: 415-965-2658

NOTICES

IRIS, Geometry Link, Geometry Partners, Geometry Accelerator, Geometry Engine, and OpenGL are registered trademarks of Silicon Graphics, Inc.

The X Window System is a registered trademark of the Massachusetts Institute of Technology.

DEC is a registered trademark of Digital Equipment Corporation.

IBM and OS/2 are registered trademarks of International Business Machines.

Motif is a trademark of Open Software Foundation, Inc.

Windows NT is a registered trademark of Microsoft, Inc.

UNIX is a registered trademark of AT&T Bell Laboratories.

The contents of this publication are subject to change without notice.

Table of Contents/Schedule

Introduction

Break (elapsed time to here: roughly 1 1/2 hours)

Return from Break

End of Session (Question & Answers)

This page is intentionally left blank.

Speakers

- Tom McReynolds
- Kathleen Danielson

Objectives

- Become familiar with the capabilities of OpenGL
- Understand the order of operations, and the major libraries
- Know how to use viewing, lighting, shading, and hidden surface removal functionality
- Know how to draw images with OpenGL and understand some basic texture mapping capabilities
- See how code is written and compiled

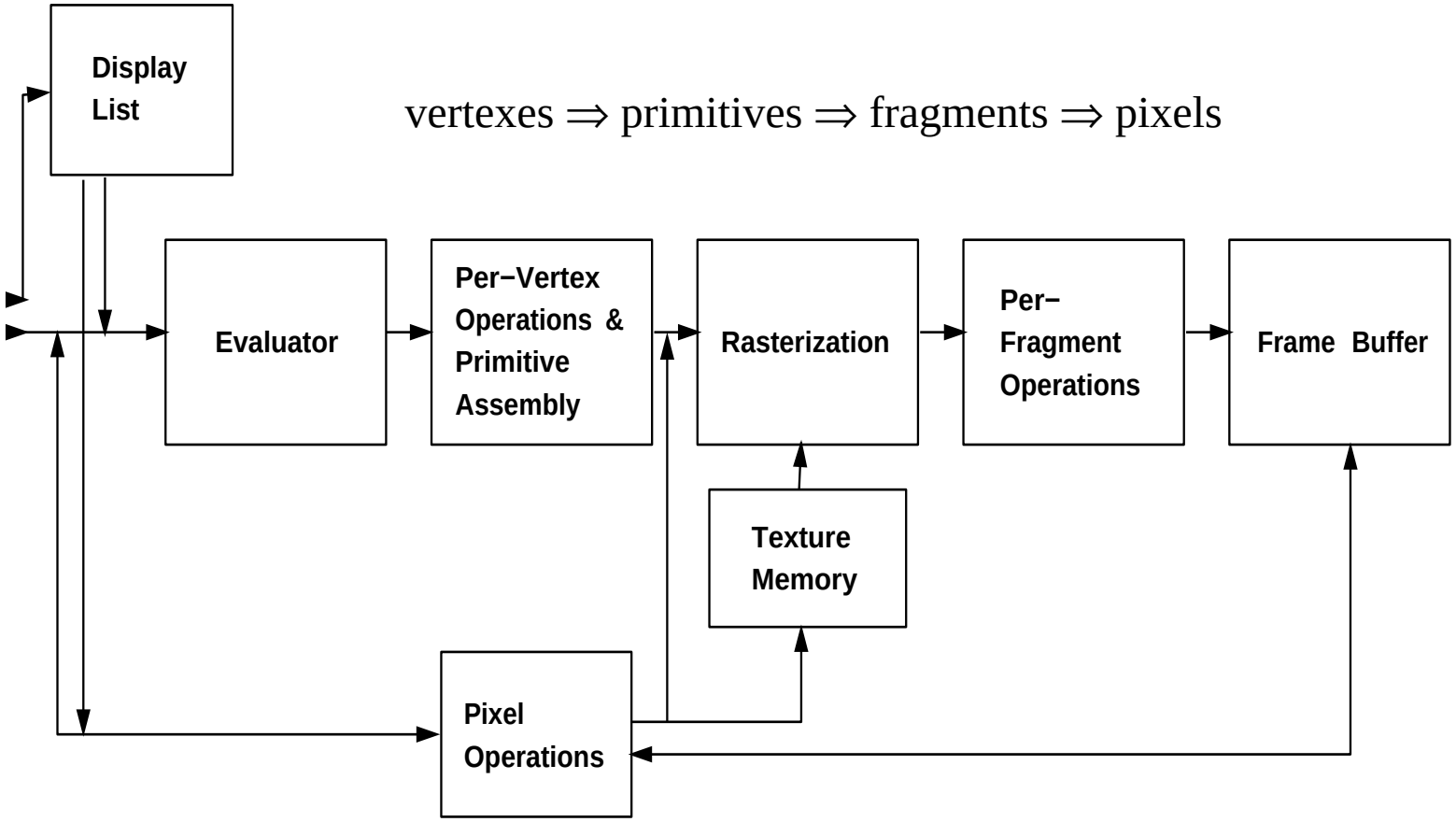
What is OpenGL?

- It's a low-level graphics rendering and imaging library
  - only includes operations which can be accelerated
- A layer of abstraction between graphics hardware and an application program
- An API to produce high-quality, color images of 3D objects (group of geometric primitives) and images (bitmaps and raster rectangles)
- A State Machine
- Window System and Operating System independent
  - use with X Window System® under UNIX®
  - or Microsoft Windows™ or Windows NT™
  - or IBM OS/2®
  - or Apple Mac OS

OpenGL State Machine

- Drawing Geometry and Clearing the Screen
- Points, Lines, and Polygons
- Images and Bitmaps
- Transformations
- Colors and Shading
- Blending and Antialiasing
- Lighting and Texturing
- Hidden Surface Removal
- Display Lists
- Fog and Depth Cueing
- Accumulation Buffer
- Stencil Planes
- Feedback and Selection
- Evaluators (NURBS)

OpenGL Data Flow



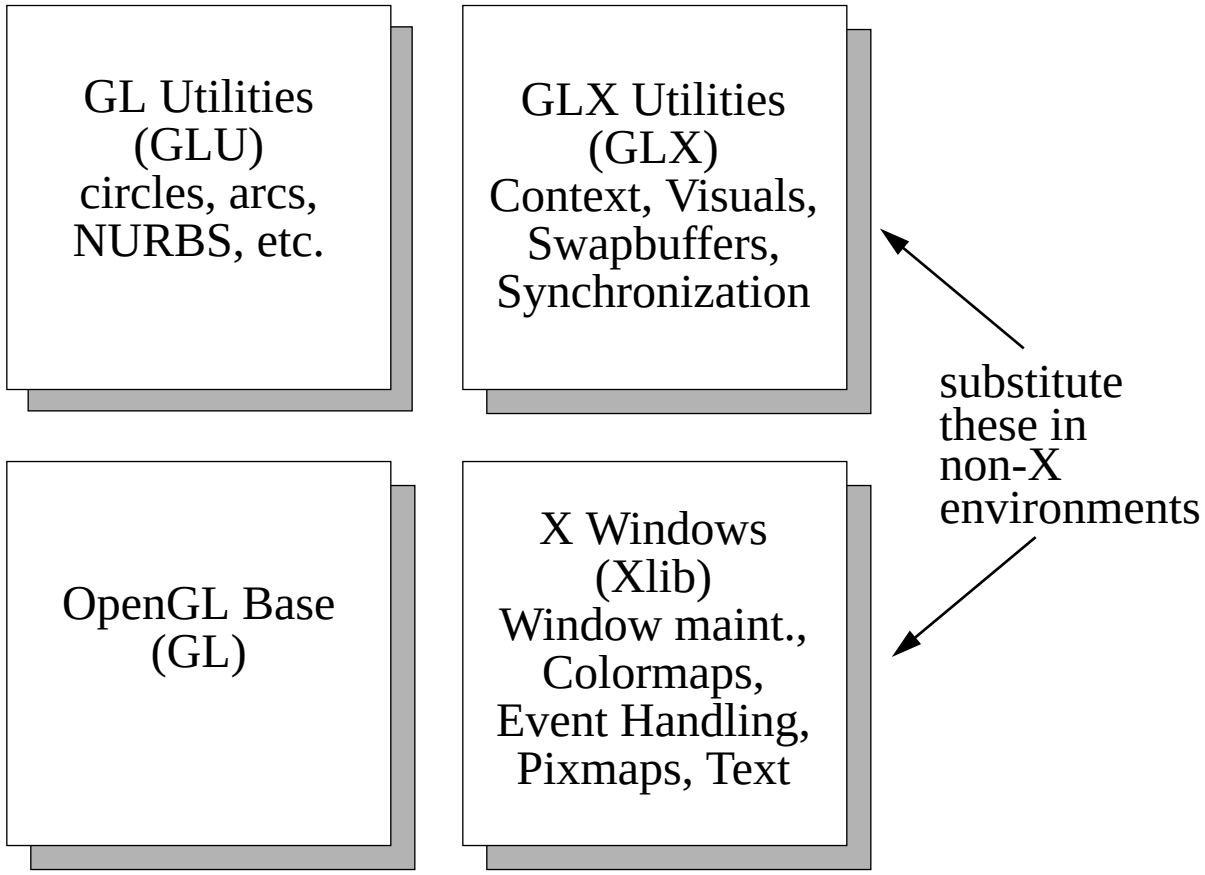
OpenGL Main Points

- Vertexes and images are fundamental primitives
- As a geometric primitive is drawn, each of its vertices is affected by the current “state” variables:
  - transformation matrices, color, lighting, texture, fog, rasterization, etc.
- Image primitives are also affected by state variables:
  - storage modes, transfer operations, zoom factor, and mapping through a look up table.
- All operations are “really” 3-D
- The frame buffer is a useful resource for hidden surface removal (depth buffer), motion blur and depth of field effects (accumulation buffer), and other effects.

OpenGL Main Points (continued)

- There is both client and server state
- Display lists are good for caching geometry, images and OpenGL state. They reside in the address space of the renderer (i.e., the *server*).
- "OpenGL doesn't do windows"
  - Relies on window system for window management, event handling, color map operations, etc.
  - Use Xlib, Motif, Microsoft Win32, IBM Presentation Manager, etc. for windowing operations.
- Conformance tests ensure consistency between implementations and prevents subsetting
- Flexible interface accommodates extensions

OpenGL API Hierarchy



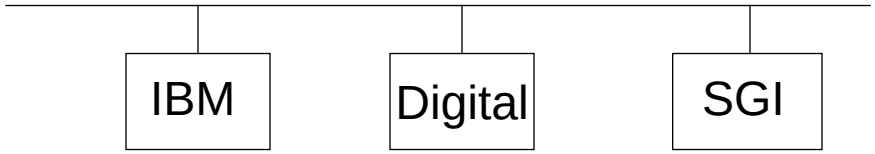
cc -o glx glx.c -lGLw -lGLU -lGL -lXm -lXext -lXt -lX11

Utility Library (GLU)

- Utility library is a set of commonly used graphics routines
  - built on top of OpenGL
  - *no* X server protocol is generated
- Quadric surface routines
  - spheres, cones, open cylinders, and tesselated disks for circles and arcs
- Polygonal surface routines
  - concave polygons, polygons with holes, self-intersecting polygons, etc.
  - control over tesselation tolerance and and which polygons are interior
- NURBS (with trimming)
  - can control sampling methods and tolerances
- Matrix and mipmap utilities

OpenGL extension to X (GLX)

- GLX extended servers have visuals which support OpenGL rendering
- Interoperability
  - GLX protocol allows heterogenous systems to interoperate



- What is a Mixed Model (X and OpenGL) program?
  - uses OpenGL and X Window System routines within the same program and inside the same window.
  - uses OpenGL for rendering
  - uses the X Window System (Xlib, widget set) for window, event, and color management, and user interface.

More on GLX

- GLX context holds OpenGL rendering state
- Can render OpenGL to pixmaps and windows
  - Window created through X
  - GLX Pixmap created from X pixmap; has associated X visual
- Must bind drawable to context before rendering
- Direct rendering
  - bypasses X and GLX protocol for rendering
  - must synchronize X and OpenGL command streams
  - rendering state resides in application process

GLUT (Graphics Library Utility Toolkit)

- GLUT is windowing utility library for OpenGL
  - hides windowing system dependencies
  - initialize and open window, including visuals
  - handle keyboard, mouse, and redraw events
  - enter event-driven loop, and render OpenGL

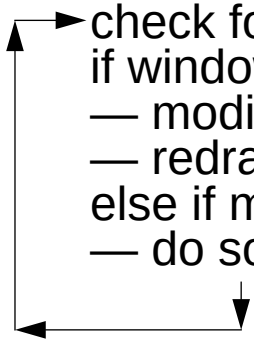
```
glutInit(&argc, argv); glutInitDisplayMode (modes);
glutCreateWindow(titleString);
glutReshapeFunc (func); glutDisplayFunc (func);
glutMainLoop ();
glutSwapBuffers();
```
make contents of back buffer visible

- more on GLUT in Mark Kilgard's class

Structure of a Typical Program

main:

find GL visual & create window
initialize GL states (e.g., viewing, color, lighting)
initialize display lists

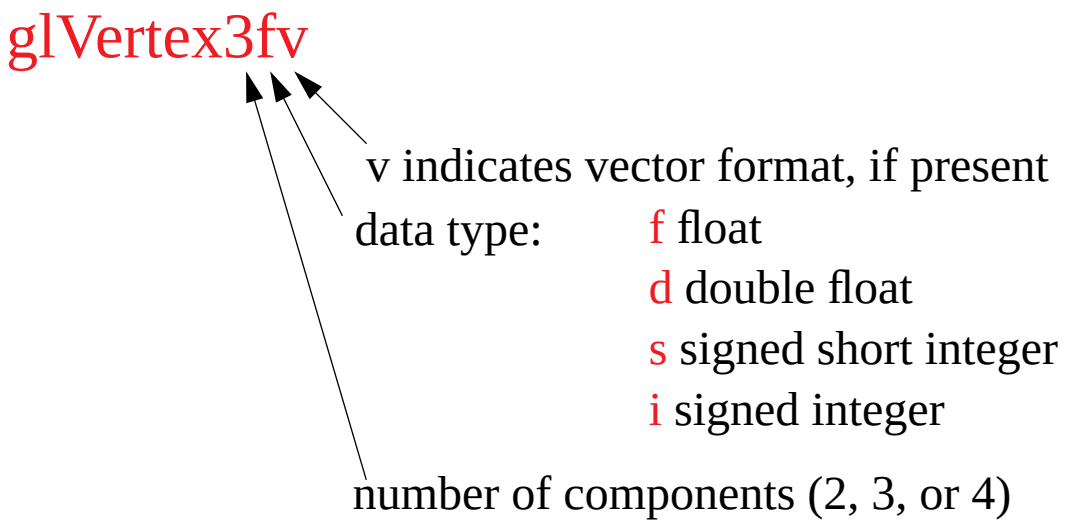


check for events (and process them)
if window event (window moved, exposed, etc.)
— modify viewport, if needed
— redraw
else if mouse or keyboard
— do something, e.g., change states & redraw

redraw:

clear screen (to background color)
change state(s), if needed
render some graphics
change more states
render some more graphics
....
swap buffers

OpenGL Command Syntax



- Other data types in OpenGL commands
  - b character
  - ub unsigned character
  - us unsigned short integer
  - ui unsinged integer
- scalar and vector formats

States

- `glEnable (GLenum capability)`
- `glDisable (GLenum capability)`
- `GLboolean glIsEnabled (GLenum cap)`
  - turn on and off OpenGL states
  - capability can be one of (partial list):

    `GL_BLEND` (alpha blending)

    `GL_DEPTH_TEST` (depth buffer)

    `GL_FOG`

    `GL_LIGHTING`

    `GL_LINE_SMOOTH` (line antialiasing)

Querying States

- `glGet*()`

```
glGetBooleanv(GLenum pname, GLboolean *params)
glGetIntegerv(GLenum pname, GLint *params)
glGetFloatv(GLenum pname, GLfloat *params)
glGetDoublev(GLenum pname, GLdouble *params)
```

man page is 14 pages long

number of color bits: `GL_ALPHA_BITS, GL_BLUE_BITS, GL_GREEN_BITS, GL_RED_BITS, GL_INDEX_BITS, GL_DEPTH_BITS, GL_ACCUM_*_BITS`

