A Collection of OpenGL Programs *

William Shoaff with lots of help

November 19, 2002

Contents

1. Hello World

Here's about the most basic OpenGL program one can write: It is from the Red book Woo et al. [1999]. It opens a window on the screen and draws a centered white square. As is popular custom, it is called `hello, world`, in homage to Kernighan and Ritchie, the creators of the `C` programming language Kernighan and Ritchie [1988]. I've rewritten this version of `hello, world` using LaTeX, the `pdfscreen` style package, and noweb a literate programming tool to make it more readable as a stand-alone program.

The program has 4 main parts: header files for inclusion, 2 functions to initialize and display the graphics, and a main function to control the program.

10a ⟨*hello* 10a⟩≡
⟨*Hello header files* 11a⟩
⟨*Hello* `display` *function* 17a⟩
⟨*Hello initialization function* 15a⟩
⟨*Hello* `main` *function* 13a⟩

1.1. Header Inclusion

There is one header file that must be included: `glut.h`. It is in `/usr/local/include/GL` on my system. GLUT is an OpenGL Utility Toolkit created by Mark Kilgard Kilgard [1996]. The `glut.h` header file includes `gl.h` and `glu.h`, so there is no need to explicitly include these header files.

The GLUT library provides functions for handling window systems and input devices. For example, GLUT provides pop-up menu support and device handling support for devices such as a keyboard and mouse. GLUT invokes user-supplied callbacks to handle window events such as exposure and resizing.

GLUT also offers utility routines for drawing several geometric shapes as solids or wireframe models, including spheres, tori, and teapots.

GLUT is available on most UNIX platforms, MacOS, and Windows, and other operating systems. It can be downloaded from

http://www.opengl.org/Developers/Documentation/glut.html.

Online documentation is available at

http://www.opengl.org.

There are other OpenGL toolkits. See the Developer's Tools page at

http://www.opengl.org.

11a ⟨*Hello header files* 11a⟩≡

```
#include <GL/glut.h>
#define OK 0
```

1.2. The `main` function

`main` performs the tasks necessary to open a window and run the program. There are 5 basic steps to initialize the window with GLUT.

1. `glutInit(int *argc, char **argv)` initializes GLUT and should be called before any other GLUT routines. Any command line arguments that initialize the display device are processed.

2. `glutInitDisplayMode(unsigned int mode)` specifies which color models and buffers to use. It takes a single argument that is the bitwise OR of values, some of which are:

   - `GLUT_RGB` or `GLUT_RGBA` informs the system to use the red, green, blue, alpha color model.
   - `GLUT_INDEX` set the system to use a color lookup table to index set colors.
   - `GLUT_SINGLE` specifies use of a single color buffer.
   - `GLUT_DOUBLE` enables double buffering.
   - `GLUT_ACCUM` enables an accumulation buffer for special effects.
   - `GLUT_DEPTH` enables a depth buffer for hidden surface removal.
   - `GLUT_STENCIL` enables a stencil buffer to restrict the drawing area.

3. `glutInitWindowSize(int width, int height)` sets the width and height of the window in pixels.

4. `glutInitWindowPosition(int x, int y)` sets the (x, y) offset of the window from the upper-left corner of the screen.

5. `glutCreateWindow(char *string)` creates, but does not display, the window.

After initializing the window, `main` calls `init()` to initialize various OpenGL attributes and calls `glutDisplayFunc(void (*func)(void))` that registers a function to be called whenever GLUT determines that the contents of the window need to be redisplayed.

The last thing `main` does is call `glutMainLoop(void)` that causes all created windows be to shown and event processing to start. Once this event processing loop is called, it never exits.

13a ⟨*Hello* `main` *function* 13a⟩≡

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(250, 250);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("hello, world");
   init();
   glutDisplayFunc(display);
   glutMainLoop();
   return OK;
}
```

1.3. An Initialization function

Before drawing into a color buffer some OpenGL attributes should be set, in particular, for efficiency, operations that need to be called only once can be executed.

- `glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)` specifies the colors to be used by `glClear()` to clear (initialize) the color buffers. Values are clamped to the range [0, 1]. Using the value 0.0 for each color component sets the background to black.

- `glMatrixMode(GLenum mode)` sets the current matrix mode. There are several stack of matrices used by OpenGL: projection, color, texture, model-view.

- `glLoadIdentity(void)` replaces the current matrix with a 4×4 identity matrix

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

- `glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar)` multiplies the current matrix by an orthographic matrix to produce a parallel projection. This orthographic matrix is

$$M = \begin{bmatrix} \frac{2}{\text{right-left}} & 0 & 0 & -\frac{\text{right+left}}{\text{right-left}} \\ 0 & \frac{2}{\text{top-bottom}} & 0 & -\frac{\text{top+bottom}}{\text{top-bottom}} \\ 0 & 0 & \frac{2}{\text{zFar-zNear}} & -\frac{\text{zFar+zNear}}{\text{zFar-zNear}} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

(left, bottom, -near) and (right, top, -near) map to the lower left and upper right corners on the near clipping plane. For the values specified in the call below this sets the lower left corner at (0.0, 0.0) and the upper right corner at (1.0, 1.0).

15a ⟨*Hello initialization function* 15a⟩≡

```
void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
```

1.4. A Display function

Here's a simple function to `display` a white square inside a window. The function calls several OpenGL functions. The Blue book Shreiner [1999] is a good reference where you can find how these OpenGL functions are called and what they do. Briefly, here's what the functions used in `display` are doing.

- `glClear(GLbitfield mask)` clears a memory buffer to a previously set value. It takes a single argument that is the bitwise OR of several values that indicate which buffers are to be cleared. These values are defined in `gl.h` and given symbolic names.
  - `GL_COLOR_BUFFER_BIT` indicates the (current) framebuffer enabled for writing pixel color data.
  - `GL_DEPTH_BUFFER_BIT` indicates the depth buffer used for hidden/visible surface determination.
  - `GL_ACCUM_BUFFER_BIT` indicates the accumulation buffer used for antialiasing, motion blur, soft shadows from multiple light sources, and other special effects.
  - `GLSTENCIL_BUFFER_BIT` indicates the stencil buffer used to restrict drawing to certain portions of the screen.
- `glColor3f(GLfoat red, GLfloat green, GLfloat blue)` specifies new red, green, and blue values for the current color. Using the value 1.0 for each color component sets the current color to white.
- `glBegin(GLenum mode)` starting delimiter for a list of vertices of a graphics primitive or group of primitives. There many primitives. In this `display` function a single convex polygon is defined by specifying the `GL_POLYGON` enumeration constant.
- `glVertex3f(GLfoat x, GLfloat y, GLfloat z)` specifies a point, line end-point, or polygon vertex. Below the four calls to `glVertex3f()` define the corners of a square that will be filled with white.

- `glEnd()` ending delimiter for a primitive definition.
- `glFlush()` forces the execution of all GL commands that may have been stored awaiting execution.

Figure 1 shows a picture of how the window and square will look.

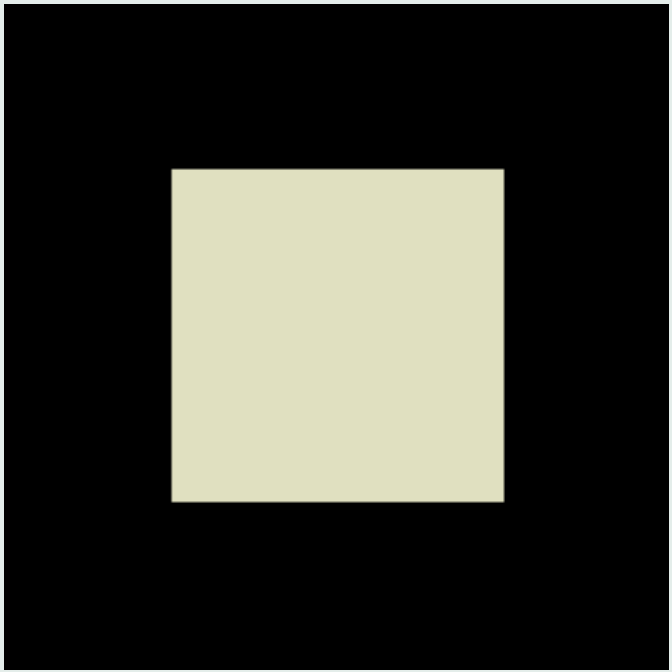

Figure 1: White square in hello world window

17a ⟨*Hello* `display` *function* 17a⟩≡

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_POLYGON) ;
      glVertex3f(0.25, 0.25, 0.0);
      glVertex3f(0.75, 0.25, 0.0);
      glVertex3f(0.75, 0.75, 0.0);
      glVertex3f(0.25, 0.75, 0.0);
    glEnd() ;
    glFlush();

}
```

1.5. Hello Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/hello.c

2. Double Buffering and Input Control

This program, from the Red book Woo et al. [1999], illustrates double buffering, resizing, and mouse button input events. As in Hello, World a white square is drawn in a window. Pressing the left mouse button rotates the square counter-clockwise. Pressing the right mouse button stops the rotation. And pressing the middle button doesn't do anything at all. The program also allows for the window to be resized or moved.

19a ⟨*double* 19a⟩≡
- ⟨*Double header files* 20a⟩
- ⟨*Double* `display` *and redisplay* 22a⟩
- ⟨*Double change spin angle callback* 25a⟩
- ⟨*Double initialization function* 22b⟩
- ⟨*Double* `reshape` *the window callback* 23a⟩
- ⟨*Double* `mouse` *event callback* 24a⟩
- ⟨*Double* `main` *function* 21a⟩

2.1. Headers and routine variables

We'll again use GLUT routines to manage windows and I/O events. A variable, named `spin`, with scope limited to this file determines the angle of rotation. `spin` must be known to both the `display` and idle loop callback function `spinDisplay`.

20a ⟨*Double header files* 20a⟩≡

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

#define OK 0

GLfloat spin = 0.0;
```

2.2. A Main function

The differences between this `main` function and the `Hello, World main` functions are described. Consult the `Hello, World` program for additional instruction on what happens here.

Two new GLUT functions are introduced:

- `glutReshapeFunc(void (*func)(int width, int height))` registers a callback function that is executed when the window is moved or resized to a new width and height.
- `glutMouseFunc(void (*func)(int button, int state, int x, int y))` registers a callback function that is executed when a mouse event is detected. The left, middle, or right button is either up or down at window coordinates (x, y).

21a ⟨*Double* `main` *function* 21a⟩≡

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE || GLUT_RGB);
   glutInitWindowSize(250, 250);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Double Buffering");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return OK;
}
```

2.3. Displaying and Redisplaying

When double buffering is enabled, the system displays one color buffer while writing to another and then swaps the roles of both buffers. This `display` function clears the color buffer that will be written, pushes the current transformation matrix onto the projection stack,

22a ⟨*Double* `display` *and redisplay* 22a⟩≡

```
void display(void) {
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glRotatef(spin, 0.0, 0.0, 1.0);
   glColor3f(1.0, 1.0, 1.0);
   glRectf(-25.0, -25.0, 25.0, 25.0);
   glPopMatrix();
   glutSwapBuffers();
}
```

2.4. An Initialization function

Most of the functionality can be taken out of the `init` function. A new GLUT function

- `glShadeModel(GLenum mode)` is introduced. When polygons are filled the interior can be one solid color (the *flat* model) or the color intensity can be changed smoothly across the polygon using the *Gouraud* shading algorithm.

22b ⟨*Double initialization function* 22b⟩≡

```
void init(void) {
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}
```

2.5. Reshaping the window

Most of the initialization steps are moved to a `reshape` callback function that is registered with GLUT to be activated anytime a window move or resize event is detected.

23a ⟨*Double* `reshape` *the window callback* 23a⟩≡

```
void reshape(int width, int height) {
   glViewport(0, 0, (GLsizei) width, (GLsizei) height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

2.6. A Mouse Event Callback function

Whenever a mouse event interrupt is detected, the function `mouse` registered with GLUT as the *mouse function* is called. The action taken depends on which mouse button was clicked and the state of the button (up or down).

The mouse event is used to animate or stop animation of the scene. The `glutIdleFunc(void (*func)(void))` is useful for continuous animation. It specifies which function executes when no other events are pending.

In case the mouse event comes from the *middle* button, nothing is done — no change in the graphics occurs. In case the mouse event comes from the *left* button and it's state is *down*, the `spinDisplay` function is registered to run continually, changing the rotation of the square and posting a redisplay event, until another event takes place. For example, when the *right* mouse button is held down, this event signals a change to set the idle function to no function at all, effectively stopping the spinning animation of the square.

24a ⟨*Double* `mouse` *event callback* 24a⟩≡

```
void mouse(int button, int state, int x, int y) {
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (GLUT_DOWN == state) {
            glutIdleFunc(spinDisplay);
            break;
         }
      case GLUT_MIDDLE_BUTTON: { break; }
      case GLUT_RIGHT_BUTTON: {
         if (state == GLUT_DOWN) {
          glutIdleFunc(NULL);
          break;
         }
      }
      default: { break; }
   }
```

```
}
```

2.7. Change the Spin Angle and Redisplay the Scene

The idle loop function `spinDisplay` simply increments `spin` by 2 (degrees) modulo 360, and post a redisplay event marking the window as needing to be redrawn. This causes `display` to be called, eventually.

25a ⟨*Double change spin angle callback* 25a⟩≡

```
void spinDisplay(void) {
   spin = spin + 2.0;
   if (spin > 360.0)
      spin = spin - 360.0;
   glutPostRedisplay();
}
```

2.8. Double Buffering Source Code

The C program code is available at:

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/double.c

3. States in OpenGl

What can be drawn and what is drawn is controlled by the *state* of a graphics system. Learning about the states of a graphic system goes a long way in understanding it. In this program we will query the state.

There are several functions used to get state information, among them are:

- `glGetgBooleanv, glGetDoublev, glGetFloatv, glGetIntegerv`.
- `glIsEnabled`.

and many that get specialized states associated with things like clipping planes, color tables, convolution filters, histograms, lights and materials, pixel maps, textures, etc. In the code below, we'll use `glGetLightfv`, `glGetMaterialfv`, `glString`, and `glGetConvolutionParameteriv`.

Here's the output from the program generated on my Sun Ultra 5 computer.

```
Vendor: Sun Microsystems, Inc.
Version: 1.2 Sun OpenGL 1.2.3 for Solaris
Rendered: SUNWm64 mmap software renderer, VIS
Extensions:  GL_EXT_texture3D GL_SGI_color_table
   GL_SGI_texture_color_table GL_EXT_abgr GL_EXT_rescale_normal
   GL_SUNX_surface_hint GL_EXT_multi_draw_arrays GL_SUN_multi_draw_arrays
   GL_SUNX_constant_data GL_EXT_polygon_offset GL_SUN_vertex
   GL_SUN_global_alpha GL_ARB_transpose_matrix GL_SUN_triangle_list
   GL_SGIS_sharpen_texture GL_SGIS_detail_texture
   GL_EXT_texture_filter_anisotropic GL_EXT_compiled_vertex_array
   GL_ARB_texture_border_clamp GL_SGIS_texture_border_clamp
   GL_ARB_texture_env_combine GL_EXT_texture_env_combine
   GL_SGIS_texture_filter4 GL_SGIX_texture_lod_bias
   GL_SGIX_texture_scale_bias GL_ARB_multitexture GL_SUN_mesh_array
Stereo views not supported
```

```
Double buffering supported
0 Auxiliary  buffers supported
Colors can be stored in RGBA mode
3 bits/pixel for Red
3 bits/pixel for Green
2 bits/pixel for Blue
0 bits/pixel for Alpha
Colors cannot be stored in index mode
8 bits for color indexing
32 bits/pixel for depth buffer
8 bits/pixel for stencil buffer
16 bits/pixel for Red in accumulation buffer
16 bits/pixel for Green in accumulation buffer
16 bits/pixel for Blue in accumulation buffer
16 bits for Alpha in accumulation buffer
3 bits of subpixel precision in x and y
Maximum texture size is 4096
Maximum 3D texture size is 1024
Up to 32 lights supported
Up to 6 users defined clipping planes supported
Maximum model-view stack depth is 32
Maximum projection stack depth is 10
Maximum texture stack depth is 10
Maximum selection-name stack depth is 128
Maximum attribute stack depth is 16
Maximum client attribute stack depth is 32
Maximum display list nesting is 64
Maximum viewport size x = 2000, y = 2000
Range of aliased point sizes 0.500000 to 100.000000
```