Drawing Geometry

- Delimit primitives with `glBegin()` and `glEnd()`
- To send down a vertex with current attributes, use `glVertex*()`
- Vertex attributes:
  - `glColor*()/glIndex*()` current vertex color
  - `glNormal*()` current vertex normal (lighting)
  - `glMaterial*()` current material property (lighting)
  - `glTexCoord*()` current texture coordinate
  - `glEdgeFlag*()` edge status (surface primitives)
- Types of primitives `glBegin (GLenum primitiveType):`
  - `GL_POINTS`
  - `GL_LINES GL_LINE_STRIP GL_LINE_LOOP`
  - `GL_POLYGON`
  - `GL_TRIANGLES GL_TRIANGLE_STRIP GL_TRIANGLE_FAN`
  - `GL_QUADS GL_QUAD_STRIP`

Drawing Geometry (continued)

- Example: Drawing a green, flat triangle strip:

```
glBegin (GL_TRIANGLE_STRIP);
glColor3f(0.f,1.f,0.f);
glNormal3f(0.f, 0.f, 1.f);
glVertex3f(-.5f, -.5f, -.5f);
glVertex3f(.5f, -.5f, -.5f);
glVertex3f(-.5f, .5f, -.5f);
glVertex3f(.5f, .5f, -.5f);
glEnd();
```

- glBegin - glEnd paradigm allows great flexibility in describing primitives.
- Any combination of color, normal, texture, material etc. information can be bound with any given vertex.
- This paradigm does incur a lot of function calls; use vertex array or display lists to reduce overhead

Vertex Arrays

- Create arrays of data with `gl*Pointer()` calls
- Array elements must have a fixed offset between elements
- Enable the arrays of data you want to use
  - `glEnableClientState(GLenum array);`
- Render an entire primitive from a set of arrays using `glDrawArrays(GLenum mode, GLint first, GLsizei count)`
  - `mode` defines type of primitive (same as `glBegin()`'s argument).
  - `first` and `count` used to subset arrays
- Render an entire primitive by indexing into arrays using `glDrawElements(GLenum mode, GLsizei count, GLenum type, GLvoid *indicies);`
  - `indicies` used to index elements in enabled arrays.
  - use to render primitives with shared verticies

Vertex Arrays (continued)

- Use `glArrayElement(GLint i)` to access the *i*th element of every enabled array; allows flexible use of enabled arrays
  - mix array and non-array data in the same Begin-End sequence.
- `glInterleavedArrays(GLenum format, Glsizei stride, GLvoid *pointer)` - high performance version for fixed format, interleaved geometry elements.
  - vertex and vertex attribute data packed in each array element in pre-defined sequence
  - elements don't have to be contiguous, but must be separated by a fixed offset
  - Example: format `C4F_N3F_V3F`, data is packed like

```
struct {
     GLfloat color[4]; /*color data*/
     GLfloat normal[3]; /*normal data*/
     GLfloat vertex[3]; /*vertex data*/
} C4F_N3F_V3Fpacking;
```

Display Lists

- Cache sequences of state and rendering commands
- Improve rendering and state change bandwidth
- Group constant sections of OpenGL to simplify code.
- Examples
  - pre-evaluate or pre-tesselate complex objects (NURBS, quadric objects, concave polygons, mipmaps, etc.)
  - can use to “name” textures (but texture object is better)
  - machine may have special display list memory, and/or accelerated display list traversal.
- *may* reside on the server, which can improve performance across network

```
glNewList (GLuint list, GLenum mode)
```

where mode is GL_COMPILE or GL_COMPILE_AND_EXECUTE

```
glEndList ()
glCallList (GLuint list)
```

Code: Immediate Mode vs. Display List

```
void display(void) {
        ...
    gluSphere(qobj, 1.0, 20, 20);
        ...
}
void gfxinit(void) {
    qobj = gluNewQuadric();
    gluQuadricDrawStyle(qobj, GLU_FILL);
        ...
}
```

```
void display(void) {
        ...
    glCallList(1);
        ...
}
void gfxinit(void) {
    qobj = gluNewQuadric();
    gluQuadricDrawStyle(qobj, GLU_FILL);
    glNewList(1, GL_COMPILE);
    gluSphere(qobj, 1.0, 20, 20);
    glEndList();
        ...
}
```

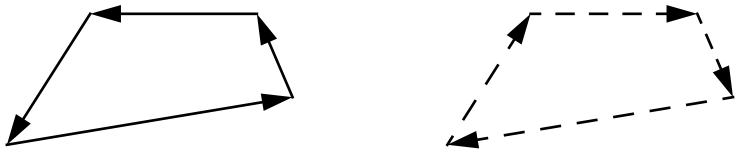
Display List Editing

- no display list editing, but `glCallList*()` calls are dynamically resolved; can delete and re-create list to change it

```
glNewList (1, GL_COMPILE);
    glIndexi (MY_RED);
glEndList ();
glNewList (2, GL_COMPILE);
    glScalef (1.2, 1.2, 1.0);
glEndList ();

glNewList (3, GL_COMPILE);
    glCallList (1);
    glCallList (2);
glEndList ();
    .
    .
glDeleteLists (1, 2);
glNewList (1, GL_COMPILE);
    glIndexi (MY_CYAN);
glEndList ();
glNewList (2, GL_COMPILE);
    glScalef (0.5, 0.5, 1.0);
glEndList ();
```

Culling

- turn on mode to eliminate rendering of front- or back-facing polygons
- reducing number of rendered polygons may increase performance
- concave surfaces, or surfaces with holes may show defects when culled
- front or back facing determined by orientation (winding) of polygons



```
glEnable (GL_CULL_FACE);
glCullFace (whichWay);
where whichWay is GL_FRONT or GL_BACK
glFrontFace (windingMode);
where windingMode is GL_CCW or GL_CW
```

Colors

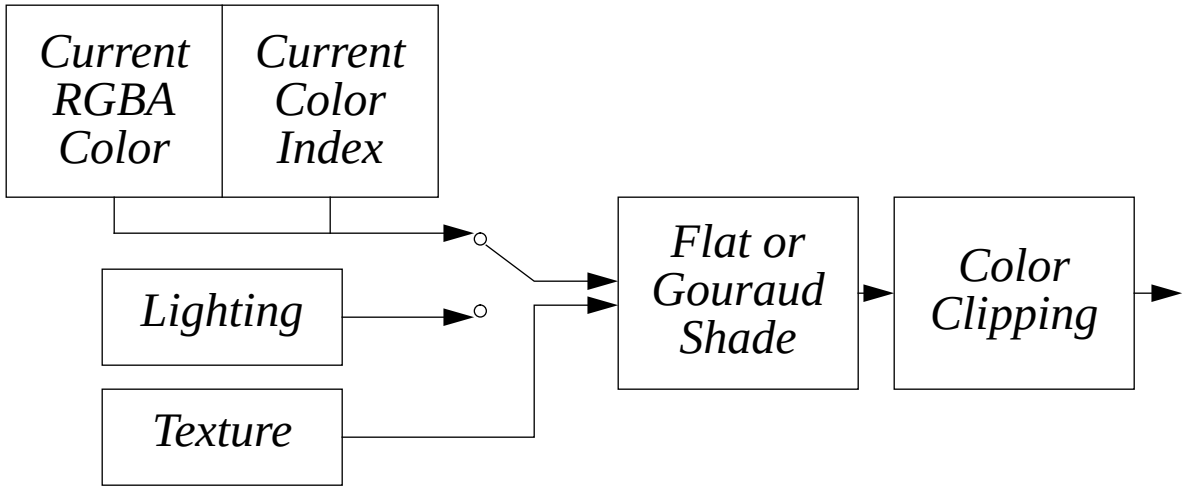
- RGBA mode, `glColor*()` is simplest way to change current color

```
GLfloat color[] = {1.f, 1.f, 0.f};
...
glBegin(GL_LINES);
glColor3f(0., 1., 1.); /*CYAN line*/
glVertex3f(100., 0., 0.); glVertex3f(0., 100., 0.);
glColor3fv(color); /*YELLOW line (glColor3fv faster)*/
glVertex3f(0., 0., 0.); glVertex3f(100., 100., 0.);
glEnd();
```

- In Color Index mode, `glIndex*()` specifies index in color look up table

	Red	Green	Blue
0	0	0	0
1	255	0	0
2			
⋮			
2^n-2			
2^n-1			

Processing of Colors



- Color is modal (RGBA or color index)
  - must be set upon window initialization
- Determine current color (RGBA)
  - if lighting is enabled, use lighting to compute color
  - else use currently set RGBA or color index value
  - then apply other color operations (blending, etc.)
- Loading color map (look up table) is a window system operation

Point, Line, and Polygon Attributes

- Point size

```
glPointSize (GLfloat size)
```

- Line width and stipple

```
glLineWidth (GLfloat size)
glLineStipple (GLint factor, GLushort pattern)
```

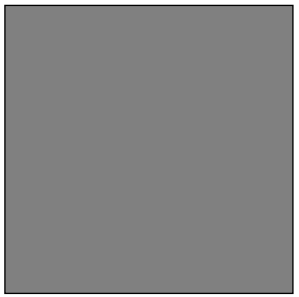
- Polygon stipple and shading

```
glPolygonStipple (const GLubyte *mask)
```

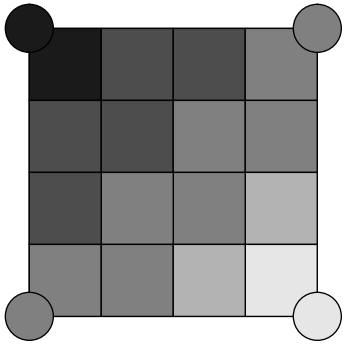
`glShadeModel (mode)` where `mode` is `GL_FLAT` or `GL_SMOOTH`

Primitives shaded with one color (flat) or a spectrum of adjacent colors (smooth)

Flat shading



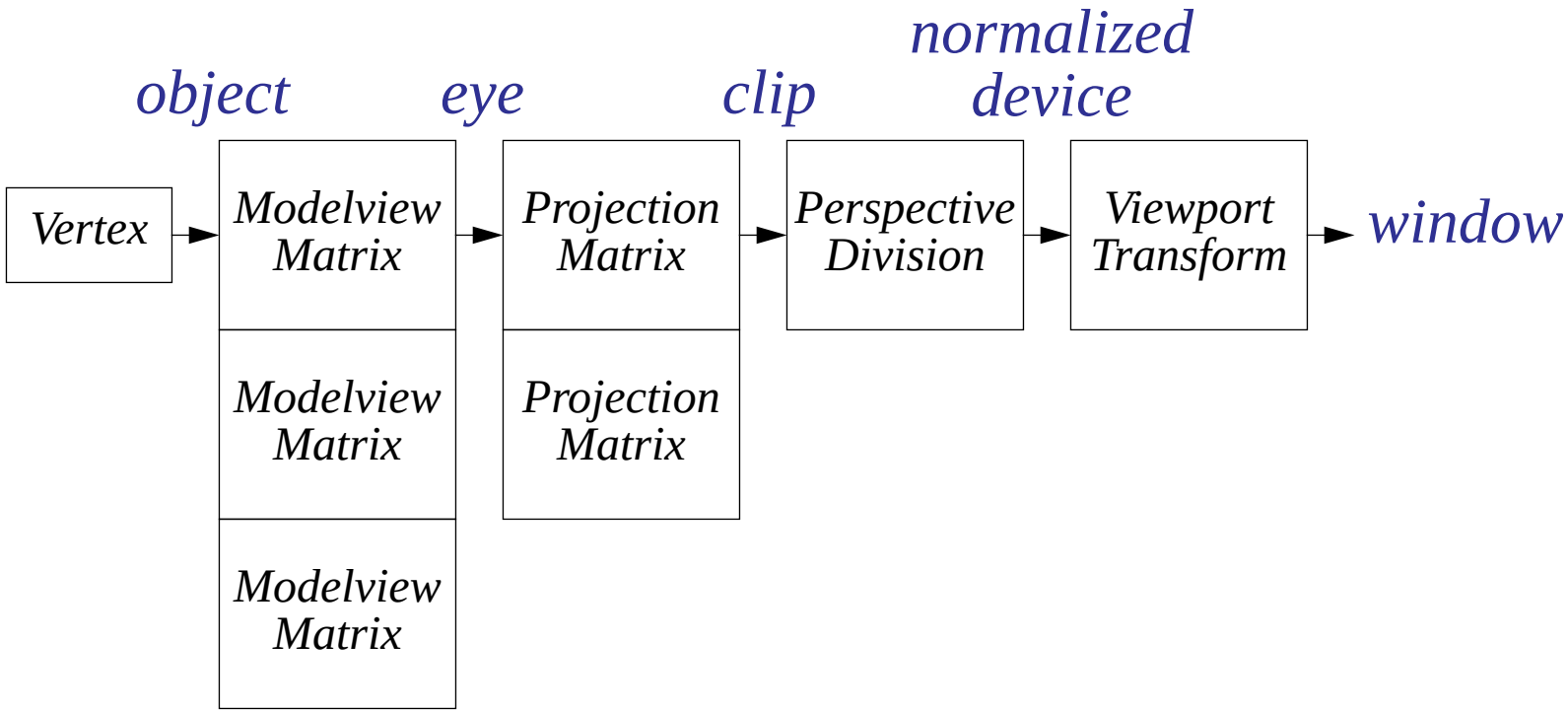
Smooth shading



Transformations and Coordinate Systems

- Four different coordinate systems
  - Object space - create object models
  - Eye space - lighting, some texture coordinate generation
  - Clip Coordinates- client and view clipping
  - Normalized Device Coordinates - -1 to 1 in x, y, z
  - Window Coordinates - x and y match window; z 0 to 1
- Normals are transformed with inverse transpose of vertex transforms.
- Clip, NDC and Window coordinates are left-handed
- Object and Eye coordinates are whatever you want, but OpenGL-provided projection matricies assume that they are right-handed.
- Lighting and Environment-mapping assume that viewport is at the origin, looking towards -Z

Transformation Flow




```
glMatrixMode(mode), glLoadIdentity(), glPushMatrix(),
glPopMatrix(), glLoadMatrix(), glMultMatrix()
```

Transformations

- Ready-to-Use Projection Matrices
  - perspective or orthographic parallel projection

```
glFrustum(left, right, bottom, top, near, far)
glOrtho(left, right, bottom, top, near, far)
gluPerspective(fovy, aspect, zNear, zFar)
gluOrtho2D(left, right, bottom, top)
```

- Projection matrix (using a camera model)

```
gluLookAt(eyex, eyey, eyez, centerx, centery, centerz,
     upx, upy, upz)
```

- Modeling (move model or eye coordinate system)

```
glTranslate{fd}(x, y, z)
glRotate{fd}(angle, x, y, z)
```
--arbitrary axis of rotation

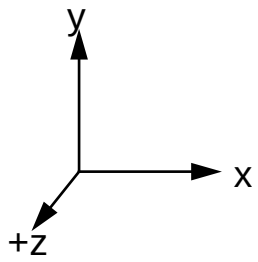
```
glScale{fd}(x, y, z)
```

- Window Coordinates

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

Using Transformations

- Using high-level routines (`glLoadIdentity, glRotate, glTranslate`, etc.) is easier and may be *faster*
- Projection transforms assume right-handed model and eye coordinates; they flip the Z axis to make clip left handled.



  - think of the znear and zfar arguments as positive distance from view point
- Be careful about the order of composited transformations
  - to do rotate -> scale -> translate operations, in order
  - `glLoadIdentity(); glTranslate(); glScale(); glRotate();`
  - matrix operations always post-multiply top of stack

Before We Look at Some Code

- Some other OpenGL and GLU routines

`glClearColor (r,g,b,a)` choose RGBA value for clearing color buffer

`glClear (bitfield)` set bitplane area of the viewport to currently selected values

```
qobj = gluNewQuadric(); gluQuadricDrawStyle(qobj,style);
gluSphere (qobj, rad, slice, stack);
```
render sphere

An OpenGL Program

```
#include <GL/glut.h>/*this includes the others*/

GLUquadricObj *qobj;

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (0., 1., 1.);
    gluSphere(qobj, /*radius*/ 1.,
          /*slices*/ 20,  /*stacks*/ 20);
    glutSwapBuffers();
}

void gfxinit(void)
{
    qobj = gluNewQuadric();
    gluQuadricDrawStyle(qobj, GLU_LINE);
}
```

```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective( /*field of view*/ 40.,
                    /*aspect ratio*/ (GLfloat)w /(GLfloat)h,
                    /*Z near*/ 1., /*Z far*/ 10.);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    gluLookAt(0., 0., 5.,/*eye is at (0,0,5)*/
        0., 0., 0., /*center is at (0,0,0)*/
        0., 1., 0.); /*up is in pos Y dir*/
    glTranslatef(0., 0., -1.);
}

void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutCreateWindow("simple");
    glutReshapeFunc (myReshape);
    glutDisplayFunc(display);
    gfxinit();
    glutMainLoop();
}
```

Lighting and Shading

- Lighting: computing color at a point using lights
- Shading: deciding where to make lighting calculations
- Three types of “light”: *ambient*, *diffuse*, *specular*
  - each type handled independently; lights emit them separately, primitives reflect them separately.
  - results are summed up to get primitive’s vertex color
- Phong lighting, *not* Phong shading
  - lighting calculated for each vertex normal
  - $\cos^n(\theta)$ (actually (N dot H)^n) used for `GL_SHININESS`
- To light primitives:
  - enable lighting
  - define lights
  - set light model and light parameters
  - define primitives with material properties

Lighting Properties

- Material Properties of Objects `glMaterial*()`
  - ambient
  - diffuse
  - specular and shininess
  - emission
- Light Source(s) `glLight*()`
  - color/brightness
  - position
  - local or infinite
  - attenuation (drop off)
  - spot (directional)
- Lighting Model `glLightModel*()`
  - global ambient light
  - local or infinite viewer
  - one or two-sided lighting

Lighting Code

- `glLight*(GL_LIGHT?, pname, params)` set light source parameters
  - *pname* is `GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_*, GL_*_ATTENUATION`
- `glMaterial(face, pname, params)` set material parameters
  - *face* is `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`
  - *pname* is `GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHININESS, GL_AMBIENT_AND_DIFFUSE, GL_EMISSION, GL_COLOR_INDEXES`
- `glLightModel*(pname, params)` set lighting model parameters
  - *pname* is `GL_LIGHT_MODEL_AMBIENT, GL_LIGHT_MODEL_LOCAL_VIEWER, GL_LIGHT_MODEL_TWO_SIDE`
- `glNormal*()`or some automated way to generate normals
- `glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);`

Lighting Program

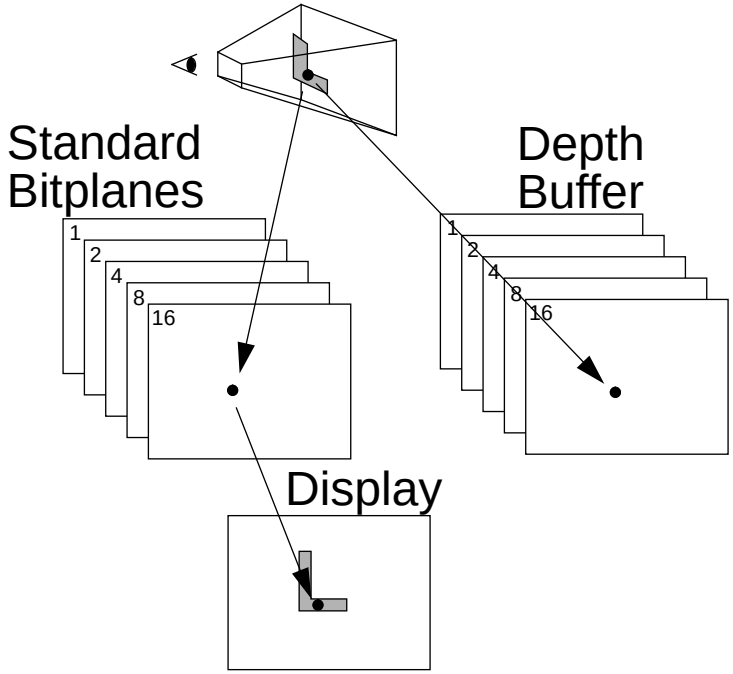
```
#include <GL/glut.h>
#include <stdio.h>

GLfloat mat_diffuse[] = {0.25, 0.25, 1., 0.};
GLfloat mat_specular[] = {1., 1., 1., 0.};
GLfloat light_position[] = {10., 10., 20., 1.};
GLUquadricObj *qobj;

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    gluSphere(qobj, /*radius*/ 1., /*slices*/ 20,
              /*stacks*/ 20);
    glPopMatrix();
    glutSwapBuffers();
}
```

```
void gfxinit(void)
{
    qobj = gluNewQuadric();
    gluQuadricDrawStyle(qobj, GLU_FILL);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 25.);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(40., (GLfloat)w/(GLfloat)h,
                   1., 10.);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    gluLookAt(0., 0., 5., 0., 0., 0., 0., 1., 0.);
    glTranslatef(0., 0., -1.);
}
```

```
void keyFunc(unsigned char key, int x, int y)
{
     if(key == 27) exit(0);
}

void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
    glutCreateWindow("light");
    glutReshapeFunc (myReshape);
    glutKeyboardFunc(keyFunc);
    glutDisplayFunc(display);
    gfxinit();
    glutMainLoop();
}
```

Hidden Surface Removal

- Depth (Z) Buffer
  - depth (Z) value stored for each pixel
  - pixel by pixel comparisons, implicit Z sort



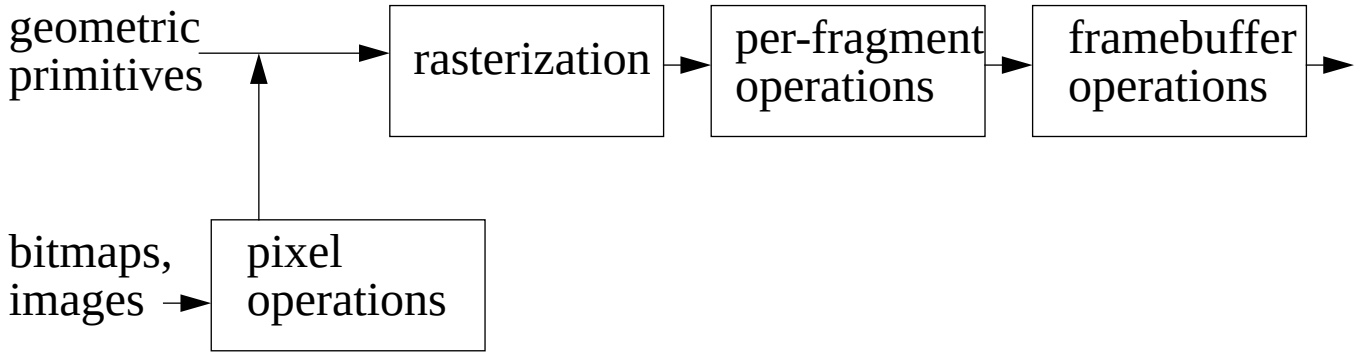
- Get a visual which supports the depth buffer
- Enable depth testing `glEnable(GL_DEPTH_TEST);`

Polygon Offset

- Allows polygon “decals” without running into depth buffer problems
- Can also be used to draw clean polygon edges on surfaces.
- Offsets polygon in Z direction.
- `glPolygonOffset(GLfloat factor, GLfloat units)`
  - *factor* offsets as a function of polygon slope
  - *units* adds a constant offset
- Factor scales slope in window coordinates, Z ranges from 0 to 1
- Units scales a constant offset; 1 is minimum Z displacement to guarantee polygon separation.

Imaging

- OpenGL was designed for both imaging and geometry
- Can take advantage of geometry operations for warping, projection
- Fundamental unit is pixel fragment
- Orthogonality of operations treats each fragment the same, whether generated from geometry, read from the framebuffer, or read from the host



Drawing Bitmaps and Images

- Bitmap (a single bit per pixel)
  - for characters in fonts

`glRasterPos*()` specifies a position for a bitmap

`glBitmap()` renders a bitmap

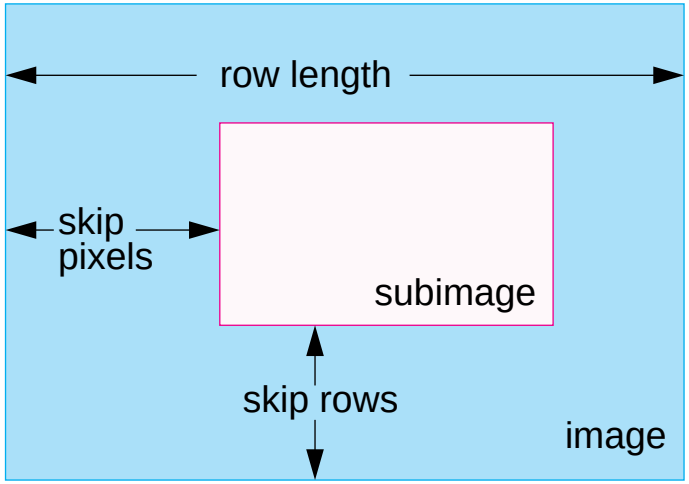
- Image (typically many bits of data per pixel)

`glReadPixels()`, `glDrawPixels()`, and `glCopyPixels()` manipulate rectangles of pixel data

  - may be zoomed
  - pixel storage, mapping and transfer modes (e.g., good for endian reversal)

Pixel Operations

- DrawPixels, ReadPixels and CopyPixels
- PixelStorage Modes
  swap bytes
  lsb first
  skip rows
  skip pixels
  alignment
  row length
- Pixel Transfer Modes
  map color
  map stencil
  index shift
  index offset
  per channel scale
  per channel bias
- Pixel Zoom



Histogram and Convolution Extensions

- Histogram
  - counts occurrences of specific color component
  - tracks minimum and maximum component values
  - optionally, discard data after histogram operation
- Convolution
  - 1 and 2 dimensional convolution
  - for drawing, reading, copying and texture definition
  - general and separable filters

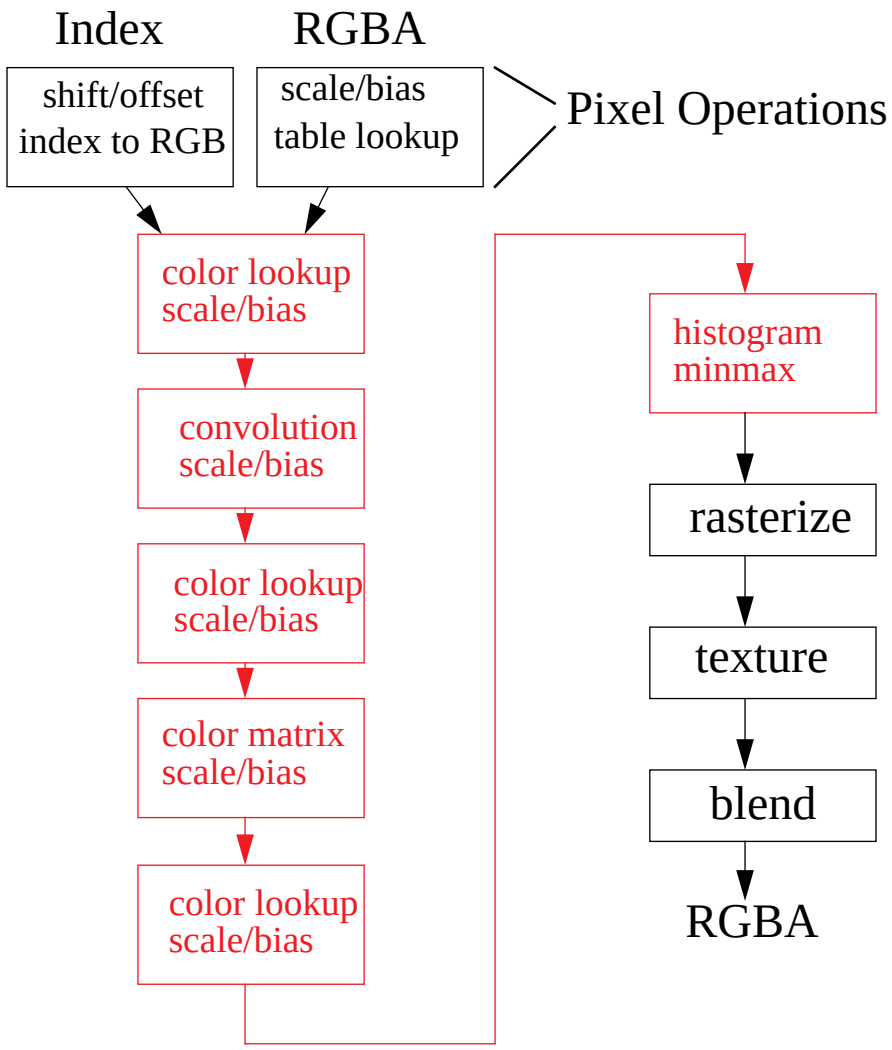
Color Table and Color Matrix Extensions

- Color Table
  - one dimensional table lookup
  - color table is processed through full pixel path
  - can add table in 3 places:
    - after pixel operations
    - after histogram and convolve
    - after color matrix
- Color Matrix
  - adds 4x4 matrix to pixel transfer path
  - operates on RGBA pixel groups
  - can be used to reassign and duplicate color components
  - scale and bias after matrix operation

Blend Extensions

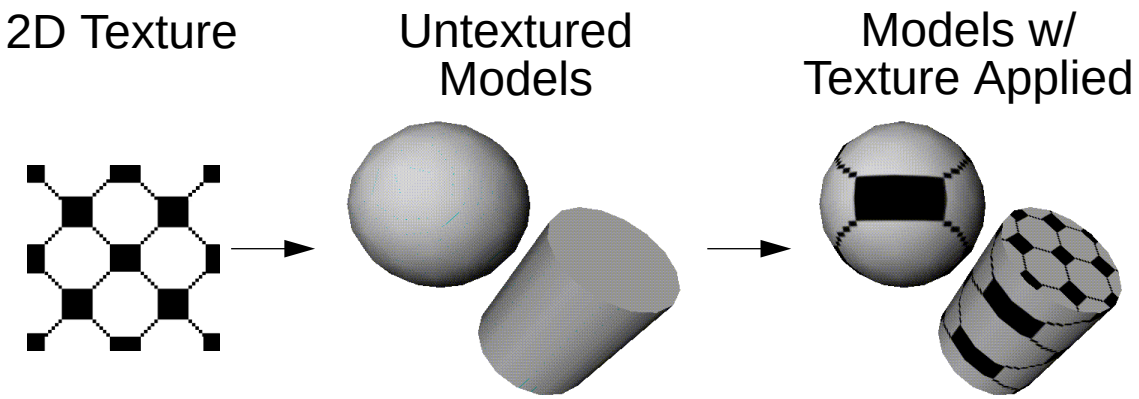
- Blend Color
  - defines constant color for use in blending equations
  - blending two RGBA images without specifying alpha channel per pixel
- New blending operators
  - minimum of source and destination colors
  - maximum of source and destination colors
  - subtract: $(C_s * S) - (C_d * D)$
  - reverse subtract: $(C_d * D) - (C_s * S)$

Imaging Pipeline with Extensions



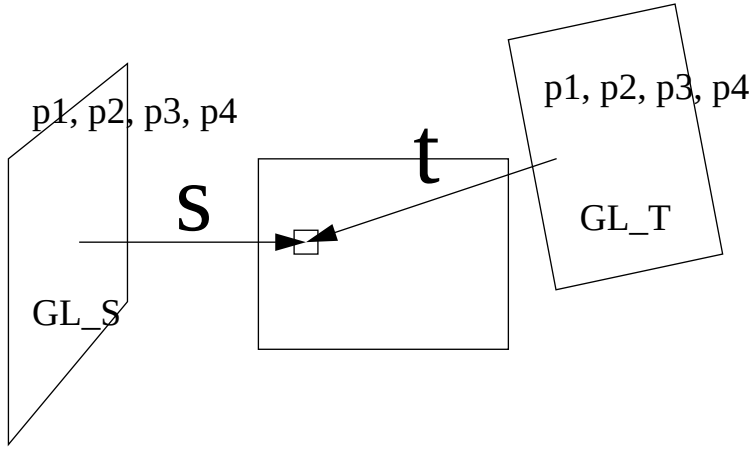
Texture Mapping

- Bringing together imaging and geometry
- Map a 1D or 2D Image onto a 2D or 3D object
- Texture Mapping has many Uses



Controlling Texturing

- Use `glTexGen*()` to automatically generate texture coordinates in primitives.
  - convenience feature, can use `glTexCoord*()` calls.
  - `GL_OBJECT_LINEAR` - attach texture to objects
  - `GL_EYE_LINEAR` - attach texture to "world"
  - `GL_SPHERE_MAP` - object reflects surrounding texture



Controlling Texturing (cont.)