```
Range of antialiased (smooth) point sizes 0.500000 to 100.000000
Antialiased point size granularity 0.125000 to 0.000000
Width range of aliased lines 0.100000 to 100.000000
Width range of antialiased (smooth) lines 0.100000 to 100.000000
Antialiased line width granularity 0.125000 to 0.000000
Maximum degree for polynomials is 30
Maximum width of 1D convolutions filter 7
Maximum height of 1D convolution filter 7
Maximum width of 2D convolutions filter 7
Maximum height of 2D convolution filter 7
Maximum width of 2D separable convolution filter 7
Maximum height of 2D separable convolution filter 7
The current (integer) color: (2147483647, 2147483647, 2147483647)
The current (floating point) color: (1.000000, 1.000000, 1.000000)
The current vertex normal vector: (0.000000, 0.000000, 1.000000)
The model view matrix is
        [1.000000       0.000000        0.000000        0.000000        ]
        [0.000000       1.000000        0.000000        0.000000        ]
        [0.000000       0.000000        1.000000        0.000000        ]
        [0.000000       0.000000        0.000000        1.000000        ]
Viewport origin (0, 0)  Viewport width = 300, height= 300
MODELVIEW matrix mode
Lighting is not enabled
Color tracking is not enabled
Viewer is at infinity
Front face ambient material color: (0.200000, 0.200000, 0.200000, 1.000000)
Front face diffuse material color: (0.800000, 0.800000, 0.800000, 1.000000)
Front face specular material color: (0.000000, 0.000000, 0.000000, 1.000000)
Front face emissive material color: (0.000000, 0.000000, 0.000000, 1.000000)
```

```
Front face material specular exponent: 0.000000
Light zero ambient color: (0.000000, 0.000000, 0.000000, 1.000000)
Light zero diffuse color: (1.000000, 1.000000, 1.000000, 1.000000)
Light zero specular color: (1.000000, 1.000000, 1.000000, 1.000000)
Light zero position: (0.000000, 0.000000, 1.000000, 0.000000)
Constant attentuation coefficient: 1.000000
Linear attentuation coefficient: 0.000000
Quadratic attentuation coefficient: 0.000000
Point antialiasing is off
Line antialiasing is off
Polygon antialiasing is off
Polygon culling is disabled
Point size in pixels: 1.000000
Line width in pixels: 1.000000
Counterclockwise front faces
Fill front and back polygons
Cull back faces
```

29a ⟨*state* 29a⟩≡
⟨*State Header files* 29b⟩
⟨*State* `main` *function* 30a⟩

3.1. State header files

All we'll need is to include `glut.h`.

29b ⟨*State Header files* 29b⟩≡

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

3.2. State `main` function

This program does not display any graphics — it just querys the state of the system. First the system must be initialized so values are set, and we use the minimal calls necessary to do this.

There are many implementation dependent state variables and we query most, but not all of them. There are close to another 200 state state variable. I've picked out a few that seem interesting to me and advise you to see the Red Book Woo et al. [1999] and the Blue book Shreiner [1999] for all details.

30a ⟨*State* `main` *function* 30a⟩≡

```
int main(int argc, char** argv) {
  ⟨Local variables 31a⟩

  ⟨Initialize the graphics system 30b⟩
  ⟨Implementation-dependent states 33a⟩
  ⟨Current values 41a⟩
  ⟨Transformation state variables 42a⟩
  ⟨Lighting state variables 43a⟩
  ⟨Rasterization state variables 45a⟩
}
```

3.2.1. Initialize the graphics system

Not until the graphics system is initialized will the state be set. Throught some experimentation, it appears that asking GLUT to create the window is the minimum call necessary to initialize the state.

30b ⟨*Initialize the graphics system* 30b⟩≡

```
glutCreateWindow("State Testing");
```

3.2.2. Variables local to `main`

Here are some local variables that will be used.

31a ⟨*Local variables* 31a⟩≡

```
int i, j; // a couple of looping variables
GLboolean bufferSupport = GL_FALSE;
GLint pixelStore = -1;
GLint lightSupport = -1;
GLint clipSupport = -1;
GLint stackSupport = -1;
GLint displayListSupport = -1;
GLint convolution = -1;
GLint polynomialDegree = -1;
GLint viewport[4] = {-1, -1, -1, -1};
GLfloat pointSizes[2] = {-1.0, -1.0};
GLfloat lineSizes[2] = {-1.0, -1.0};
GLint iColor[3] = {-1, -1, -1};
GLfloat fColor[3] = {-1, -1, -1};
GLfloat normal[3] = {-1, -1, -1};
GLfloat matrix[16];
GLint matrixMode;
GLboolean localViewer = GL_FALSE;
GLfloat ambientColor[4] = {-1.0, -1.0, -1.0, -1.0};
GLfloat diffuseColor[4] = {-1.0, -1.0, -1.0, -1.0};
GLfloat specularColor[4] = {-1.0, -1.0, -1.0, -1.0};
GLfloat emissionColor[4] = {-1.0, -1.0, -1.0, -1.0};
GLfloat lightPosition[4] = {-1.0, -1.0, -1.0, -1.0};
GLfloat shininess = -1.0;
GLfloat constantAttenuation;
GLfloat linearAttenuation;
GLfloat quadraticAttenuation;
GLint faceType;
```

```
GLint fillMode;
GLint cullMode;
```

3.2.3. Implementation dependent state variables

Several capablities the graphic system depend on the particular implementatation and the hardware supported. Let's query these implementation dependent state variables. In particular,

- What buffers are contained in the framebuffer?
- How many bits per pixel do these buffers hold?
- How many lights are available?
- Hom many user defined clipping planes can be set?
- What is the depth of matrix stacks?
- How deeply can display lists be nested?

And some other information.

33a ⟨*Implementation-dependent states* 33a⟩≡
⟨*Version information* 34a⟩
⟨*Buffer support* 35a⟩
⟨*Light support* 37a⟩
⟨*Clip support* 37b⟩
⟨*Stack support* 38a⟩
⟨*Display list support* 38b⟩
⟨*Viewport size* 39a⟩
⟨*Point information* 39b⟩
⟨*Line information* 39c⟩
⟨*Polynomial evaluation* 40a⟩
⟨*Convolution filters* 40b⟩

3.2.3.1. What version is being used?

34a ⟨*Version information* 34a⟩≡

```
printf("Vendor: %s\n", glGetString(GL_VENDOR));
printf("Version: %s\n", glGetString(GL_VERSION));
printf("Rendered: %s\n", glGetString(GL_RENDERER));
printf("Extensions %s\n", glGetString(GL_EXTENSIONS));
```

3.2.3.2. Buffer support Here we'll learn if the system supports stereoptic views double buffering and auxiliary buffers. We'll also learn how many bits of resolution are supported in

35a ⟨*Buffer support* 35a⟩≡

```
glGetBooleanv(GL_STEREO, &bufferSupport);
if(bufferSupport) { printf("Stereo views supported\n"); }
else { printf("Stereo views not supported\n"); }

glGetBooleanv(GL_DOUBLEBUFFER, &bufferSupport);
if(bufferSupport) { printf("Double buffering supported\n"); }
else { printf("Double buffering not supported\n"); }

glGetIntegerv(GL_AUX_BUFFERS, &pixelStore);
printf("%d Auxiliary  buffers supported\n", pixelStore);

glGetBooleanv(GL_RGBA_MODE, &bufferSupport);
if(bufferSupport) { printf("Colors can be stored in RGBA mode\n"); }
else { printf("Colors cannot be stored in RGBA mode\n"); }

glGetIntegerv(GL_RED_BITS, &pixelStore);
printf("%d bits/pixel for Red\n", pixelStore);

glGetIntegerv(GL_GREEN_BITS, &pixelStore);
printf("%d bits/pixel for Green\n", pixelStore);

glGetIntegerv(GL_BLUE_BITS, &pixelStore);
printf("%d bits/pixel for Blue\n", pixelStore);

glGetIntegerv(GL_ALPHA_BITS, &pixelStore);
printf("%d bits/pixel for Alpha\n", pixelStore);

glGetBooleanv(GL_INDEX_MODE, &bufferSupport);
```

```
if(bufferSupport) { printf("Colors can be stored in index mode\n"); }
else { printf("Colors cannot be stored in index mode\n"); }

glGetIntegerv(GL_INDEX_BITS, &pixelStore);
printf("%d bits for color indexing\n", pixelStore);

glGetIntegerv(GL_DEPTH_BITS, &pixelStore);
printf("%d bits/pixel for depth buffer\n", pixelStore);

glGetIntegerv(GL_STENCIL_BITS, &pixelStore);
printf("%d bits/pixel for stencil buffer\n", pixelStore);

glGetIntegerv(GL_ACCUM_RED_BITS, &pixelStore);
printf("%d bits/pixel for Red in accumulation buffer\n", pixelStore);

glGetIntegerv(GL_ACCUM_GREEN_BITS, &pixelStore);
printf("%d bits/pixel for Green in accumulation buffer\n", pixelStore);

glGetIntegerv(GL_ACCUM_BLUE_BITS, &pixelStore);
printf("%d bits/pixel for Blue in accumulation buffer\n", pixelStore);

glGetIntegerv(GL_ACCUM_ALPHA_BITS, &pixelStore);
printf("%d bits for Alpha in accumulation buffer\n", pixelStore);

glGetIntegerv(GL_SUBPIXEL_BITS, &pixelStore);
printf("%d bits of subpixel precision in x and y\n", pixelStore);

glGetIntegerv(GL_MAX_TEXTURE_SIZE, &pixelStore);
printf("Maximum texture size is %d\n", pixelStore);

glGetIntegerv(GL_MAX_3D_TEXTURE_SIZE, &pixelStore);
printf("Maximum 3D texture size is %d\n", pixelStore);
```

The maximum number of lights may be more than the minimum number of 8.

37a ⟨*Light support* 37a⟩≡