- `glTexImage*()` is used to load in textures.
  - also has an optional texture border
- `glTexParameter*()` controls filtering and wrapping when applying textures.
  - `GL_TEXTURE_MIN_FILTER` - texels smaller than pixels (texture far away)
  - `GL_TEXTURE_MAG_FILTER` - texels bigger than pixels (texture close up)
- `glTexEnv*()` how texels and texture environment color are combined with primitive's existing pixel values
  - `GL_MODULATE` - scale pixel with texel
  - `GL_DECAL` - replace pixel with texel RGB (blend alpha)
  - `GL_REPACE` - replace pixel with entire texel
  - `GL_BLEND` - mix pixel with texture environment color using texel to control ratio

New Texture Functionality

- `glSubTexLoad()` - allows you to replace parts of any given texture.
- `glCopyTexture()` - directly copy a section of frame buffer into a given texture.
  - may be faster than `glReadPixel*()` -> `glTexImage*()`
  - good for multi-pass algorithms
- Texture Object - a way to name textures, similar to display lists. Can prioritize textures.
  - Texture object is texture named by an unsigned integer.
  - `glBindTexture()` connects number to `GL_TEXTURE_1D` or `GL_TEXTURE_2D`.
  - now use regular texture functions.
  - used to switch between multiple textures, allows texture re-use without having to destroy and re-create.

Texturing Program

```
#include <GL/glut.h>
#include <stdio.h>

GLfloat mat_diffuse[] = {0.25, 0.25, 1., 0.};
GLfloat mat_specular[] = {1., 1., 1., 0.};
GLfloat light_position[] = {10., 10., 20., 1.};
GLUquadricObj *qobj;

/*Create 1D texture--white, with red stripes*/
#define   stripeImageWidth 32
GLubyte stripeImage[4*stripeImageWidth];

void makeStripeImage(void)
{
    int j;
    for (j = 0; j < stripeImageWidth; j++) {
          stripeImage[4*j] = 255;
          stripeImage[4*j+1] = (j < 8) ? 0: 255;
          stripeImage[4*j+2] = (j < 8) ? 0:255;
          stripeImage[4*j+3] = 0;
    }
}
```

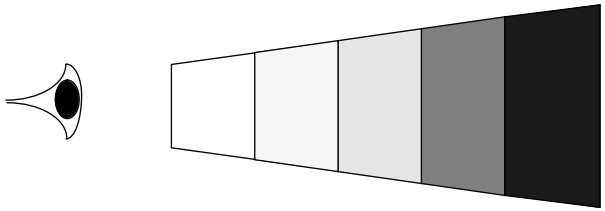
```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    gluSphere(qobj, /*radius*/ 1.,
          /*slices*/ 20,  /*stacks*/ 20);
    glPopMatrix ();
    glutSwapBuffers();
}
/*define 'plane' for texture coord generation*/
GLfloat sgenparams[] = {1.5, 1.5, 1.5, 0.};

void gfxinit(void)
{
    qobj = gluNewQuadric();
    gluQuadricDrawStyle(qobj, GLU_FILL);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
```

```
    makeStripeImage();
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
   glTexImage1D(GL_TEXTURE_1D, 0, 4, stripeImageWidth, 0, GL_RGBA,
                    GL_UNSIGNED_BYTE, stripeImage);

    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGenfv(GL_S, GL_OBJECT_PLANE, sgenparams);

    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_1D);
    glEnable(GL_CULL_FACE); /*default:  CCW front; cull BACK*/
}

void keyFunc(unsigned char key, int x, int y)
{
     if(key == 27) exit(0);
}
```

```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40., (GLfloat)w/(GLfloat)h,
                   1., 10.);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
   gluLookAt(0., 0., 5., 0., 0., 0., 0., 1., 0.);
    glTranslatef(0., 0., -1.);
}

void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
                        GLUT_DEPTH);
    glutCreateWindow("texture");
    glutReshapeFunc (myReshape);
    glutKeyboardFunc(keyFunc);
    glutDisplayFunc(display);
    gfxinit();
    glutMainLoop();
}
```

Atmospheric Effects

- Density Changes with Distance
- Fog, haze, smoke, and smog are all the same



- Also use for “depth cueing”
- Linear or non-linear fog math

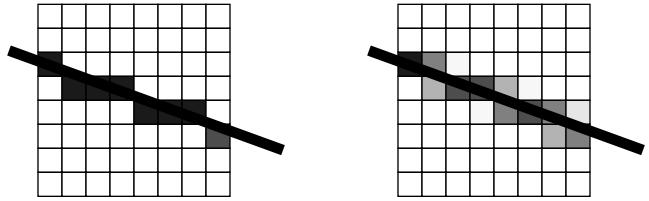
```
glEnable(GL_FOG);
glFogi(GL_FOG_MODE, GL_*);
```

Antialiasing

- Cures the "Jaggies"; creates smooth points & lines

aliased antialiased

- Lines redrawn 2 or 3 times
- Pixel averaging algorithm



aliased antialiased

`glEnable (GL_POINT_SMOOTH), glEnable (GL_LINE_SMOOTH)`

- in RGBA mode, use alpha values
- in color index mode, use last 4 bits of index

Per Fragment Operations

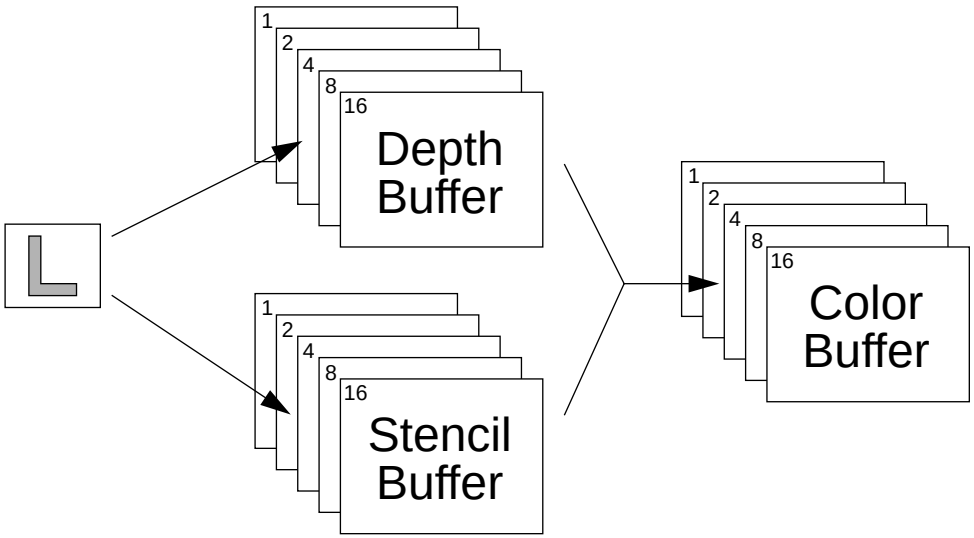
- Scissor Test
  - Is fragment inside scissor rectangle?
- Alpha Test
  - compare reference value with fragment's alpha value
  - accept/reject fragment based on results of test
  - must be in RGBA mode
- Stencil Test
  - compare reference value with value in stencil buffer
  - pixel in stencil buffer is modified according to the results of the test
- Depth test
  - compare z value with value in depth buffer

Per Fragment Operations (continued)

- Blending
  - combine incoming RGBA values with values in the framebuffer
  - must be in RGBA mode
  - if no alpha buffer then alpha value of one is used
- Dithering
  - improves color resolution on systems without many bitplanes
- Logic Operations
  - same set of operations as X
  - must be in color index mode in OpenGL 1.0
  - can do logic ops on RGBA components in OpenGL 1.1
  - overrides blending

Stencil Planes

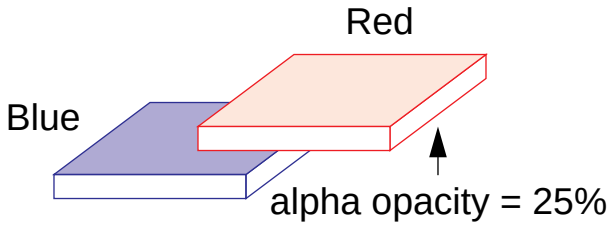
- Additional Pixel Test
- Uses
  - Pixel Masking
  - Capping Solid Geometry



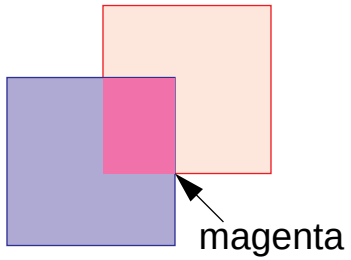

```
glStencilFunc(func, ref, mask);
glStencilOp(fail, zfail, zpass);
```

Alpha Blending

- Translucency effects
- 0 % to 100 % opacity



Side View



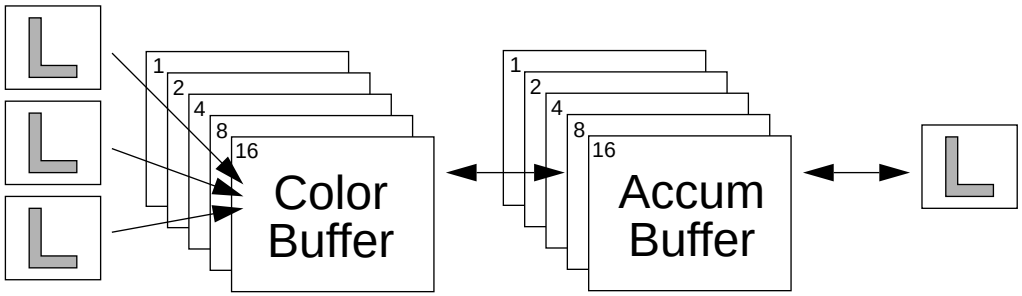
Top View

- order of drawing is important
- don’t need alpha buffer to do translucency

```
glBlendFunc (GLenum srcFactor, GLenum destFactor)
```

Accumulation Buffer

- Store Multiple Images
  - multiple exposures
- Uses
  - motion blur
  - depth of field (out of focus)
  - scene antialiasing




```
glAccum (op, value);
where op is GL_ACCUM, GL_LOAD, GL_ADD, GL_MULT,
or GL_RETURN
```

Feedback & Selection

- Usually, transformed vertices and colors generate image in the display buffer
- In feedback mode, the transformed values are returned to the application in an array
- Special tricks for picking objects

Evaluators/NURBS

- Support for polynomials (for splines or surfaces)
- Evaluators are foundation for NURBS
- NURBS supported in Utility Library (glu)
  - Non-Uniform Rational B-Splines
  - parametric, curved surfaces

OpenGL 1.1 features

- Polygon Offset: lines, points and polygons which lie in the same plane can be rendered without interaction
- Vertex Array: geometric primitives can be defined with fewer subroutine calls
- Texture Object: facilitates texture caching and allows textures to be prioritized
- Subtexture definition: redefine a portion of an already existing texture
- Copy Texture: load texture images from the framebuffer
- Texture: internal formats for textures and REPLACE texture environment; proxy texture allows texture memory to be more fully utilized
- Blend logic op: apply logic operations (e.g., XOR) to fragments in RGBA mode

GLX 1.1 and GLX 1.2

- GLX 1.1
  - glXQueryExtensionsString indicates which extensions are supported on the connection
  - can also query vendor name, version and extension string for client library and server
- GLX 1.2
  - supports OpenGL 1.1
  - glXGetCurrentDisplay returns display for current context
  - texture objects can be shared by contexts that share display lists

GLU 1.2

- Better polygon tesselator
- Handles self-intersecting and co-incident vertices
- Important for rendering Type 1 fonts

Extensions

- Extensions provide proof of concept for new OpenGL features
- Many imaging and texturing extensions
  - histogram, convolve, color tables, additional blend modes, ...
  - texture3D, texture lookup table, ...
- Extensions with EXT suffix are supported by at least 2 OpenGL vendors
- Extensions with vendor-specific suffix are typically only supported by that vendor

Using Extensions and Versions

- Check library entry points at compile time

```
#ifdef GL_EXT_texture3D
#ifdef GL_VERSION_1_1
```

- Make sure renderer supports it at run-time

```
str = glGetString(GL_EXTENSIONS);
str = glGetString(GL_VERSION);
```

- For GLX, extensions are only supported if the version is 1.1 or greater

```
glXQueryVersion(dpy, &major, &minor);
str = "";
#ifdef GLX_VERSION_1_1
if (minor > 0 || major > 1)
str = glXQueryExtensionsString(dpy,screen);
#endif
```

Summary

- OpenGL is a low-level API, providing mechanism, but not enforcing policy, to accomplish sophisticated rendering and imaging.
- “State machine” is the OpenGL metaphor
- OpenGL integrates imaging and geometry
- OpenGL is network interoperable
- OpenGL is window system independent

For more information

- Usenet Group comp.graphics.api.opengl
- ftp OpenGL Specification and Man Pages from sgigate.sgi.com
  - in ~ftp/pub/opengl/doc directory
  - {mangl,manglu,manglx}.tar.Z and specs.tar.Z files
- WWW URL http://www.sgi.com/Technology/openGL/
  - many vendors have web sites too!
- Addison-Wesley Publishing

  OpenGL Programming Guide (ISBN 0-201-63274-8)
  - authors: J. Neider, Davis, and Woo

  OpenGL Reference Manual (ISBN 0-201-63276-4)
  - author is OpenGL ARB (Architectural Review Board)

OpenGL Frequently Asked Questions (FAQ)

You may also want to see the OpenGL Web Page:

WWW URL http://www.sgi.com/Technology/openGL/

* marks recently modified answers or new questions

----- PART 1 - general questions.

Q_1_01: How do I submit changes or additions to this FAQ?

Q_1_02: What is OpenGL?

Q_1_03: Where are World Wide Web sites with information about OpenGL?

Q_1_04: What does the .gl or .GL file format have to do with OpenGL?

Q_1_05: What documentation is available for OpenGL? (A bibliography of OpenGL documents is listed here.)

Q_1_06: Where can I get the OpenGL specification?

Q_1_07: Which vendors are licensing OpenGL?

Q_1_08: What OpenGL implementations are available?

Q_1_09: Does Windows 95 support OpenGL?

Q_1_10: What interest is there for OpenGL in Japan?

----- PART 2 - OpenGL governance and the ARB.

Q_2_01: Who needs to license OpenGL? Who doesn't?

Q_2_02: What are the conformance tests?

Q_2_03: What is Silicon Graphics policy on "free" implementations of APIs which resemble the OpenGL API?

Q_2_04: What is Mesa 3D and where can I get it?

Q_2_05: How does a university or research institution acquire access to OpenGL source code?

Q_2_06: How is a commercial license acquired?

Q_2_07: How is the OpenGL governed? Who decides what changes can be made?

Q_2_08: Who are the current ARB members?

Q_2_09: What is the philosophy behind the structure of the ARB?

Q_2_10: How does the OpenGL ARB operate logistically? When does the ARB have meetings?

Q_2_11: How do additional members join the OpenGL ARB?

Q_2_12: So if I'm not a member of the ARB, am I shut out of the decision making process?

Q_2_13: Are ARB meetings open to observers?

Q_2_14: What benchmarks exist for OpenGL?

----- PART 3 - technical questions.

Q_3_01: Where can I find OpenGL source code examples?

For instance, where is an example which combines OpenGL with Motif, using the Motif widget?

Q_3_02: How do I contribute OpenGL code examples to a publicly accessible archive?

Q_3_03: What is the GLUT toolkit? Where do I get it?

Q_3_04: What is the relationship between IRIS GL and OpenGL?

Is OpenGL source code or binary code compatible with IRIS GL?

Q_3_05: Why should I port my IRIS GL application to OpenGL?

Q_3_06: How much work is it to convert an IRIS GL program to OpenGL?

What are the major differences between them?

Q_3_07: When using Xlib, how do I create a borderless window?

Q_3_08: How do I switch between single buffer and double buffer mode?

Q_3_09: On my machine, it appears that glXChooseVisual is only able to match double-buffered visuals. I want to have more bits of color resolution, so how do I render in single buffer mode?

Q_3_10: I’ve got a 24-bit machine, but my OpenGL windows are not using

the full color resolution. What's wrong? My program looks fine on one machine, but the depth buffer doesn't work on another. What's wrong?

Q_3_11: What information is available about OpenGL extensions?

Q_3_12: How do I make shadows in OpenGL?

Q_3_13: How can I use 16 bit X fonts?

Q_3_14: What's in the new GLU 1.2 tesselator?

Q_3_15: Why is my glDrawPixels (or glCopyPixels or glReadPixels) slow?

Q_3_16: How can I used OpenGL with Tcl/Tk?

Q_3_17: OpenGL pixel-exact rasterization.

Q_3_18: Saving OpenGL screen output.