```
glGetIntegerv(GL_MAX_LIGHTS, &lightSupport);
printf("Up to %d lights supported\n", lightSupport);
```

Additional clipping planes can be defined by application programmers. The OpenGL standard requires a minimum of 6 such planes to be supported.

37b ⟨*Clip support* 37b⟩≡

```
glGetIntegerv(GL_MAX_CLIP_PLANES, &clipSupport);
printf("Up to %d users defined clipping planes supported\n", clipSupport);
```

The model-view, projection and texture matrix stack should be at least 32, 2, and 2 deep. There are some other stacks: a *name* stack used in determining what object might have been selected in interactive selection mode; an *attribute group* stack used to store collections of attributes, for example, information about a line – its width, stipple, and aliasing mode; a *client attribute group* stack used grouping state on a client machine.

38a ⟨*Stack support* 38a⟩≡

```
glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, &stackSupport);
printf("Maximum model-view stack depth is %d\n", stackSupport);

glGetIntegerv(GL_MAX_PROJECTION_STACK_DEPTH, &stackSupport);
printf("Maximum projection stack depth is %d\n", stackSupport);

glGetIntegerv(GL_MAX_TEXTURE_STACK_DEPTH, &stackSupport);
printf("Maximum texture stack depth is %d\n", stackSupport);

glGetIntegerv(GL_MAX_NAME_STACK_DEPTH, &stackSupport);
printf("Maximum selection-name stack depth is %d\n", stackSupport);

glGetIntegerv(GL_MAX_ATTRIB_STACK_DEPTH, &stackSupport);
printf("Maximum attribute stack depth is %d\n", stackSupport);

glGetIntegerv(GL_MAX_CLIENT_ATTRIB_STACK_DEPTH, &stackSupport);
printf("Maximum client attribute stack depth is %d\n", stackSupport);
```

Display lists can contain display lists leading to complex hierarchical strucutures. But there may be a limit to this nesting. OpenGL requires this limit to be at least 64.

38b ⟨*Display list support* 38b⟩≡

```
glGetIntegerv(GL_MAX_LIST_NESTING, &displayListSupport);
printf("Maximum display list nesting is %d\n", displayListSupport);
```

What is the maximum view port size?

39a ⟨*Viewport size* 39a⟩≡

```
glGetIntegerv(GL_MAX_VIEWPORT_DIMS, viewport);
printf("Maximum viewport size x = %d, y = %d\n", viewport[0], viewport[1]);
```

What information can be learned about support for point primitives? In particular what is the range of sizes for aliased and antialiased lines and what is the granularity of point sizes?

39b ⟨*Point information* 39b⟩≡

```
glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, pointSizes);
printf("Range of aliased point sizes %f to %f\n", pointSizes[0], pointSizes[1]);

glGetFloatv(GL_SMOOTH_POINT_SIZE_RANGE, pointSizes);
printf("Range of antialiased (smooth) point sizes %f to %f\n", pointSizes[0], pointSizes[

glGetFloatv(GL_SMOOTH_POINT_SIZE_GRANULARITY, pointSizes);
printf("Antialiased point size granularity %f to %f\n", pointSizes[0]);
```

What information can be learned about support for line primitives?

39c ⟨*Line information* 39c⟩≡

```
glGetFloatv(GL_ALIASED_LINE_WIDTH_RANGE, lineSizes);
printf("Width range of aliased lines %f to %f\n", lineSizes[0], lineSizes[1]);

glGetFloatv(GL_SMOOTH_LINE_WIDTH_RANGE, lineSizes);
printf("Width range of antialiased (smooth) lines %f to %f\n", lineSizes[0], lineSizes[1]

glGetFloatv(GL_SMOOTH_LINE_WIDTH_GRANULARITY, lineSizes);
printf("Antialiased line width granularity %f to %f\n", lineSizes[0]);
```

Polynomials can be used to approximate complex curves. The maximum degree of these polynomials is bounded, but must be at least 8.

40a ⟨*Polynomial evaluation* 40a⟩≡

```
glGetIntegerv(GL_MAX_EVAL_ORDER, &polynomialDegree);
printf("Maximum degree for polynomials is %d\n", polynomialDegree);
```

Convolution filters are used in antialiasing. They have some width and height. We'll ask for the maximum width and height supported.

40b ⟨*Convolution filters* 40b⟩≡

```
glGetConvolutionParameteriv(GL_CONVOLUTION_1D, GL_MAX_CONVOLUTION_WIDTH, &convolution);
printf("Maximum width of 1D convolutions filter %d\n", convolution);

glGetConvolutionParameteriv(GL_CONVOLUTION_1D, GL_MAX_CONVOLUTION_HEIGHT, &convolution);
printf("Maximum height of 1D convolution filter %d\n", convolution);

glGetConvolutionParameteriv(GL_CONVOLUTION_2D, GL_MAX_CONVOLUTION_WIDTH, &convolution);
printf("Maximum width of 2D convolutions filter %d\n", convolution);

glGetConvolutionParameteriv(GL_CONVOLUTION_2D, GL_MAX_CONVOLUTION_HEIGHT, &convolution);
printf("Maximum height of 2D convolution filter %d\n", convolution);

glGetConvolutionParameteriv(GL_SEPARABLE_2D, GL_MAX_CONVOLUTION_WIDTH, &convolution);
printf("Maximum width of 2D separable convolution filter %d\n", convolution);

glGetConvolutionParameteriv(GL_SEPARABLE_2D, GL_MAX_CONVOLUTION_HEIGHT, &convolution);
printf("Maximum height of 2D separable convolution filter %d\n", convolution);
```

3.3. Current values and associated data

There are many *current* values you may have an interest in. These values are often changed by the application programmer, for example, color or vertex normal, and you may need to know the current values. This code will simply return their default values.

41a ⟨*Current values* 41a⟩≡

```
glGetIntegerv(GL_CURRENT_COLOR, iColor);
printf("The current (integer) color: (%d, %d, %d)\n", iColor[0], iColor[1], iColor[2]);

glGetFloatv(GL_CURRENT_COLOR, fColor);
printf("The current (floating point) color: (%f, %f, %f)\n", fColor[0], fColor[1], fColor

glGetFloatv(GL_CURRENT_NORMAL, normal);
printf("The current vertex normal vector: (%f, %f, %f)\n", normal[0], normal[1], normal[2
```

3.4. Transformation state variables

It may be useful for that application programmer to know the contents of some particular matrix, or the size of the viewport, or other attributes associated with tranformations.

42a ⟨*Transformation state variables* 42a⟩≡

```
glGetFloatv(GL_MODELVIEW_MATRIX, matrix);
printf("The model view matrix is\n");
for (i = 0; i < 4; i++) {
  printf("\t[");
  for (j = 0; j < 4; j++) {
    printf("%f\t",matrix[4*i+j]);
  }
  printf("]\n");
}

glGetIntegerv(GL_VIEWPORT, viewport);
printf("Viewport origin (%d, %d)", viewport[0], viewport[1]);
printf("\tViewport width = %d, height= %d\n", viewport[2], viewport[3]);

glGetIntegerv(GL_MATRIX_MODE, &matrixMode);
if (GL_MODELVIEW == matrixMode) {printf("MODELVIEW matrix mode\n");}
else if (GL_PROJECTION == matrixMode) {printf("PROJECTION matrix mode\n");}
else if (GL_TEXTURE == matrixMode) {printf("TEXTURE matrix mode\n");}
else {printf("Unknown matrix mode\n");}
```

3.5. Lighting state variables

Lighting and related modes start off disabled, but we can still query their status.

43a ⟨*Lighting state variables* 43a⟩≡