Q_3_19: No Logicop in RGB for OpenGL

Q_3_20: Why does the raster position get clipped by the viewing volume?

------ Q_1_01: How do I submit changes or additions to this FAQ?

A: To request changes or additions, please send e-mail to the FAQ maintainer. See the “Reply-To:” field in the header for the e-mail address.

OpenGL licensees may want to contribute information to the question: “What OpenGL implementations are available?” That space is available for any company who wishes to state status reports, release dates, contact names and phone numbers, or other information for their OpenGL implementation.

It is asked that this information be relatively brief. Also, for the sake of civility, each implementor is asked not to make comparisons of their implementations against others.

Note that although a Silicon Graphics employee maintains this FAQ, Silicon Graphics does not speak for any other company, nor does it uphold the veracity of anyone else’s information.

------ Q_1_02: What is OpenGL?

A: OpenGL(R) is the software interface for graphics hardware that allows graphics programmers to produce high-quality color images of 3D objects. OpenGL is a rendering only, vendor neutral API providing 2D and 3D graphics functions, including modelling, transformations, color, lighting, smooth shading, as well as advanced features like texture mapping, NURBS, fog, alpha blending and motion blur. OpenGL works in both immediate and retained (display list) graphics modes. OpenGL is window system and operating system independent. OpenGL has been integrated with Windows NT and with the X Window System under UNIX. Also, OpenGL is network transparent. A defined common extension to the X Window System allows an OpenGL client on one vendor’s platform to run across a network to another vendor’s OpenGL server.

------ Q_1_03: Where are World Wide Web sites with information about OpenGL?

A: OpenGL -- The Integration of Windowing and 3D Graphics WWW URL http://hertz.eng.ohio-state.edu/~hts/opengl/article.html Maintained by Harry Shamansky.

OpenGL WWW Center WWW URL http://www.sgi.com/Technology/openGL/ Maintained by Thomas McReynolds <tomcat@sgi.com>.

IBM WWW Center for OpenGL WWW URL http://www.austin.ibm.com/software/OpenGL

Template Graphics Software WWW Center for OpenGL WWW URL http://www.sd.tgs.com/~template/Products/opengl.html

Microsoft Developer Network OffRamp Web Server WWW URL http://www.microsoft.com

Portable Graphics, Inc. WWW URL http://www.portable.com/opengl/oglndx.htm

------ Q_1_04: What does the .gl or .GL file format have to do with OpenGL?

A: .gl files have nothing to do with OpenGL, but are sometimes confused with it. .gl is a file format for images, which has no relationship to IRIS GL or OpenGL.

------ Q_1_05: What documentation is available for OpenGL?

A: A 2 volume set, The OpenGL Technical Library (The OpenGL Programming Guide and The OpenGL Reference Manual) is published by Addison-Wesley. The ISBN numbers for both English and Japanese versions are listed below. You can purchase the books in extremely large volume by calling Addison-Wesley (+1-617-944-3700).

What follows is a bibliography of articles, books, and papers written about OpenGL.

Books (in English)

Neider, Jackie, Tom Davis, and Mason Woo, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1, Addison-Wesley, Reading, Massachusetts, 1993 (ISBN 0-201-63274-8).

OpenGL Architecture Review Board, OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1, Addison-Wesley, Reading, Massachusetts, 1992 (ISBN 0-201-63276-4).

Magazine articles

Bruno, Lee. “Graphics Users Debate Three Hot Topics,” Open Systems Today, December 12, 1994, p. HP3, HP8.

Bruno, Lee. “Sun Continues to Resist OpenGL Tide,” Open Systems Today, November 28, 1994, p. SF1, SF5-6.

Davis, Tom. “Moving to OpenGL,” IRIS Universe, Number 25, Summer, 1993.

Deffeyes, Suzy and John Spitzer. “OpenGL on OS/2”, OS/2 Developer Magazine, Nov/Dec 94, pages 34-45.

Glazier, Bill. “The ‘Best Principle’: Why OpenGL is emerging as the 3D graphics standard,” Computer Graphics World, April, 1992.

“Industry group pushing 3-D graphics standard,” Computer Design, July, 1994, p. 50, 52.

Karlton, Phil. “Integrating the GL into the X environment: a high performance rendering extension working with and not against X,” The X Resource: Proceeding of the 6th Annual X Technical Conference, O’Reilly Associates, Issue 1, Winter, 1992.

Kilgard, Mark, Simon Hui, Allen Leinwand, and Dave Spalding. “X Server Multi-rendering for OpenGL and PEX,” The X Resource Proceedings of the 8th Annual X Technical Conference, O’Reily and Associates, Sebastopol, California, January 1994.

Kilgard, Mark J. “OpenGL & X: An Introduction,” The X Journal. November-December, 1993, page 36-51.

Kilgard, Mark J. “Using OpenGL with Xlib,” The X Journal. January-February, 1994, page 46-65.

Kilgard, Mark J. “Using OpenGL with Motif,” The X Journal. July-August, 1994.

“OpenGL Programs a New Horizon for Sun,” SunWorld, January, 1994, page 15-17.

Prosise, Jeff. “Advanced 3-D Graphics for Windows NT 3.5: Introducing the OpenGL Interface, Part I,” Microsoft Systems Journal, October, 1994, Vol. 9, Number 10, pages 15-29.

Prosise, Jeff. “Advanced 3-D Graphics for Windows NT 3.5: Introducing the OpenGL Interface, Part II,” Microsoft Systems Journal, November, 1994, Vol. 9, Number 11.

Prosise, Jeff. “Understanding Modelview Transformations in OpenGL for Windows NT,” Microsoft Systems Journal, February, 1995, Vol. 10, Number 2.

Japanese language magazine articles and books

“Interview with Masamichi Tachi about OpenGL_Japan,” Nikkei Computer Graphics, 3/1995, p. 56-57.

Matsumoto, Masayuki, PIXEL, “OpenGL, A 3D Graphics Standard”, 10/1994, p. 138-145.

Matsumoto, Masayuki, Toragi Computer, “A introduction to OpenGL for PC users”, 11/1994, p. 147-150.

Neider, Jackie, Tom Davis, and Mason Woo, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1, Addison-Wesley Publishers Japan, Tokyo, 1993 (ISBN 4-7952-9645-6).

Nikkei Electronics, No. 616, Sept. 5, 1994, p. 99-105.

OpenGL Architecture Review Board, OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1, Addison-Wesley Publishers Japan, Tokyo, 1992 (ISBN 4-7952-9644-8).

“OpenGL,” Nikkei Computer Graphics, 1/1995, p. 203-209.

PIXEL, No. 143, 8/94, p. 65 (“From the Editor’s Desk”), p. 117-121 (“3D API, OpenGL”).

Sasaki, Akiko and Masayuki Matsumoto, Software Design, “Chapter 3: OpenGL”, November 11, 1994, ISSN 0916-6297, p. 26-48.

“Windows NT and OpenGL,” Nikkei Computer Graphics, 3/1995, p. 156-161.

Woo, Mason, “OpenGL,” Nikkei Computer Graphics, 11/1994, p. 142.

“X Windows and OpenGL,” Nikkei Computer Graphics, 2/1995, p. 155-160.

Technical reports

Segal, Mark and Kurt Akeley. The OpenGL Graphics System: A Specification. Technical report, Silicon Graphics Computer Systems, Mountain View, California, 1992, revised 1993.

Segal, Mark and Kurt Akeley. The OpenGL Graphics Interface. Technical paper, Silicon Graphics Computer Systems, Mountain View, California, 1993.

------ Q_1_06: Where can I get the OpenGL specification?

A: A PostScript version of OpenGL specification, along with the OpenGL Utility Library and GLX protocol specifications, are available via anonymous, public ftp, on the machine sgigate.sgi.com in ~ftp/pub/opengl/doc. They are all in the file, specs.tar.Z, which has been tar’d and compressed. The man pages for the OpenGL API, its Utility Library (GLU), and the X server extension API (GLX) are also here.

Please read the accompanying README file, which explains the copyright and trademark rules for usage of the specification. Possession of the OpenGL Specification does not grant the right to reproduce, create derivative works based on or distribute or manufacture, use or sell anything that embodies the specification without an OpenGL license from SGI.

An HTML version of the OpenGL specification can be found on http://www.sgi.com/Technology/openGL/glspec/glspec.html

An HTML version of the OpenGL man pages can be found on http://www.digital.com:80/pub/doc/opengl/

------ Q_1_07: Which vendors are licensing OpenGL?

A: OpenGL is supported by many hardware and software vendors. As of September, 1995, OpenGL has been licensed to:

3Dlabs, AT&T, AccelGraphics, Cirrus Logic, Cray Research, Daikin, Digital Equipment, Division, Dynamic Pictures, Evans & Sutherland, HP Harris Computer, Hitachi, IBM, Intel, Intergraph, Japan Radio Co., Kendall Square Research, Media Vision, Metro Link, Microsoft, Miro, NCD, NEC, NeTpower, Peritek, Portable Graphics, SPEA, Samsung, Sony, SunSoft, Template Graphics Software, The Institute for Information Industry, Univel

------ Q_1_08: What OpenGL implementations are available?

A: - AccelGraphics

AccelGraphics, Inc. is currently shipping the AG300, a high-performance PCI-based OpenGL(R) graphics board for the PC. AccelGraphics, Inc. is an OpenGL licensee and provides full support of OpenGL via client-loadable library on Windows NT(R) 3.5.

Running on a standard Pentium(TM), Alpha(TM), or MIPS(TM) PC with a PCI bus, the AG300 graphics accelerator card lets you manipulate larger and more complex 3D models and assemblies dynamically. True color and full 3D acceleration, with a 16-bit Z-buffer and smooth double-buffered display at full screen (1280x1024) resolution, combine to provide high-performance dynamic viewing and rendering.

Hardware Support Scalable architecture efficiently leverages the system CPU 7.5 MB of total RAM

- 5 MB of VRAM for 32 plane frame-buffer, - 2.5 MB of DRAM for 16-bit Z-buffer Rectangle clipping, Alpha blending, Logic Operations, Bilinear Interpolation, and Dithering

Supported Drivers and Applications Microsoft Windows NT 3.5 or higher with OpenGL Microsoft Windows 3.1 Windows 95(R) with OpenGL (late 1995) Pro/ENGINEER(TM) and Pro/JR.(TM) from Parametric Technology Corporation AutoCAD(TM) from Autodesk MicroStation(TM) from Bentley Systems Virtually any 3D application that utilizes OpenGL.

Supported Hardware Platforms Any Intel Pentium system with 1 free PCI slot Digital’s Alpha based PC’s MIPS based PC’s.

Supported Operating Systems DOS, Windows 3.1, Windows NT.

AccelGraphics, Inc. is headquartered in San Jose, CA with regional offices in Atlanta, Orlando, Los Angeles, Cincinnati and London, England.

For more information on the AG300, please call AccelGraphics, Inc. at 1-800-444-5699.

AutoCAD is a registered trademark of Autodesk. Microstation is a registered trademark of Bentley Systems. Pentium is a trademark of Intel Corporation. AG300, ActionGraphics and AccelGraphics are trademarks of AccelGraphics, Inc. Windows, Windows NT and Windows 95 are registered trademarks of Microsoft Corporation. OpenGL is a registered trademark of Silicon Graphics. Pro/ENGINEER and Pro/JR. are trademarks of Parametric Technology Corporation. All other trademarks are the property of the companies that issued them.

- Digital (DEC)

Digital Equipment Corporation offers OpenGL to its customers as part of the the DEC Open3D layered product. DEC Open3D is available for DEC OSF/1 AXP and DEC OpenVMS AXP workstations. Supported graphics devices include:

PXG (all devices in the PXG family with z-buffers) ZLX-M1 ZLX-M2 ZLX-E1 ZLX-E2

At this time, Digital Equipment Corporation has no plans to offer Open3D on either VAXstations or DECstations.

Digital Equipment Corporation (DEC) is shipping accelerated OpenGL for Windows NT on our AlphaStation models 200 and 400 using the ZLXp-E1, ZLXp-E2, and ZLXp-E3 graphics options.

Now you can unleash the industry-leading speed and power of Digital's Alpha AXP technology with low-cost high-performance graphics accelerators that will change the way you view your work.

The ZLXp-E1 provides leading 2D performance inexpensively. And breakthrough dithering technology lets the ZLXp-E1 display 3D smooth shaded images in 8 planes with outstanding quality. Coupled with the ZLXp-E1's excellent

performance, Digital's patented dithering capability provides an ideal solution for professionals in CASE, ECAD, and mechanical product design.

The ZLXp-E2 delivers the high-performance and 24 plane, true color capability needed for image processing, medical imaging, desktop publishing, graphics arts, and multimedia. The ZLXp-E2 can be configured to run 8 bits double buffered with a 16 bit Z buffer, providing full dedicated hardware support for 3D applications.

The ZLXp-E3, featuring true color capability and a full 24 bit Z buffer for even more complex solid model rendering, is ideal for mechanical CAD and computer-aided molecular design.

For further information contact your Digital Equipment sales representative.

- IBM

IBM offers OpenGL 1.0 at several different price and performance points, on most configurations of it's RS/6000 workstation line. Hardware accelerated OpenGL is available through two recently announced products: the mid-range POWERgraphics GXT1000 and the high-end Freedom Series /6000. Both platforms provide h/w support for texture mapping, accumulation, stencil and alpha buffers, as well as a h/w accelerated lighting and geometry transformation pipeline. The Freedom Series is based on an architecture developed by Evans and Sutherland.

OpenGL is supported on most other RS/6000 configurations, including the GXT100 and GXT150 graphics adapters for the PowerPC-based /6000's as well as on the CGDA, the Gt1 family, the Gt3 family, and the Gt4 family of graphics adapters. This support is provided through SoftGraphics, a highly-tuned, highly-optimized pure software implementation of OpenGL. (Because of the lack of support for an RGB X11 TrueColor visual on the 3D-HP-CGP and GTO adapters, OpenGL is not offered on these machines. This is the only exception to OpenGL support on the RS/6000 line. Sorry). OpenGL requires AIX 3.2.5 or later.

At the Fall '93 Comdex, IBM exhibited a software technology that allowed OpenGL to run under OS/2. The interface that integrates OpenGL with OS/2 was presented to the OpenGL ARB for review. Beta versions of that interface will be available in the first half of 1994, through the OS/2 Developer CD-ROM distribution.

To purchase these products, contact your local IBM sales office.

- Intergraph

Intergraph Computer Systems is currently shipping high-performance, PCI-based, OpenGL accelerators on its TD series of Personal Workstations. Using state of the art dedicated hardware, the GLZ and GLI graphics products dramatically accelerate OpenGL and offer the high performance and features traditionally found only on much more expensive workstations. These accelerators are available on Intergraph's TD-4 and TD-5 dual-Pentium Personal Workstations running Windows NT.

GLZ and GLI offer advanced features such as: - 24-bit, double buffered image planes at all display resolutions up to 2 Mpixels - 24-bit (GLZ) or 32-bit (GLI) Z-Buffer - Full hardware support for Gouraud shading - Full hardware support for texture processing (GLI only) with 8 MTexels of texture storage - Industry-standard PCI bus interface with DMA engine - Support for multi-sync monitors up to 2 Mpixels at 76Hz vertical refresh - Stereo ready - Multiple color palette support - 10-bit gamma correction

For additional information call 1 (800) 763-0242 or browse Intergraph's WWW pages at http://www.intergraph.com.

- Microsoft

OpenGL is offered as a standard feature of Microsoft Windows NT Workstation version 3.5. The Microsoft implementation of OpenGL runs with any computer and video hardware that is compatible with Windows NT 3.5. Microsoft also provides documentation, sample source code, and development tools to help build OpenGL applications in the Win32 Software Development Kit. The Win32 SDK is available via Microsoft's Developer Network.

For more information on the Win32 SDK, please call: US at 1-800-759-5474 International at +1-402-691-0173

- Portable Graphics, Inc.

3D Graphics Development and Porting Tools

Portable Graphics, a wholly-owned subsidiary of Evans & Sutherland Computer Corporation, provides GL-based development and porting tools for a variety of workstation and PC platforms.

OpenGL for HP

Portable Graphics is developing a hardware accelerated implementation of OpenGL for the HP 9000 family of graphics workstations, through an agreement with Hewlett-Packard. OpenGL for HP from Portable Graphics will fully utilize HP's VISUALIZE Graphics Technology for 3D systems, as well as HP's patented Color Recovery technology, to create true color visuals (24-plane) on low-cost (8-plane) graphics workstations. It will also include the OpenGL Utility Libraries, which contain routines using lower-level OpenGL commands to set up specific viewing orientations and projections matrices, perform polygon tessellation, and render surfaces. OpenGL for HP from Portable Graphics will be available in March 1996.