```
if (glIsEnabled(GL_LIGHTING)) { printf("Lighting is enabled\n"); }
else { printf("Lighting is not enabled\n"); }

if (glIsEnabled(GL_COLOR_MATERIAL)) { printf("Color tracking is enabled\n"); }
else { printf("Color tracking is not enabled\n"); }

glGetBooleanv(GL_LIGHT_MODEL_LOCAL_VIEWER, &localViewer);
if(localViewer) { printf("Viewer is local\n"); }
else { printf("Viewer is at infinity\n"); }

glGetMaterialfv(GL_FRONT, GL_AMBIENT, ambientColor);
printf("Front face ambient material color: (%f, %f, %f, %f)\n",
            ambientColor[0], ambientColor[0], ambientColor[2], ambientColor[3]);

glGetMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseColor);
printf("Front face diffuse material color: (%f, %f, %f, %f)\n",
            diffuseColor[0], diffuseColor[0], diffuseColor[2], diffuseColor[3]);

glGetMaterialfv(GL_FRONT, GL_SPECULAR, specularColor);
printf("Front face specular material color: (%f, %f, %f, %f)\n",
            specularColor[0], specularColor[0], specularColor[2], specularColor[3]);

glGetMaterialfv(GL_FRONT, GL_EMISSION, emissionColor);
printf("Front face emissive material color: (%f, %f, %f, %f)\n",
            emissionColor[0], emissionColor[0], emissionColor[2], emissionColor[3]);

glGetMaterialfv(GL_FRONT, GL_SHININESS, &shininess);
printf("Front face material specular exponent: %f\n", shininess);
```

```
glGetLightfv(GL_LIGHT0, GL_AMBIENT, ambientColor);
printf("Light zero ambient color: (%f, %f, %f, %f)\n",
            ambientColor[0], ambientColor[0], ambientColor[2], ambientColor[3]);

glGetLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseColor);
printf("Light zero diffuse color: (%f, %f, %f, %f)\n",
            diffuseColor[0], diffuseColor[0], diffuseColor[2], diffuseColor[3]);

glGetLightfv(GL_LIGHT0, GL_SPECULAR, specularColor);
printf("Light zero specular color: (%f, %f, %f, %f)\n",
            specularColor[0], specularColor[0], specularColor[2], specularColor[3]);

glGetLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
printf("Light zero position: (%f, %f, %f, %f)\n",
            lightPosition[0], lightPosition[0], lightPosition[2], lightPosition[3]);

glGetLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, &constantAttenuation);
printf("Constant attentuation coefficient: %f\n", constantAttenuation);

glGetLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION, &linearAttenuation);
printf("Linear attentuation coefficient: %f\n", linearAttenuation);

glGetLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, &quadraticAttenuation);
printf("Quadratic attentuation coefficient: %f\n", quadraticAttenuation);
```

3.6. Rasterization state variables

Control of how objects are pixelized can be determined by querying these rasterization state variables.

45a ⟨*Rasterization state variables* 45a⟩≡

```
if (glIsEnabled(GL_POINT_SMOOTH)) { printf("Point antialiasing is on\n"); }
else { printf("Point antialiasing is off\n"); }

if (glIsEnabled(GL_LINE_SMOOTH)) { printf("Line antialiasing is on\n"); }
else { printf("Line antialiasing is off\n"); }

if (glIsEnabled(GL_POLYGON_SMOOTH)) { printf("Polygno antialiasing is on\n"); }
else { printf("Polygon antialiasing is off\n"); }

if (glIsEnabled(GL_CULL_FACE)) { printf("Polygon culling is enabled\n"); }
else { printf("Polygon culling is disabled\n"); }

glGetFloatv(GL_POINT_SIZE, pointSizes);
printf("Point size in pixels: %f\n", pointSizes[0]);

glGetFloatv(GL_LINE_WIDTH, lineSizes);
printf("Line width in pixels: %f\n", lineSizes[0]);

glGetIntegerv(GL_FRONT_FACE, &faceType);
if (GL_CCW == faceType) { printf("Counterclockwise front faces\n"); }
else { printf("Clockwise front faces\n"); }

glGetIntegerv(GL_POLYGON_MODE, &fillMode);
if (GL_FILL == fillMode) { printf("Fill front and back polygons\n"); }
else { printf("Don't fill front and back polygons\n"); }

glGetIntegerv(GL_CULL_FACE_MODE, &cullMode);
```

```
if (GL_BACK == cullMode) { printf("Cull back faces\n"); }
else { printf("Don't cull back faces\n"); }
```

3.7. Hello Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/state.c

4. Basic OpenGL Primitives

This program illustrates the basic graphics primitives supported by OpenGL and GLUT. It is not from the OpenGL Red Book Woo et al. [1999], but fits nicely with the development of programs from that book.

47a ⟨*primitives* 47a⟩≡
⟨*Primitives header files* 47b⟩
⟨`display` *primitives* 50a⟩
⟨*Primitives initialization function* 63a⟩
⟨*Primitives* `main` *function* 48a⟩

4.1. Headers and routine variables

We'll again use GLUT routines to manage windows.

47b ⟨*Primitives header files* 47b⟩≡

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

#define OK 0
```

4.2. A Main function

The `main` function is similar to previous programs, for example Hello, World! and Double Buffering!.

48a ⟨*Primitives* `main` *function* 48a⟩≡

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(250, 250);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("OpenGL Primitives");
   init();
   glutDisplayFunc(display);
   glutMainLoop();
   return OK;
}
```

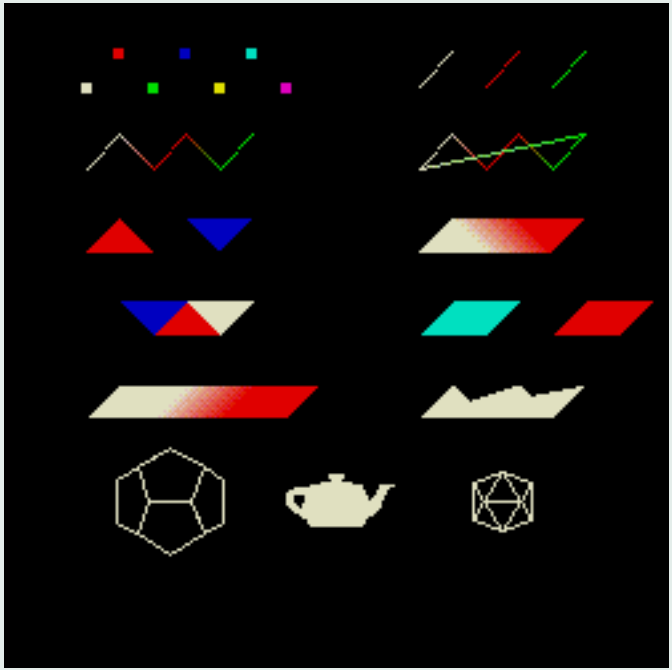

Figure 2: OpenGL and GLUT Primitives.

4.3. Displaying the Primitives

The `display` function defines a list of eight 3D coordinates and uses these to define the basic geometric primitives supported by OpenGL: points, lines, line strips and loops, triangles, triangle strips and fans, quadrangles, quadrangle strips, and polygons. Then `display` renders the GLUT supported "primitives." See the lectures on primitives and models for additional discussion of these topics.

The eight 3D points start at the origin and jump up and down at ten unit intervals creating a "saw-tooth" pattern. The rendered images are shown in Figure 2. Then several of the GLUT primitives are rendered.

50a ⟨`display` *primitives* 50a⟩≡

```
void display(void) {
   GLfloat vertex[8][3] = {{ 0.0, 0.0, 0.0}, {10.0, 10.0, 0.0},
                           {20.0, 0.0, 0.0}, {30.0, 10.0, 0.0},
                           {40.0, 0.0, 0.0}, {50.0, 10.0, 0.0},
                           {60.0, 0.0, 0.0}, {70.0, 10.0, 0.0}};

   glClear(GL_COLOR_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   ⟨Display points 52a⟩
   ⟨Display lines 53a⟩
   ⟨Display line strips 54a⟩
   ⟨Display line loop 55a⟩
   ⟨Display triangles 56a⟩
   ⟨Display triangle strip 57a⟩
   ⟨Display triangle fan 58a⟩
   ⟨Display quads 59a⟩
   ⟨Display quad strip 60a⟩
   ⟨Display polygon 61a⟩
   ⟨Display GLUT models 62a⟩
```


```
    glFlush();
}
```

The display of each primitive follows the same pattern:

1. Duplicate the top of the `MODELVIEW` matrix stack;
2. Concatenate the top of the stack with a translation;
3. Define the primitive;
4. Pop the current matrix off the stack.

We'll start with points and make them 4×4 pixels so they are more visible.

52a ⟨*Display points* 52a⟩≡

```
glPushMatrix();
glTranslatef(-75.0, 75.0, 0.0);
glPointSize(4.0);
glBegin(GL_POINTS);
  glColor3f(1.0, 1.0, 1.0);  // white
  glVertex3fv(vertex[0]);
  glColor3f(1.0, 0.0, 0.0);  // red
  glVertex3fv(vertex[1]);
  glColor3f(0.0, 1.0, 0.0);  // green
  glVertex3fv(vertex[2]);
  glColor3f(0.0, 0.0, 1.0);  // blue
  glVertex3fv(vertex[3]);
  glColor3f(1.0, 1.0, 0.0);  // yellow
  glVertex3fv(vertex[4]);
  glColor3f(0.0, 1.0, 1.0);  // cyan
  glVertex3fv(vertex[5]);
  glColor3f(1.0, 0.0, 1.0);  // magenta
  glVertex3fv(vertex[6]);
glEnd();
glPopMatrix();
```

Lines are the displayed as described above, except we'll change the color to identify each line clearly. Each consecutive pair of vertices defines a line.

53a ⟨*Display lines* 53a⟩≡

```
glPushMatrix();
glTranslatef(25.0, 75.0, 0.0);
glBegin(GL_LINES);
  glColor3f(1.0, 1.0, 1.0);  // white
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[1]);
  glColor3f(1.0, 0.0, 0.0); // red
  glVertex3fv(vertex[2]);
  glVertex3fv(vertex[3]);
  glColor3f(0.0, 1.0, 0.0); // green
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[5]);
glEnd();
glPopMatrix();
```

Line strips are like lines, except vertex is connected to the next. Notice in the rendering in Figure 2 how the color is interpolated across the lines, using the smooth (Gouraud) shading method.

54a ⟨*Display line strips* 54a⟩≡

```
glPushMatrix();
glTranslatef(-75.0, 50.0, 0.0);
glBegin(GL_LINE_STRIP);
glColor3f(0.0, 1.0, 0.0);
  glColor3f(1.0, 1.0, 1.0);
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[1]);
  glColor3f(1.0, 0.0, 0.0);
  glVertex3fv(vertex[2]);
  glVertex3fv(vertex[3]);
  glColor3f(0.0, 1.0, 0.0);
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[5]);
glEnd();
glPopMatrix();
```

Line loops are like line strips, except the first and last vertex are connected. We'll change the color from white to red to green to blue.

55a ⟨*Display line loop* 55a⟩≡

```
glPushMatrix();
glTranslatef(25.0, 50.0, 0.0);
glBegin(GL_LINE_LOOP);
  glColor3f(1.0, 1.0, 1.0); // white
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[1]);
  glColor3f(1.0, 0.0, 0.0); // red
  glVertex3fv(vertex[2]);
  glVertex3fv(vertex[3]);
  glColor3f(0.0, 1.0, 0.0); // green
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[5]);
  glColor3f(0.0, 0.0, 1.0); // blue
glEnd();
glPopMatrix();
```

Triangles are defined by consecutive triples of points. The first triangle will be filled in red the next in blue. Notice the clockwise ordering of the vertices. What face are we seeing?

56a ⟨*Display triangles* 56a⟩≡

```
glPushMatrix();
glTranslatef(-75.0, 25.0, 0.0);
glBegin(GL_TRIANGLES);
  glColor3f(1.0, 0.0, 0.0); // red
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[1]);
  glVertex3fv(vertex[2]);
  glColor3f(0.0, 0.0, 1.0); // blue
  glVertex3fv(vertex[3]);
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[5]);
glEnd();
glPopMatrix();
```

Triangle strips connect successive points as triangles. We'll change the color of vertices. Notice in the rendering in Figure 2 how the color is interpolated across the polygon, using the smooth (Gouraud) shading method.

57a ⟨*Display triangle strip* 57a⟩≡

```
glPushMatrix();
glTranslatef(25.0, 25.0, 0.0);
glBegin(GL_TRIANGLE_STRIP);
  glColor3f(1.0, 1.0, 1.0);
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[1]);
  glVertex3fv(vertex[2]);
  glColor3f(1.0, 0.0, 0.0);
  glVertex3fv(vertex[3]);
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[5]);
glEnd();
glPopMatrix();
```

Triangle fans connect points as triangles from the first *source* vertex. For the vertices defined, we'll fan triangles out from `vertex[3]`. Let's change the shading model to `GL_FLAT`.

58a ⟨*Display triangle fan* 58a⟩≡

```
glShadeModel(GL_FLAT);
glPushMatrix();
glTranslatef(-75.0, 0.0, 0.0);
glBegin(GL_TRIANGLE_FAN);
  glColor3f(1.0, 1.0, 1.0); // first one white
  glVertex3fv(vertex[3]);
  glVertex3fv(vertex[5]);
  glVertex3fv(vertex[4]);
  glColor3f(1.0, 0.0, 0.0); // next one red
  glVertex3fv(vertex[2]);
  glColor3f(0.0, 0.0, 1.0); // last one green
  glVertex3fv(vertex[1]);
glEnd();
glPopMatrix();
```

Quadrangle have four vertices. Here's a cyan one and a red one. Notice counter-clockwise ordering of the vertices.

59a ⟨*Display quads* 59a⟩≡

```
glPushMatrix();
glTranslatef(25.0, 0.0, 0.0);
glBegin(GL_QUADS);
  glColor3f(0.0, 1.0, 1.0);
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[2]);
  glVertex3fv(vertex[3]);
  glVertex3fv(vertex[1]);
  glColor3f(1.0, 0.0, 0.0);
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[6]);
  glVertex3fv(vertex[7]);
  glVertex3fv(vertex[5]);
glEnd();
glPopMatrix();
```

Quadrangle strips are formed from consecutive quadruples of points.

60a ⟨*Display quad strip* 60a⟩≡

```
glShadeModel(GL_SMOOTH);
glPushMatrix();
glTranslatef(-75.0, -25.0, 0.0);
glBegin(GL_QUAD_STRIP);
  glColor3f(1.0, 1.0, 1.0);
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[1]);
  glVertex3fv(vertex[2]);
  glVertex3fv(vertex[3]);
  glColor3f(1.0, 0.0, 0.0);
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[5]);
  glVertex3fv(vertex[6]);
  glVertex3fv(vertex[7]);
glEnd();
glPopMatrix();
```

This is not a *good* polygon! It is not convex and it is not simple. Edges cross other edges. OpenGL does not provide any guarantee about the rendering of such ill-defined polygons.

61a ⟨*Display polygon* 61a⟩≡

```
glPushMatrix();
glTranslatef(25.0, -25.0, 0.0);
glBegin(GL_POLYGON);
  glColor3f(1.0, 1.0, 1.0);
  glVertex3fv(vertex[0]);
  glVertex3fv(vertex[1]);
  glVertex3fv(vertex[2]);
  glVertex3fv(vertex[3]);
  glVertex3fv(vertex[4]);
  glVertex3fv(vertex[5]);
glEnd();
glPopMatrix();
```

Now, we'll render the some of the GLUT models. In particular, the *famous* Utah teapot, an a couple of the *Platonic* solids. Other GLUT models not shown include: spheres, tori, cones, cubes, ocathedra, and tetrahedra. These models can be rendered in wire frame or as solids.

62a ⟨*Display GLUT models* 62a⟩≡

```
glPushMatrix();
glTranslatef(0.0, -50.0, 0.0);
glutWireTeapot(10.0);  // The Utah Teapot
glPopMatrix();
glPushMatrix();
glScalef(10.0, 10.0, 10.0);
glTranslatef(-5.0, -5.0, 0.0);
glutWireDodecahedron();  // The twelve sided Platonic solid
glPopMatrix();
glPushMatrix();
glScalef(10.0, 10.0, 10.0);
glTranslatef(5.0, -5.0, 0.0);
glutWireIcosahedron(); // The twenty sided Platonic solid
glPopMatrix();
```

4.4. An Initialization function

All the `init` function will do is set the background color to black and set a flat shading model.

63a ⟨*Primitives initialization function* 63a⟩≡

```
void init(void) {
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-100.0, 100.0, -100.0, 100.0, -100.0, 100.0);
}
```

4.5. Primitives Source Code

The C program code is available at:

http://www.cs.fit.edu/wds/classes/prog-graphics/src/primitives.c

5. GLUT Models

This program illustrates the models by GLUT. It is not from the OpenGL Red Book Woo et al. [1999], but fits nicely with the development of programs from that book.

64a ⟨*glutmodels* 64a⟩≡
```
⟨Models header files 64b⟩
⟨Set drawing parameters 66a⟩
⟨display models 68a⟩
⟨Models initialization function 76b⟩
⟨Models main function 65a⟩
```

5.1. Headers and routine variables

We'll again use GLUT routines to manage windows. To make the program somewhat input driven we'll define some global variables that could be set from command line or via some dynamic input event. Since OpenGL functions and callback do not support sharing of data, the variables must be global.

64b ⟨*Models header files* 64b⟩≡
```
#include <GL/glut.h>

#define OK 0

GLdouble radius;       // a radius for spheres, bases of cones, and size of the teapot
GLdouble height;       // heights of cones
GLint slices, stacks;  // slices and stacks for spheres cones, and tori
GLdouble outerradius;   // for tori
```

5.2. A Main function

The `main` function is similar to previous programs, for example Hello, World! and Double Buffering!. We will write a function `setParameters()` to set the parameters defined above.

65a ⟨*Models* `main` *function* 65a⟩≡

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(250, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("GLUT Models");
   setParameters();
   init();
   glutDisplayFunc(display);
   glutMainLoop();
   return OK;
}
```

5.3. Displaying the Models

Here we'll just set them to reasonable values and encourage you to see how alterating them affects their display.