OpenGL for Linux

OpenGL for Linux, an Evans & Sutherland product, is a software implementation that passes the ARB's OpenGL compliance tests and runs as an extension to the standard X package on Linux. Users can configure additional extension and video drivers as needed using the LinkKit. OpenGL for Linux is priced at U.S. $79, plus shipping and handling. It is available directly from Portable Graphics. For a copy of the OpenGL for Linux FAQ, send email to: linuxogl@portable.com.

Open Inventor 2.1.1 3D Developer's Toolkit

Developed by Silicon Graphics and licensed to Portable Graphics, Open Inventor is an object-oriented toolkit used to streamline the development of interactive 3D graphics applications based on OpenGL. Open Inventor is used to develop a wide range of applications, including geophysical visualization, interactive 3D games, animation, CAD, modeling and industrial design, collaborative work, visual simulation, desktop publishing, scientific data visualization, commercial database visualization, computer-based training, presentations, and virtual reality. Portable Graphics has, or is porting, Open Inventor 2.1.1 to the following workstation and PC platforms: Digital UNIX, HP-UX, IBM AIX, Linux, OS/2 Warp, Solaris, Windows NT and Windows 95. The AIX implementation is shipped on IBM's OpenGL for AIX CD-ROM.

EDISON Extensions to Open Inventor

EDISON seamlessly integrates powerful software modules with Open Inventor, further extending its capabilities and simplifying the development of complex 3D applications.

The first software module available for EDISON integrates the SHAPES geometric computing system from XOX Corporation (Minneapolis, MN) with Open Inventor. This technology creates for the first time an industry-standard application development tool that can accurately build, display, and modify geometric models with full material properties. EDISON/SHAPES currently supports Silicon Graphics workstations. A Windows NT version is in development.

V-Realm 3D Web Browser/V-Realm Builder

V-Realm is a 3D browser designed to run on the World Wide Web (WWW). This commercial product is jointly offered by Portable Graphics and IDS to Internet users, content providers, and corporations developing WWW sites. Based on the emerging VRML standard and the established Open Inventor 3D developer's toolkit from Silicon Graphics, Inc., V-Realm integrates 3D viewing, browsing, and navigation into a WWW browser. It incorporates advanced image,.video, audio, animation (behaviors), virtual reality techniques and basic Internet functions, such as HTML. V-Realm runs stand-alone, or as a Netscape plug-in. V-Realm Builder, a VRML authoring tool, is under development for release in 1996.

For more information about Portable Graphics products, contact:

Portable Graphics, Inc. 3006 Longhorn Blvd., Suite 105 Austin, TX 78758

Tel: (512) 719-8000 Fax: (512) 832-0752 e-mail: info@portable.com http:// www.portable.com

- Silicon Graphics

Starting with IRIX 5.2, OpenGL is supported for the following graphics workstations:

Indy - Indy XL 8 or 24 bits, XZ (XZ, as of IRIX 5.3) Indigo - Entry Level, XS, XS24, XZ, Elan Indigo2 - XL, XZ, Extreme Crimson - Entry Level, XS, XS24,

Elan, Extreme, RealityEngine Onyx - VTX, RealityEngine, RealityEngine2 4D30/35 - Elan

With IRIX 5.3, OpenGL is supported for these workstations:

Personal IRIS Graphics: 8-bit, G, TG (except GR1.1) VGX, VGXT, Skywriter

This leaves the following graphics families with no OpenGL implementation:

IRIS 1000, 2000, and 3000 series IRIS 4D/G, GT, GTX Personal IRIS GR1.1 (suggest purchasing graphics board upgrade to GR1.2)

- Sony

Sony offers OpenGL on the complete range of its RISC based NEWS workstations. Sony OpenGL requires NEWS OS 6.0.1 and later. This is a pure software implementation.

Starting in September 1994, Sony have hardware support for OpenGL on its 3D graphics workstations: on the NWS-5000G and the 3D graphics acceleration card NWB-1501 for NWS-5000 series workstations.

- Template Graphics Software, Inc. (TGS)

The Standard in Graphics Tools

OpenGL - Accelerated to Hardware (Sun, Apple, Microsoft Windows 3.1)

TGS is providing OpenGL direct to Sun SPARC Solaris 2.x acceleration hardware. This differs from other software-only products in that it avoids the additional XGL software layer. The result is a fast performing and fully functional OpenGL for Sun workstations and clone systems. (OpenGL for Solaris from TGS was recently selected by Aries Research as the OpenGL to be sold with their SPARC systems.) * In final beta now, available via ftp * 100% functional today * Direct acceleration for Sun ZX, Turbo ZX * Direct support for GX, TGX, SX board sets * X11 network rendering to X terminals, PC-X servers, etc. * PostScript hardcopy (on final release) * GLX server extension for Solaris (on final) * Does not require XGL for rendering TGS will also be providing OpenGL for Apple Power Macintosh in early 1995, with a software-rendering and graphics acceleration version. TGS is working with

3D chip/board vendors to deliver accelerated OpenGL for the Power Mac platform.

TGS will also be providing OpenGL for Windows 3.1, direct to GDI, to ISV and and OEM customers. OpenGL for Windows 3.1 is fully portable with the OpenGL for Windows NT 3.5 provided by Microsoft, including the WGL component.

Open Inventor - C++ 3D Graphics Toolkit

TGS will be a single-stop solution for Open Inventor 2.0 on UNIX and PC systems, outside of SGI of course!

* Open Inventor for Solaris - shipping (beta) * Open Inventor for IBM AIX - shipping (beta) * Open Inventor for Windows NT 3.5 - in alpha * Open Inventor for Windows 3.1 - 2Q95 * Open Inventor for Windows 95 - TBA * Open Inventor for DEC OSF/1 - 2Q95 * Open Inventor for HP - 2Q95 * Open Inventor for Apple - 3Q95 * Open Inventor for OS/2 - TBA

Note: All of our Open Inventor products are tightly integrated into the OpenGL on each system, including support for 3rd party acceleration boards from Evans & Sutherland, GLINT, and others. TGS is unique in our support for a direct to hardware OpenGL for Solaris, Apple and is the only vendor to support OpenGL for Win32s (Windows 3.1). TGS Power Tools (tm) for Open Inventor

TGS is also developing TGS Power Tools(tm) for Open Inventor which will include:

* Power Filters (tm) - Import/Export of 3D metafiles * Power Viewers (tm) - 3D Desktop Utilties * Web3D (tm) - 3D Internet tools

Additional information on TGS Power Tools will be provided on request.

Sales and Support

TGS has supported ISV’s and professional graphics software developers since 1982 from our San Diego headquarters. TGS has regional sales offices in San Jose, Houston, Atlanta and Boston. We also have distribution partners in Europe and Asia.

For additional information on TGS graphics software:

Template Graphics Software 9920 Pacific Heights Blvd. #200 San Diego, CA 92121

WWW = http://www.sd.tgs.com/~template info@tgs.com

Robert J. Weideman, V.P. Marketing (619)457-5359 x229 (619)452-2547 (fax) robert@tgs.com

- 3Dlabs

3Dlabs is currently shipping the GLINT 300SX, a high performance graphics processor providing workstation class 3D graphics acceleration in a single chip. Designed to accelerate OpenGL, the GLINT 300SX implements in hardware 3D rendering operations such as Gouraud shading, depth buffering, anti-aliasing and alpha blending.

Implemented around a scalable memory architecture, the GLINT 300SX reduces the cost and complexity of delivering high performance 3D graphics - making it ideal for a wide range of graphics products from PC boards to high-end workstation accelerators. GLINT based products are already shipping from several companies with other developments in progress.

Key features of the GLINT 300SX are:

- Full hardware support for Gourard shading, depth buffering, alpha blending, anti-aliasing and dithering; - 8, 16 or 32-bits per pixel RGBA and 4 or 8-bit color indexed; - Screen resolutions up to 2560x2048; - 16, 24 or 32-bit Z buffer; - 4 or 8-bit stencil buffer - Double buffering, stereo and overlay support ; - PCI-bus Rev 2.0 interface with on-chip DMA; - 112-bit memory interface.

Since hardware is worthless without software, 3Dlabs have developed a highly optimized OpenGL driver for the GLINT 300SX. This OpenGL driver is currently available under Windows NT 3.5 and will be ported to other operating systems such as Windows 95. With GLINT 300SX and 3Dlabs' OpenGL GLINT driver, applications have achieved up to a 3,000 percent 3D display performance increase.

To find out more:

3Dlabs Inc 2010 North First Street, Suite 403 San Jose, CA 95131, Tel: (408) 436 3455 Fax: (408) 436 3458 Email: info@3Dlabs.com WWW URL http://www.3Dlabs.com/3Dlabs

------ Q_1_09: Does Windows 95 support OpenGL?

A: The Windows 95 OpenGL DLLs are available to Microsoft Developers' Network Level II subscribers. Applications developers may redistribute them freely.

------ Q_1_10: What interest is there for OpenGL in Japan?

A: OpenGL_Japan user group:

http://www.sgi.com/Technology/openGL/opengl.japan.html

------ Q_2_01: Who needs to license OpenGL? Who doesn't?

A: Companies which will be creating or selling binaries of the OpenGL library will need to license OpenGL. Typical examples of licensees include hardware vendors, such as Digital Equipment, and IBM who would distribute OpenGL with the system software on their workstations or PCs. Also, some software vendors, such as Portable Graphics and Template Graphics, have a business in creating and distributing versions of OpenGL, and they need to license OpenGL.

Applications developers do NOT need to license OpenGL. If a developer wants to use OpenGL, that developer needs to obtain copies of a linkable OpenGL library for a particular machine. Those OpenGL libraries may be bundled in with the development and/or run-time options or may be purchased from a third-party software vendor, without licensing the source code or use of the OpenGL(R) trademark.

Since many implementations will be a shared library on a hardware platform, the royalty sometimes will be charged for each hardware platform. In those cases, it would not be charged for each application which used OpenGL.

In general, licensing a source code implementation of OpenGL would not be useful for an application developer, because the binary created from that implementation would not be accelerated and optimized to run on the graphics hardware of a machine.

------ Q_2_02: What are the conformance tests?

A: The conformance tests are a suite of programs which judge the success of an OpenGL implementation. Each implementor is required to run these tests and pass them in order to call their implementation with the registered trademark OpenGL. Passing the conformance tests ensures source code compatibility of applications across all OpenGL implementations.

------ Q_2_03: What is Silicon Graphics policy on "free" implementations of APIs which resemble the OpenGL API?

A: Silicon Graphics, as licensor of the OpenGL(R) trademark, does not permit non-licensed use of the OpenGL trademark, nor does it permit non-licensed use of the OpenGL conformance tests. Silicon Graphics provides a source code sample

implementation of OpenGL, but only to companies and organizations which agree to the terms and conditions of an OpenGL license.

Silicon Graphics does give permission to others to create and distribute their own implementations of the OpenGL API, provided they do not state nor imply they have the right to use the OpenGL(R) trademark to name their product, nor make claims to conformance based upon the ARB controlled OpenGL conformance tests. Silicon Graphics agrees to allow others to copy the OpenGL header files, as much as is necessary, for the creation of other implementations.

Silicon Graphics is in no way associated nor endorsing these other graphics libraries. Silicon Graphics does not make any claims or guarantees as to the quality, performance, nor completeness of an unlicensed library.

------ Q_2_04: What is Mesa 3D and where can I get it?

A: From brianp@ssec.wisc.edu (Brian Paul)

The “Mesa 3-D graphics library” (or just Mesa) is a free implementation of the OpenGL API. About 90% of OpenGL’s functionality is implemented. Mesa works on almost any Unix system with ANSI C and X. MS Windows and Macintosh drivers are also available.

See the WWW page at http://www.ssec.wisc.edu/~brianp/Mesa.html for more information. You can get Mesa by anonymous ftp from iris.ssec.wisc.edu in the pub/Mesa directory or from the sunsite mirrors in pub/packages/development/graphics/mesa.

------ Q_2_05: How does a university or research institution acquire access to OpenGL source code?

A: There is a university/research institution licensing program. A university license entitles the institution to generate binaries and copy them anywhere, so long as nothing leaves the institution. The OpenGL source and derived binaries can only be used for non-commercial purposes on-campus.

A university license costs $500 US. This license provides source code for a sample implementation of OpenGL. This source code is best designed for porting onto a system which supports the X Window System. You can drop this into the X

Consortium's X11 server source tree and build a server with the OpenGL extension. To do this properly, you should have the MIT source for an X Server and some experience modifying it.

Note that this gets you a software renderer only. If your machine includes a graphics accelerator, the sample implementation is not designed to take any advantage of it.

To obtain a university license, contact John Schimpf, OpenGL Licensing Manager at Silicon Graphics (jsch@sgi.com). Please provide a mailing address, telephone and fax number.

Universities may also be interested in Mesa 3D. See Q_2_04.

------ Q_2_06: How is a commercial license acquired?

A: If you need a license or would like more information, call John Schimpf at (415)933-3062 or e-mail him at jsch@sgi.com. There are licenses available restricted to site (local) usage, or permitting redistribution of binary code. The limited source license provides a sample implementation of OpenGL for $50,000. The license for commerical redistribution of OpenGL binaries has two most commonly chosen levels. Level 1 costs $25,000. Level 2 costs $100,000, and includes the sample implementation of OpenGL. Both levels require a $5 royalty for every copy of the OpenGL binary, which is redistributed.

------ Q_2_07: How is the OpenGL governed? Who decides what changes can be made?

A: OpenGL is controlled by an independent board, the Architecture Review Board (ARB). Each member of the ARB has one vote. The permanent members of the ARB are Digital Equipment, IBM, Intel, Microsoft, and Silicon Graphics. Additional members will be added over time. The ARB governs the future of OpenGL, proposing and approving changes to the specification, new releases, and conformance testing.

------ Q_2_08: Who are the current ARB members?

A: In alphabetical order:

Digital Equipment Evans & Sutherland IBM Intel Intergraph Microsoft Silicon Graphics

------ Q_2_09: What is the philosophy behind the structure of the ARB?

A: The ARB is intended to be able to respond quickly and flexibly to evolutionary changes in computer graphics technology. The ARB is currently “lean and mean” to encourage speedy communication and decision-making. Its members are highly motivated in ensuring the success of OpenGL.

------ Q_2_10: How does the OpenGL ARB operate logistically? When does the ARB have meetings?

A: ARB meetings are held about once a quarter. The meetings rotate among sites hosted by the ARB members. To learn the date and place of the next OpenGL ARB meeting, watch the news group comp.graphics.opengl for posting announcing the next “OpenGL ARB meeting”, check the Web site http://www.sgi.com/Technology/openGL/arb.location.html or e-mail opengl-secretary@sgi.com and ask for the information.

Meetings are run by a set of official by-laws. A copy of the by-laws may be requested from the Secretary of the OpenGL ARB.

Minutes to the ARB meeting are posted to comp.graphics.opengl and are available via ftp://sgigate.sgi.com/pub/opengl/arb/

------ Q_2_11: How do additional members join the OpenGL ARB?

A: The intention is that additional members may be added on a permanent basis or for a one-year term. The one-year term members would be voting members, added on a rotating basis, so that different viewpoints (such as ISV’s) could be incorporated into new releases. Under the by-laws, SGI formally nominates new members.

------ Q_2_12: So if I’m not a member of the ARB, am I shut out of the decision making process?

A: There are many methods by which you can influence the evolution of OpenGL.

1) Contribute to the comp.graphics.opengl news group. Most members of the ARB read the news group religiously. 2) Contact any member of the ARB and convince that member that your proposal is worth their advocacy. Any ARB member may present a proposal, and all ARB members have equal say. 3) Come to OpenGL ARB and speak directly to ARB.

------ Q_2_13: Are ARB meetings open to observers?

A: The ARB meeting will be open to observers, but we want to keep the meeting small. Currently, up to five non-voting representatives who inform the ARB secretary in advance, can observe and participate in the ARB meeting. At any time, the ARB reserves the right to change the number of observers.

------ Q_2_14: What benchmarks exist for OpenGL?

A: OpenGL Performance Characterization for the GPC: http://sunsite.unc.edu/gpc/opc.html

Viewperf 3.0 source: ftp://net1.uspro.fairfax.va.us/pub/gpc/opc/viewperf

------ Q_3_01: Where can I find OpenGL source code examples? For instance, where is an example which combines OpenGL with Motif, using the Motif widget?