66a ⟨*Set drawing parameters* 66a⟩≡

```
void setParameters(void) {
 radius = 15.0;
 height = 25.0;
 slices = 15;
 stacks = 15;
 outerradius = 25;
}
```

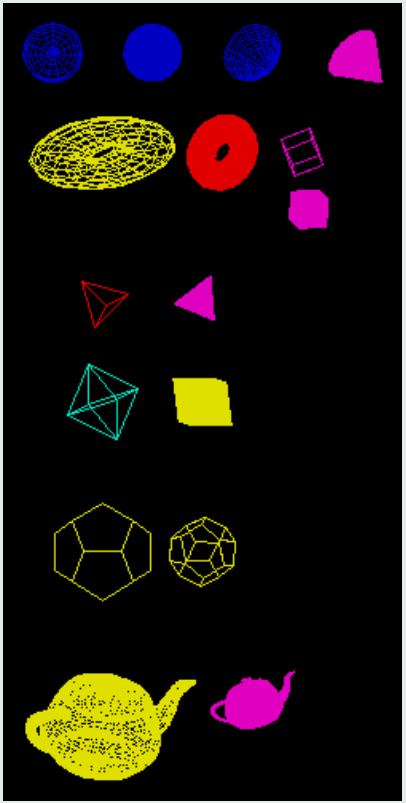

Figure 3: GLUT Models.

5.4. Displaying the Models

The `display` function renders the GLUT supported models: spheres, cones, tori, tetrahedra, hexahedra, octahedra, dodecahedra, icosahedra, and the teapots. See the lectures on models for additional discussion of these topics. The rendered images are shown in Figure 3.

68a ⟨`display` *models* 68a⟩≡

```
void display(void) {
   glClear(GL_COLOR_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   ⟨Display spheres 69a⟩
   ⟨Display cones 70a⟩
   ⟨Display tori 71a⟩
   ⟨Display tetrahedra 73a⟩
   ⟨Display hexahedra 72a⟩
   ⟨Display octahedra 74a⟩
   ⟨Display dodecahedra 75a⟩
   ⟨Display icosahedra 75b⟩
   ⟨Display teapots 76a⟩
   glFlush();
}
```

The display of each model follows the same pattern:

1. Duplicate the top of the `MODELVIEW` matrix stack;
2. Concatenate the top of the stack with a translation;
3. Define the models;
4. Pop the current matrix off the stack.

We'll start with spheres: One wire frame, one solid and draw them at the upper left of the screen using a simple translation.

69a ⟨*Display spheres* 69a⟩≡

```
glColor3f(0.0, 0.0, 1.0);
glPushMatrix();
  glTranslatef(-75.0, 175.0, 0.0);
  glutWireSphere(radius, slices, stacks);
  glTranslatef(50.0, 0.0, 0.0);
  glutSolidSphere(radius, slices, stacks);
glPopMatrix();
```

Next, we'll draw some cones and apply some rotation to them so you'll see more than a circle (the cone is drawn along the z axis).

The order of matrix operations is important. The `MODELVIEW` stack now has an identity matrix I on top — it started that way some translations were pushed above, but they were popped off. In the code below, a translate matrix T multiples the top of the stack and then a rotation matrix R multiples the top of the stack. Thus, before the cone, call it c, is drawn the vertices that define it are multiplied by TR to create a new cone TRc. This positions in the cone in an entirely different location than the cone RTc — try it!

Most often, you'll want to scale or rotate vertices prior to translating them. This means the `glTranlate*()` call should be first — stack are first-in, last-out.

By the way, the rotation R is by 30° about the axis $\langle 1, 1, 0 \rangle$.

70a ⟨*Display cones* 70a⟩≡

```
glPushMatrix();
  glTranslatef(25.0, 175.0, 0.0);
  glRotatef(30.0, 1.0, 1.0, 0.0);
  glutWireCone(radius, height, slices, stacks);
glPopMatrix();

glColor3f(1.0, 0.0, 1.0);
glPushMatrix();
  glTranslatef(75.0, 175.0, 0.0);
  glRotatef(60.0, 1.0, 1.0, 0.0);
  glutSolidCone(radius, height, slices, stacks);
glPopMatrix();
```

Now we'll draw a wire and solid torus. Let's make the wire one yellow and the solid one red, both a little smaller than the default `radius`, and each tipped a bit.

71a ⟨*Display tori* 71a⟩≡

```
glColor3f(1.0, 1.0, 0.0);
glPushMatrix();
  glTranslatef(-50.0, 125.0, 0.0);
  glScalef(1.0, 0.5, 0.5);
  glRotatef(45.0, 1.0, 1.0, 0.0);
  glutWireTorus(radius, outerradius, slices, stacks);
glPopMatrix();

glColor3f(1.0, 0.0, 0.0);
glPushMatrix();
  glTranslatef(10.0, 125.0, 0.0);
  glScalef(0.5, 0.5, 0.5);
  glRotatef(45.0, 1.0, 1.0, 1.0);
  glutSolidTorus(radius, outerradius, slices, stacks);
glPopMatrix();
```

The hexahedron is a fancy name for a cube. To see more than a square, we'll rotate it a bit.

72a ⟨*Display hexahedra* 72a⟩≡

```
glPushMatrix();
  glTranslatef(50.0, 125.0, 0.0);
  glRotatef(30.0, 1.0, 0.0, 1.0);
  glutWireCube(radius);
glPopMatrix();

glColor3f(1.0, 0.0, 1.0);
glPushMatrix();
  glRotatef(30.0, 1.0, 1.0, 0.0);
  glTranslatef(50.0, 100.0, 0.0);
  glutSolidCube(radius);
glPopMatrix();
```

Here's a 4-sided *pyramid*, each face an equilateral triangle. It is centered at the origin and fits in a bounding sphere with radius $\sqrt{3} \approx 1.73$. Since its defining call has no arguments, we must scale its size by using a modeling matrix.

73a ⟨*Display tetrahedra* 73a⟩≡

```
glPushMatrix();
  glTranslatef(-50.0, 50.0, 0.0);
  glScalef(15.0, 15.0, 15.0);
  glRotatef(45.0, 1.0, 1.0, 0.0);
  glutWireTetrahedron();
glPopMatrix();

glColor3f(1.0, 0.0, 1.0);
glPushMatrix();
  glTranslatef(0.0, 50.0, 0.0);
  glScalef(15.0, 15.0, 15.0);
  glRotatef(60.0, 0.0, 1.0, 1.0);
  glutSolidTetrahedron();
glPopMatrix();
```

The octahedron is an 8-sided object with each face an equilateral triangle. It fits in a bounding unit sphere centered at the origin. We'll leave the solid one centered at the origin.

74a ⟨*Display octahedra* 74a⟩≡

```
glColor3f(0.0, 1.0, 1.0);
glPushMatrix();
  glTranslatef(-50.0, 0.0, 0.0);
  glScalef(20.0, 20.0, 20.0);
  glRotatef(30.0, 0.0, 1.0, 1.0);
  glutWireOctahedron();
glPopMatrix();

glColor3f(1.0, 1.0, 0.0);
glPushMatrix();
  glScalef(20.0, 20.0, 20.0);
  glRotatef(-60.0, 1.0, 0.0, 1.0);
  glutSolidOctahedron();
glPopMatrix();
```

The dodecahedron is a 12-sided solid.

75a ⟨*Display dodecahedra* 75a⟩≡

```
glColor3f(1.0, 1.0, 0.0);
glPushMatrix();
  glTranslatef(-50.0, -75.0, 0.0);
  glScalef(15.0, 15.0, 15.0);
  glutWireDodecahedron();
glPopMatrix();

glColor3f(1.0, 1.0, 0.0);
glPushMatrix();
  glTranslatef(0.0, -75.0, 0.0);
  glScalef(10.0, 10.0, 10.0);
  glRotatef(-60.0, 1.0, 0.0, 1.0);
  glutWireDodecahedron();
glPopMatrix();
```

The icosahedron is a 20-sided solid.

75b ⟨*Display icosahedra* 75b⟩≡

```
glPushMatrix();
  glScalef(10.0, 10.0, 10.0);
  glTranslatef(-50.0, -125.0, 0.0);
  glutWireIcosahedron();
glPopMatrix();
```

Finally, GLUT support the Utah teapot — the modern Platonic solid.