A: You can get the source code examples which are found in the OpenGL Programming Guide via anonymous, public ftp from ftp://sgigate.sgi.com/pub/opengl/opengl.tar.Z

Mark Kilgard has created an ftp site for source code, which is part of his articles in the X Journal magazine. This includes the GLUT toolkit (version 2.0) and OpenGL with Motif examples. The directory is ftp://sgigate.sgi.com/pub/opengl/xjournal

Some contributed source code of useful tools for developing OpenGL code can be found on ftp://sgigate.sgi.com/pub/opengl/contrib Source code found here includes: isfast--compares the performance of OpenGL states samples--more OpenGL program examples toogl--helps port IRIS GL code to OpenGL xglinfo--display information on X visuals extended for OpenGL xscope--examines OpenGL protocol sent to an extended X server

Nate Robins <narobins@ES.COM> has a web page that contains OpenGL information. In particular, source for the Windows NT/95 version of GLUT. The site is http://www.cs.utah.edu/~narobins/opengl.html.

------ Q_3_02: How do I contribute OpenGL code examples to a publicly accessible archive?

A: To contribute to the public OpenGL archive, send mail to opengl-contrib@sgi.com. Your mail should contain:

The material to be archived or instructions for obtaining it. An announcement suitable for posting to comp.graphics.opengl.

SGI will place the material in the opengl/contrib directory on sgigate.sgi.com and post the announcement to this newsgroup.

To retrieve something from the archive, use anonymous ftp to sgigate.sgi.com. Once connected, cd to the directory OpenGL. (Case is significant.) Currently there are two subdirectories:

doc - Manual pages for OpenGL and related libraries. contrib - Contributions from the public.

Note that all contributions are distributed as-is; neither SGI nor the other companies on the OpenGL Architecture Review Board make any legally valid claims about the robustness or usefulness of this software.

If you do not have access to anonymous ftp, consider using an “ftp-by-mail” server. For information on one such server, send mail to ftpmail@decwrl.dec.com with a message body containing only the word “help”.

------ Q_3_03: What is the GLUT toolkit? Where do I get it?

A: GLUT is a portable toolkit which performs window and event operations to support OpenGL rendering.

GLUT version 2.0 has: o window functions, including multiple windows for OpenGL rendering o callback driven event processing o sophisticated input devices, including dials and buttons box, tablet, Spaceball(TM) o idle routines and timers o a simple cascading pop-up menu facility o routines to generate wire and solid objects o bitmap and stroke fonts o request and queries for multisample and stereo windows o OpenGL extension query support

The version 2 functionality is fully backward compatible with the version 1 functionality.

The specification, source code (including FORTRAN bindings), and articles for GLUT (Graphics Library Utility Toolkit) is in: URL ftp://sgigate.sgi.com/pub/opengl/xjournal/GLUT

This distribution of GLUT should compile on: o DEC Alpha workstation running OSF/1 with Open3D layered product o IBM RS/6000 workstations running AIX with OpenGL support o SGI workstation running IRIX 5.2 or higher supporting OpenGL o Template Graphics Software’s OpenGL for Sun workstations o Mesa 1.1 for Unix workstations.

------ Q_3_04: What is the relationship between IRIS GL and OpenGL? Is OpenGL source code or binary code compatible with IRIS GL?

A: IRIS GL is the predecessor to OpenGL. After other implementors had experience trying to port the IRIS GL to their own machines, it was learned that the IRIS GL was too tied to a specific window system or hardware. Based upon consultations with several implementors, OpenGL is much more platform independent.

IRIS GL is being maintained and bugs will be fixed, but SGI will no longer add enhancements. OpenGL is now the strategic interface for 3-D computer graphics.

OpenGL code is neither binary nor source code compatible with IRIS GL code. It was decided to bite the bullet at this time to make OpenGL incompatible with IRIS GL and fix EVERYTHING that made IRIS GL difficult to port or use. For example, the gl prefix has been added to every command: glVertex(), glColor(), etc.

------ Q_3_05: Why should I port my IRIS GL application to OpenGL?

A: SGI will be maintaining the old IRIS GL, but not enhancing it. OpenGL is the API of choice on all new SGI machines.

OpenGL has no subsets. You can use the same functionality on all machines from SGI or from other vendors.

OpenGL is better integrated with the X Window System than the old IRIS GL. For example, you can mix OpenGL and X or Display PostScript drawing operations in the same window.

The OpenGL naming scheme, argument list conventions, and rendering semantics are cleaner than those of IRIS GL. This should make OpenGL code easier to understand and maintain.

------ Q_3_06: How much work is it to convert an IRIS GL program to OpenGL? What are the major differences between them?

A: from Mason Woo (woo@sgi.com) and Debbie Herrington (debbie@portable.com)

There is a fair amount of work, most of which is in substituting for window management or input handling routines, for which the equivalents are not OpenGL, but the local window system, such as the X Windows System or Windows NT.

To help ease the way, port to “mixed model” right away, mixing the X Window System calls to open and manage windows, cursors, and color maps and read events of the window system, mouse and keyboard. You can do that now with IRIS GL, if you are running IRIX 4.0.

In the X Window System, display mode choices (such as single or double buffering, color index or RGBA mode) must be declared before the window is initially opened. You may also substitute for other IRIS GL routines, such as using a OSF/Motif menu system, in place of the IRIS GL pop-up menus. You should use glXUseXFont(), whenever you were using the font manager with IRIS GL.

OpenGL uses standard, predictable naming conventions, which required that all names have been changed from IRIS GL. The OpenGL naming scheme uses unique prefixes, suffixes, and capitalization to help prevent potential conflicts among application, system, and library routine names.

And all routine names have changed, at least, minimally; for example: ortho() is now glOrtho().

Tables for states such as lighting or line and polygon stipples will be gone. Instead of using a def/set or def/bind sequence to load a table, you turn on the state with glEnable() and also declare the current values for that state.

Colors are best stored as floating point values, scaled from 0.0 to 1.0 (0% to 100%). Alpha is fully integrated in the RGBA mode and at least source alpha will be available on all OpenGL implementations. OpenGL will not arbitrarily limit the number of bits per color to 8. Clearing the contents of buffers no longer uses the current color, but a special “clearing” color for each buffer (color, depth, stencil, and accumulation).

The transformation matrix has changed. In OpenGL, there is no single matrix mode. Matrices are now column-major and are post-multiplied, although that does NOT change the calling order of these routines from IRIS GL to OpenGL. OpenGL’s glRotate*() now allows for a rotation around an arbitrary axis, not just the x, y, and z axes. lookat() of IRIS GL is now gluLookAt(), which takes an up vector value, not merely a twist. There is no polarview() in OpenGL, but a series of glRotate*()s and glTranslate*()s can do the same thing.

There are no separate depth cueing routines in OpenGL. Use linear fog.

Feedback and selection (picking) return values, which are different from those found on any IRIS GL implementation. For selection and picking, depth values will be returned for each hit. In OpenGL, feedback and selection will now be standardized on all hardware platforms.

------ Q_3_07: When using Xlib, how do I create a borderless window?

A: from blythe@sgi.com (David Blythe)

Essentially you can create the window with override-redirect (see man xcreatewindow) which is the sledgehammer approach or you can change the _MOTIF_WM_HINTS property to tell the window manager to leave your windows undecorated.

from alex@eagle.hd.hac.com (Alex Madarasz)

Also of note is that the window manager decorations of any client can be turned off by putting something like the following in the .Xdefaults file in your home directory - assuming you aren't overriding them in your app:

4Dwm*ClientAppOrClassName*clientDecoration: none

(see the 4Dwm / mwm man pages for a full description of this resource)

(you must restart the window manager or logout/login for 4Dwm resources to take effect)

"none" will remove all of the window manager decorations - border, title bar etc.

------ Q_3_08: How do I switch between single buffer and double buffer mode?

A: from mjk@sgi.com (Mark Kilgard) and blythe@sgi.com (David Blythe)

When using OpenGL with X, switching between a double buffered and single buffered window can be accomplished by creating a "container" X window and creating two subwindows, one with a double buffer visual, the other with a single buffer visual. Make sure the subwindows are each the full size of their parent window.

You can then use XRaiseWindow or XLowerWindow to change the stacking order of the two subwindows to switch between double buffering and single buffering.

You will need to create a separate context for each of the two windows since they have different visual types. You will need to make the appropriate window/context pair current when you switch modes.

IRIS GL made it easy to switch between double buffering and single buffering. But essentially, IRIS GL implemented the above process internally.

------ Q_3_09: On my machine, it appears that glXChooseVisual is only able to match double-buffered visuals. I want to have more bits of color resolution, so how do I render in single buffer mode?

A: from mjk@sgi.com (Mark Kilgard)

On mid- to low-end machines with double buffer hardware, you'll probably find you get twice as much color resolution using a single buffer visual. But if there is no hardware double buffering support, the double buffered and single buffered visuals are generally the same depth (the back buffer is "carved" out of software).

Search again for a double buffered visual. If you find one, use it instead. Call glDrawBuffer(GL_FRONT) though to make sure you are drawing to the front buffer (the default for a double buffered visual is to draw into the back buffer).

------ Q_3_10: I've got a 24-bit machine, but my OpenGL windows are not using the full color resolution. What's wrong? My program looks fine on one machine, but the depth buffer doesn't work on another. What's wrong?

A: from mjk@sgi.com (Mark Kilgard) and woo@sgi.com (Mason Woo)

An unfortunate (but documented) semantic of glXChooseVisual is that if you don't request GLX_RED_SIZE, GLX_GREEN_SIZE, or GLX_BLUE_SIZE, glXChooseVisual assumes zero for these parameters which means pick the visual with the _smallest_ amount of red, green, and blue that matches the other visual attributes. Make sure you ask for at least 1 bit of GLX_RED_SIZE, GLX_GREEN_SIZE, and GLX_BLUE_SIZE. If these configuration parameters are non-zero, it matches the visual with the _largest_ amount of red, green, and blue with at least 1 bit of each (probably what you want).

Similarly, if you don't request GLX_DEPTH_SIZE, you may get a visual with zero bits of depth buffer. Some systems may have few visuals available, and those visuals all have at least 1 bit of depth buffer. On other systems, there may be dozens of visuals available, some with zero bits for the depth buffer. In short, if hidden surface removal appears to fail, check to see if you have explicitly specified any bits of depth buffer you have requested. Also check to see what visual you have received.

------ Q_3_11: What information is available about OpenGL extensions?

A: Examples of extensions include vertex arrays (calling several vertexes or related data, such as normals or colors, with a single function call), blending of constant colors, polygon offset (multiple coplanar polygons can be rendered without interaction),

Procedure names and tokens for OpenGL extensions are either suffixed with EXT or a vendor-specific acronym: such as SGI for Silicon Graphics machines, or INGR for Intergraph. Also note that Silicon Graphics extensions to OpenGL are suffixed to indicate whether they will be available on all machines (SGI), on just a subset of machines (SGIS), or are very experimental and may become unavailable or completely changed (SGIX).

Vendors are encouraged to add extension information to their documentation. For Silicon Graphics, extension information is summarized on the glIntro man page.

------ Q_3_12: How do I make shadows in OpenGL?

A: There are no individual routines to control shadows nor an OpenGL state for shadows. However, code can be written to render shadows.

from woo@sgi.com (Mason Woo)

To project a shadow onto a flat plane (such as in the insect demo), draw the stippled object, flattened using matrix transformations. The easiest way to flatten an object is to use the scale function. For example, use glScalef(1., 0., 1.) to create from an infinite light shining straight down the y axis. A transformation matrix can be used to cast a shadow from an infinite or local light source in an arbitrary direction. See the article:

Thant Tessman, “Casting Shadows on Flat Surfaces,” IRIS Universe, Winter, 1989.

from shreiner@sgi.com (Dave Shreiner)

Check out the fast shadow and projective texture multi-pass algorithms for producing realistic shadows using texture mapping. See the SIGGRAPH paper:

Mark Segal, Carl Korobkin, et al. “Fast Shadows and Lighting Effects using Texture Mapping” 1992 SIGGRAPH Proceedings

------ Q_3_13: How can I use 16 bit X fonts?

A: from James A. (“Jim”) Miller

Here is some code that will load any font into the server and use glXUseXFont to build the display lists for you (this does work with 16 bit fonts, it has been tested on IBM, DEC and SGI machines at an OpenGL interop testing). Once your display lists are created for each character, you can use the same basic logic to figure out which characters are valid (using first, last, firstrow and lastrow in the sample code to get : firstchar = 256 * firstrow + first lastchar = 256 * lastrow + last) and use glCallList with those characters to print them out.

static int LoadFont(char *fontName) { Font id; int first, last, firstbitmap, i; GLuint base; Display *display=0; int firstrow, lastrow; int maxchars;

tkGetSystem(TK_X_DISPLAY, &display);

fontInfo = XLoadQueryFont(display, fontName); if (fontInfo == NULL) { return 0; } id = fontInfo->fid; /* * First and Last char in a row of chars */ first = (int)fontInfo->min_char_or_byte2; last = (int)fontInfo->max_char_or_byte2; /* * First and Last row of chars, important for multibyte charset’s */ firstrow = (int)fontInfo->min_byte1; lastrow = (int)fontInfo->max_byte1; /* * How many chars in the charset */ maxchars = 256 * lastrow + last; base = glGenLists(maxchars+1); if (base == 0) { return 0; } /* * Get offset to first char in the charset */ firstbitmap = 256 * firstrow + first; /* * for each row of chars, call glXUseXFont to build the bitmaps. */ for(i=firstrow; i<=lastrow; i++) { glXUseXFont(id, firstbitmap, last-first+1, base+firstbitmap); firstbitmap += 256; } return base; }

------ Q_3_14: What's in the new GLU 1.2 tesselator?

A: from Mark Kilgard

Our friends at Digital have the answers: http://www.digital.com:80/pub/doc/opengl/opengl_new_glu.html

------ Q_3_15: Why is my glDrawPixels (or glCopyPixels or glReadPixels) slow?

A: from Allen Akin

It's not, for the most common cases. After all, similar microcode and the same hardware are used for both commands. However, there are three issues to keep in mind.

First, some midrange and low-end SGI graphics adaptors (particularly XS, XZ, Elan, and Extreme) transfer ABGR-ordered images much faster than they transfer RGBA-ordered images. The normal image format in IrisGL was ABGR, while in OpenGL it's RGBA. So to achieve the same performance in OpenGL that you did in IrisGL on those machines, you need to use ABGR-format images in OpenGL. The ABGR extension available in Irix 5.3 and later releases allows you to do this. See ``man glintro'' for background information on using OpenGL extensions, and ``man gldrawpixels'' for details on ABGR. Note that RealityEngine, IMPACT, and all future machines will process RGBA data at least as fast as ABGR, so RGBA is the way to go for new code.

Second, some OpenGL pixel data types are faster than others. For most machines, unsigned byte RGBA (or ABGR) is the fastest full-color type. Unsigned byte and unsigned short are usually the fastest gray-scale types. Signed integer types are slower.

Third, OpenGL pixel operations have a much richer set of features than IrisGL, and if any of those features are enabled, then image transfer can be significantly slower. Always disable the features that you don't need. The following code fragment disables features that are likely to make glDrawPixels slow:

/* * Disable stuff that's likely to slow down glDrawPixels. * (Omit as much of this as possible, when you know in advance * that the OpenGL state will already be set correctly.) * Note that all of these are the default settings, except * for

GL_DITHER! */ glDisable(GL_ALPHA_TEST); glDisable(GL_BLEND); glDisable(GL_DEPTH_TEST); glDisable(GL_DITHER); glDisable(GL_FOG); glDisable(GL_LIGHTING); glDisable(GL_LOGIC_OP); glDisable(GL_STENCIL_TEST); glDisable(GL_TEXTURE_1D); glDisable(GL_TEXTURE_2D); glPixelTransferi(GL_MAP_COLOR, GL_FALSE); glPixelTransferi(GL_RED_SCALE, 1); glPixelTransferi(GL_RED_BIAS, 0); glPixelTransferi(GL_GREEN_SCALE, 1); glPixelTransferi(GL_GREEN_BIAS, 0); glPixelTransferi(GL_BLUE_SCALE, 1); glPixelTransferi(GL_BLUE_BIAS, 0); glPixelTransferi(GL_ALPHA_SCALE, 1); glPixelTransferi(GL_ALPHA_BIAS, 0);

/* * Disable extensions that could slow down glDrawPixels. * (Actually, you should check for the presence of the proper * extension before making these calls. I've omitted that * code for simplicity.) */

#ifdef GL_EXT_convolution glDisable(GL_CONVOLUTION_1D_EXT); glDisable(GL_CONVOLUTION_2D_EXT); glDisable(GL_SEPARABLE_2D_EXT); #endif

#ifdef GL_EXT_histogram glDisable(GL_HISTOGRAM_EXT); glDisable(GL_MINMAX_EXT); #endif

#ifdef GL_EXT_texture3D glDisable(GL_TEXTURE_3D_EXT); #endif

------ Q_3_16: How can I used OpenGL with Tcl/Tk?

A: TIGER 1.2: ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/TIGER1.2/
Tix: http://www.cis.upenn.edu/~ioi/tix/sgi.html

------ Q_3_17: OpenGL pixel-exact rasterization.

A: from Kurt Akeley

1) Why is the center for points, lines and bitmaps different than for polygon vertices and pixel image positions?

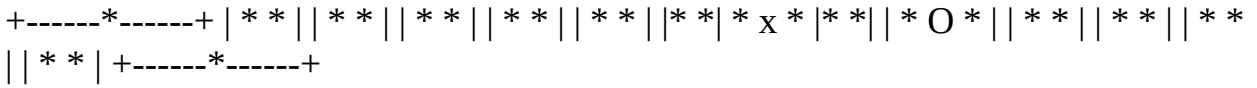
Let's just consider lines and polygons. Polygons have real area, lines have area only because they would otherwise not be visible. A polygon is sampled by determining if its area intersects a pixel. OpenGL makes this determination by

testing a single point within each pixel (this way only a single polygon in a 2D mesh can contain the sample point). It is symmetric and obvious for this point to be in the middle of the pixel, so that's what we chose.

There were lots of considerations for our choice of line rasterization algorithms. We ended up with what we call the "diamond-exit rule". Each pixel center is surrounded by a square rotated 45 degrees, whose vertexes just touch the 1x1 pixel boundaries. If a line segment *exits* this diamond, the pixel is rasterized; otherwise it is not. This algorithm generates Bresenham lines, does not multiply rasterize pixels at the shared vertexes of line segments, and guarantees to generate pixels, no matter how convoluted and intricate the path of a sequence of segments is. (It has other properties as well, I don't remember all the considerations.)

2) How does translationg 0.375 ensure predicatable rasterization for all primitives?

To generate lines reliably the vertexes must be within the diamonds, and to fill polygons reliably, the vertexes must *not* be too near the pixel centers. I chose 0.375 as the best compromise, as seen in the diagram below ('x' is the center of the pixel, 'O' is the point 0.375, 0.375, and the diamond is shown with asterisks).



------ Q_3_18: Saving OpenGL screen output.

A: from Reto Koradi

/* OpenGL image dump, written by Reto Koradi (kor@spectrospin.ch) */

/* This file contains code for doing OpenGL off-screen rendering and saving the result in a TIFF file. It requires Sam Leffler's libtiff library which is available from ftp.sgi.com. The code is used by calling the function StartDump(..), drawing the scene, and then calling EndDump(..). Please note that StartDump creates a new context, so all attributes stored in the current context (colors, lighting parameters, etc.) have to be set again beforing performing the actual redraw. This can be rather painful, but unfortunately GLX does not allow sharing/copying of attributes between direct and nondirect rendering contexts. */

```
#include <stdio.h> #include <stdlib.h> #include <X11/Xlib.h> #include <X11/
Intrinsic.h> #include <GL/gl.h> #include <GL/glx.h>

#include <tiffio.h>

/* X servers often grow bigger and bigger when allocating/freeing many pixmaps,
so it's better to keep and reuse them if possible. Set this to 0 if you don't want to use
that. */ #define KEEP_PIXMAP 1

static FILE *TiffFileP; static int Orient; static int ImgW, ImgH; static Bool
OutOfMemory; static Display *Dpy; static Pixmap XPix = 0; static GLXPixmap
GPix = 0; static GLXContext OldCtx, Ctx; static float OldVpX, OldVpY, OldVpW,
OldVpH;

static void destroyPixmap(void) { glXDestroyGLXPixmap(Dpy, GPix); GPix = 0;
XFreePixmap(Dpy, XPix); XPix = 0; }

static int xErrorHandler(Display *dpy, XErrorEvent *evtP) { OutOfMemory =
True; return 0; }

int StartDump(char *fileName, int orient, int w, int h) /* Prepare for image dump.
fileName is the name of the file the image will be written to. If orient is 0, the image
is written in the normal orientation, if it is 1, it will be rotated by 90 degrees. w and
h give the width and height (in pixels) of the desired image. Returns 0 on success,
calls RaiseError(..) and returns 1 on error. */ { Widget drawW = GetDrawW(); /*
the GLwMDrawA widget used */ XErrorHandler oldHandler; int attrList[10];
XVisualInfo *visP; int n, i;

TiffFileP = fopen(fileName, "w"); if (TiffFileP == NULL) { RaiseError("could not
open output file"); return 1; }

#if KEEP_PIXMAP if (GPix != 0 && (w != ImgW || h != ImgH))
destroyPixmap(); #endif

Orient = orient; ImgW = w; ImgH = h;

Dpy = XtDisplay(drawW);
```

```
 n = 0; attrList[n++] = GLX_RGBA; attrList[n++] = GLX_RED_SIZE;
attrList[n++] = 8; attrList[n++] = GLX_GREEN_SIZE; attrList[n++] = 8;
attrList[n++] = GLX_BLUE_SIZE; attrList[n++] = 8; attrList[n++] =
GLX_DEPTH_SIZE; attrList[n++] = 1; attrList[n++] = None; visP =
glXChooseVisual(Dpy,  XScreenNumberOfScreen(XtScreen(drawW)), attrList); if
(visP == NULL) { RaiseError("no 24-bit true color visual available"); return 1; }

 /* catch BadAlloc error */ OutOfMemory = False; oldHandler =
XSetErrorHandler(xErrorHandler);

 if (XPix == 0) { XPix = XCreatePixmap(Dpy, XtWindow(drawW), w, h, 24);
XSync(Dpy, False); /* error comes too late otherwise */ if (OutOfMemory) {  XPix
= 0;  XSetErrorHandler(oldHandler);  RaiseError("could not allocate Pixmap");
return 1; } }

 if (GPix == 0) { GPix = glXCreateGLXPixmap(Dpy, visP, XPix); XSync(Dpy,
False); XSetErrorHandler(oldHandler); if (OutOfMemory) {  GPix = 0;
XFreePixmap(Dpy, XPix);  XPix = 0;  RaiseError("could not allocate Pixmap");
return 1; } }

 Ctx = glXCreateContext(Dpy, visP, NULL, False); if (Ctx == NULL) {
destroyPixmap(); RaiseError("could not create rendering context"); return 1; }

 OldCtx = glXGetCurrentContext(); (void) glXMakeCurrent(Dpy, GPix, Ctx);

 return 0; }

 static int writeTiff(void) { TIFF *tif; int tiffW, tiffH; int rowsPerStrip, bufSize,
rowI; unsigned char *buf; int res;

 tif = TIFFFdOpen(fileno(TiffFileP), "output file", "w"); if (tif == NULL) {
RaiseError("could not create TIFF file"); return 1; }

 if (Orient == 0) { tiffW = ImgW; tiffH = ImgH; bufSize = 4 * ((3 * tiffW + 3) / 4);
glPixelStorei(GL_PACK_ALIGNMENT, 4); } else { tiffW = ImgH; tiffH = ImgW;
bufSize = 3 * tiffW; glPixelStorei(GL_PACK_ALIGNMENT, 1); }

 rowsPerStrip = (8 * 1024) / (3 * tiffW); if (rowsPerStrip == 0) rowsPerStrip = 1;
```

```
 TIFFSetField(tif, TIFFTAG_IMAGEWIDTH, tiffW); TIFFSetField(tif,
TIFFTAG_IMAGELENGTH, tiffH); TIFFSetField(tif,
TIFFTAG_BITSPERSAMPLE, 8); TIFFSetField(tif, TIFFTAG_COMPRESSION,
COMPRESSION_LZW); TIFFSetField(tif, TIFFTAG_PHOTOMETRIC,
PHOTOMETRIC_RGB); TIFFSetField(tif, TIFFTAG_FILLORDER,
FILLORDER_MSB2LSB); TIFFSetField(tif, TIFFTAG_DOCUMENTNAME,
“My Name”); TIFFSetField(tif, TIFFTAG_IMAGEDESCRIPTION, “My
Description”); TIFFSetField(tif, TIFFTAG_SAMPLESPERPIXEL, 3);
TIFFSetField(tif, TIFFTAG_ROWSPERSTRIP, rowsPerStrip); TIFFSetField(tif,
TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);

 buf = malloc(bufSize * sizeof(*buf));

 res = 0; for (rowI = 0; rowI < tiffH; rowI++) { if (Orient == 0)  glReadPixels(0,
ImgH - 1 - rowI, ImgW, 1,  GL_RGB, GL_UNSIGNED_BYTE, buf); else
glReadPixels(rowI, 0, 1, ImgH,  GL_RGB, GL_UNSIGNED_BYTE, buf);

if (TIFFWriteScanline(tif, buf, rowI, 0) < 0) {  RaiseError(“error while writing
TIFF file”);  res = 1;  break; } }

 free(buf);

 TIFFFlushData(tif); TIFFClose(tif);

 return res; }

 int EndDump(void) /* Write current image to file. May only be called after
StartDump(..). Returns 0 on success, calls RaiseError(..) and returns 1 on error. */ {
int res;

 res = writeTiff(); (void) fclose(TiffFileP);

 (void) glXMakeCurrent(Dpy, XtWindow(GetDrawW()), OldCtx);

 #if KEEP_PIXMAP #else destroyPixmap(); #endif

 glXDestroyContext(Dpy, Ctx);

 return res; }
```

------ Q_3_19: No Logicop in RGB for OpenGL?

A: - from Kurt Akeley

The best solution that I know of is to save and restore the portion of the 3D image that will be damaged by the 2D rendering. If the machine supports double buffering, you can use the back buffer for this purpose. Use glCopyPixels to do the transfers. Otherwise use glReadPixels to save the region and glDrawPixels to restore it.

Note that an advantage to the save/restore approach is that you have control of the colors of your 2D rendering. Using logical operations provides much less control.

- From Brian Paul (brianp@ssec.wisc.edu)

There is an extension EXT_blend_logic_op, available in many implementations which allows you to do what you need. Use #ifdef GL_EXT_blend_logic_op / #endif in your source code to test for the extension at compile time and glGetString(GL_EXTENSIONS) to test for it at run-time. Then the call to glBlendEquationEXT(GL_LOGIC_OP) will tell GL that blending is to be done with the logic op specified by glLogicOp().

If neither overlay planes nor this extension are available, you may have to resort to your window system's XOR drawing facility. If you're using X, you can set your GC's function to GXxor and use the Xlib drawing functions. If you do this be sure to synchronize the GL and X drawing with glXWaitX() and glXWaitGL().

------ Q_3_20: Why does the raster position get clipped by the viewing volume?

A: from Kurt Akeley

A vertex on the Z=0 plane in clip coordinates is projected to infinity if not clipped. Since OpenGL is a 3D library, such a vertex is always a possibility, even in a 2D application. Thus the raster position is always clipped.

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 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.

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 goemetric 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

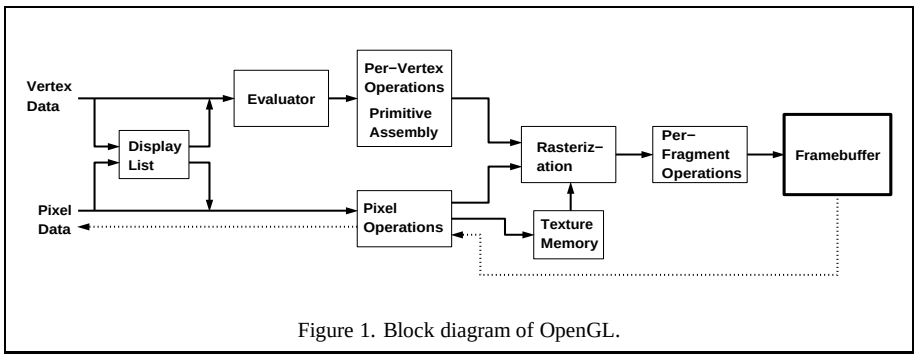


Figure 1. Block diagram of OpenGL.

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 suficient 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 illustratations 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 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:

```
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 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 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 procedure calls was deemed more important than any inefficiency induced 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

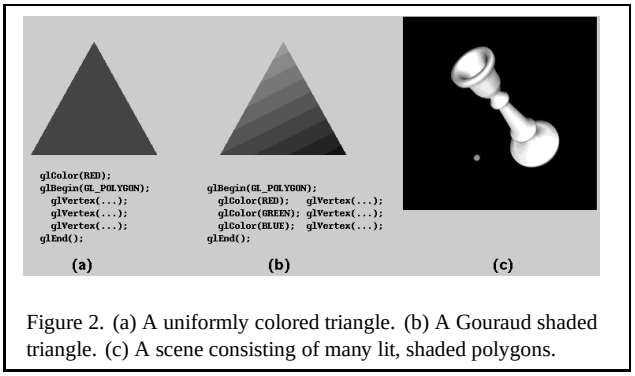


Figure 2. (a) A uniformly colored triangle. (b) A Gouraud shaded triangle. (c) A scene consisting of many lit, shaded polygons.

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 primitive'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 considerable state, or modes, that determines how primitives are rendered. 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 proportion 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 distinct primitives. In such cases, a mode change must be broadcast to all processors so that each receives the new parameters before it processes 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[1].

Another way to handle state changes relies on defining groups of named state settings which can then be invoked simply by providing the appropriate name (this is the approach taken by X and PEX). With this approach, a single command naming the state setting changes the server's settings. This approach was rejected for OpenGL for several reasons. Keeping track of a number of state vectors (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 control; there are cases, as when changing the state of a single mode, when transmitting the change directly is more convenient and efficient 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 *texture* matrix, which is applied to texture coordinates; and the *projection* 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 4×4.

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 manipulations. 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 simplifies 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 effects 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 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 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 interactive 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 surface removal using depth buffering are two examples). Although OpenGL may be used to provide information for driving such devices as pen-plotters and vector displays, such use is secondary.

Multipass Algorithms

One useful effect of making the framebuffer explicit is that it enables 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 anti-aliasing, 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 z) 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.

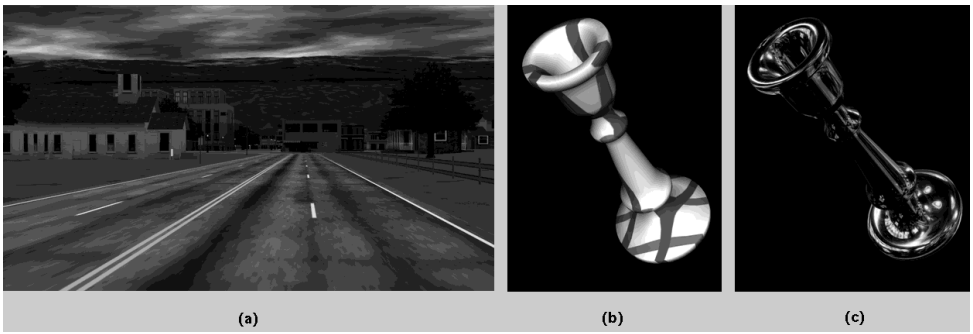


Figure 3. (a) A scene with a number of textures mapped onto primitives. (b) Contouring achieved with texture mapping and a texture coordinate generation function. (c) Reflectance mapping with a texture coordinate generation function.

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.

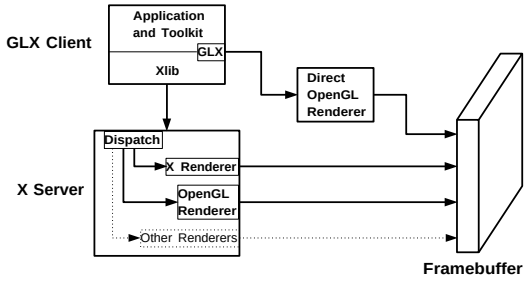


Figure 4. GLX client, X server, and OpenGL renderers.

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 regard 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 implementation 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:

(a) (b) (c)

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 applications. 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 provides 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, interoperability, 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 implementations through optimization, and extending the API to handle 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 Conference 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 fundamental 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) functional 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 Programming 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 interferences. *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 Proceedings)*, 26(2):249–252, July 1992.

[15] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 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.