76a ⟨*Display teapots* 76a⟩≡

```
glPushMatrix();
  glTranslatef(-50.0, -160.0, 0.0);
  glRotatef(45.0, 1.0, 1.0, 0.0);
  glutWireTeapot(2.0*radius);
glPopMatrix();

glColor3f(1.0, 0.0, 1.0);
glPushMatrix();
  glTranslatef(25.0, -150.0, 0.0);
  glRotatef(30.0, 1.0, 1.0, 1.0);
  glutSolidTeapot(radius);
glPopMatrix();
```

5.5. An Initialization function

All the `init` function will do is set the background color to black, set the shading model to (Gouraud) smooth, which is not necessary since it is the default, and set up an orthographic view in a $200 \times 200 \times 200$ volume centered at the origin.

76b ⟨*Models initialization function* 76b⟩≡

```
void init(void) {
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_SMOOTH);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-100.0, 100.0, -200.0, 200.0, -100.0, 100.0);
}
```

5.6. Modelss Source Code

The C program code is available at:

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/glutmodels.c

6. Modeling a Cube

This program illustrates modeling of a cube.

78a ⟨*cube* 78a⟩≡
```
⟨Cube header files 78b⟩
⟨display cube 80a⟩
⟨Cube initialization function 82b⟩
⟨Cube main function 79a⟩
```

6.1. Headers and routine variables

We'll again use GLUT routines to manage windows.

78b ⟨*Cube header files* 78b⟩≡
```
#include <GL/glut.h>

#define OK 0
```

6.2. A Main function

The `main` function is similar to previous programs, for example Hello, World! or Double Buffering!. The significant difference is that we'll enable depth buffering. This is crucial since the cube will be drawn as a collection of triangles and they may be processed in an arbitrary order from the view point. Depth buffering will ensure that only the triangles visible from the view point will be drawn.

79a ⟨*Cube* `main` *function* 79a⟩≡

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DEPTH | GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(250, 250);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Modeling a Cube");
   init();
   glutDisplayFunc(display);
   glutMainLoop();
   return OK;
}
```

6.3. Displaying the Cube

When displaying the cube we clear the depth buffer as well as the color (frame) buffer. The cube is drawn as a collection of triangles.

80a ⟨display *cube* 80a⟩≡

```
void display(void) {
 int i;
 ⟨Define the vertices 80b⟩
 ⟨Define the triangles 81a⟩

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
 glColor3f(0.0, 0.0, 0.0);
 ⟨Draw its faces as triangles 82a⟩
}
```

@ Figure 4 show a cube with vertices numbered v_0 to v_7. Here we'll center the cube at the origin and place its vertices one unit from the origin.

The vertices are defined to be one unit from the origin. That means they are at $(\pm X, \pm X, \pm X)$ where $X = 1/\sqrt{3}$.

80b ⟨Define the vertices 80b⟩≡

```
#define X 0.5773502692  // 1 over the square root of 3

static GLfloat vertices[8][3] = {
  {-X, -X, -X},   {-X, -X, X}, {X, -X, -X}, {X, -X, X},
  {X, X, -X},   {X, X, X}, {-X, X, -X}, {-X, X, X}
};
```

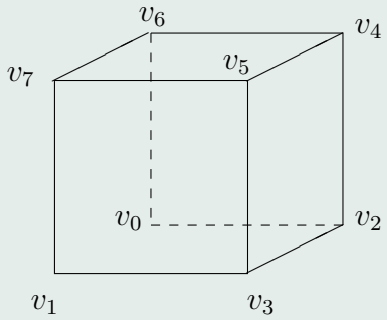

Figure 4: The Unit Cube

The cube was modeled in the lecture notes on the graphics pipeline using quad strips. Here well use triangles. There are six faces and each will be devided into two triangles. The orientation of each triangle will be counterclockwise. Only the indices into the the `vertices` array need to be listed. You need to be careful here and think this through.

81a ⟨*Define the triangles* 81a⟩≡

```
static GLuint triIndices[12][3] = {
 {0, 3, 1}, {0, 2, 3}, // bottom face
 {2, 4, 5}, {2, 5, 3}, // right side face
 {1, 5, 7}, {1, 3, 5}, // front side face
 {1, 7, 6}, {1, 6, 0}, // left side face
 {0, 6, 4}, {0, 4, 2}, // back side face
 {4, 5, 7}, {4, 7, 6}, // top face
};
```

Now we can draw the triangles. To see each of them we'll draw them in shades of redy.

82a ⟨*Draw its faces as triangles* 82a⟩≡

```
glRotatef(45.0, 1.0, 1.0, 1.0);
glBegin(GL_TRIANGLES);
  for (i = 0; i < 12; i++) {
    glColor3f((float) i/((float) i + 3.0), 0.0, 0.0);
    glVertex3fv(&vertices[triIndices[i][0]][0]);
    glVertex3fv(&vertices[triIndices[i][1]][0]);
    glVertex3fv(&vertices[triIndices[i][2]][0]);
  }
glEnd();
glFlush();
```

6.4. An Initialization function

All the `init` function will do is set the background color to black, set the shading model to (Gouraud) smooth, which is not necessary since it is the default, and set up an orthographic view in a $200 \times 200 \times 200$ volume centered at the origin.

82b ⟨*Cube initialization function* 82b⟩≡

```
void init(void) {
   glClearColor(1.0, 1.0, 1.0, 0.0);
   glShadeModel(GL_FLAT);
   glMatrixMode(GL_PROJECTION);
   glEnable(GL_DEPTH_TEST);
   glLoadIdentity();
   glOrtho(-2.0, 2.0, -2.0, 2.0, -2.0, 2.0);
}
```

6.5. Cube Source Code

The C program code is available at:

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/cube.c

7. Modeling a Sphere by Recursive Subdivision

This program illustrates how a sphere can be modeled starting from a cube and recursively subdividing its faces until a sphere is approximated. We'll need some functions to make this work. One will normalize vectors so they are unit length and lie on a unit sphere. Another will draw one triangle, setting vertex normals as a vector from the origin to the vertex itself, since sphere surface normals point from the origin to the sphere. Another will perform the recursive subdivision described more fully below. Figure 5 show the cube, it divided into 48, then 196, then 784 triangles.

84a ⟨*subdivide* 84a⟩≡

```
⟨Subdivide header files 84b⟩
⟨Normalize a vector 93a⟩
⟨Draw a triangle 94a⟩
⟨Recursive subdivision function 91a⟩
⟨display sphere by recursion 87a⟩
⟨Subdivide initialization function 90b⟩
⟨Subdivide main function 86a⟩
```

7.1. Headers and routine variables

We'll again use GLUT routines to manage windows. A little error checking is done so we'll use some output routines prototyped in the standard library.

84b ⟨*Subdivide header files* 84b⟩≡

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

#define OK 0
```

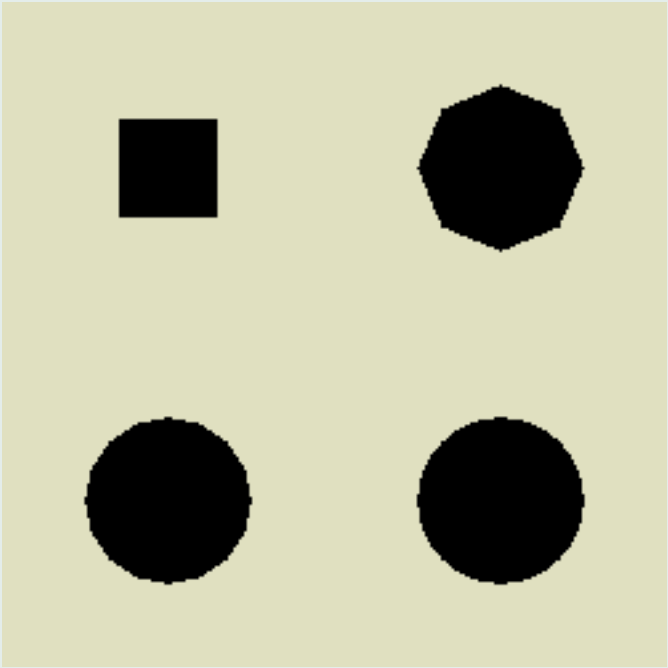

Figure 5: The Unit Cube Approximating a Sphere by Recursive Subdivision

7.2. A Main function

The `main` function is similar to previous programs, for example Modeling a Cube program.

86a ⟨*Subdivide* `main` *function* 86a⟩≡

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DEPTH | GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(250, 250);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Modeling a Sphere");
   init();
   glutDisplayFunc(display);
   glutMainLoop();
   return OK;
}
```

7.3. Displaying the Sphere

The `display` functions will draw 4 spheres: One the original cube of 12 triangles. The second an approximation formed from the cube and made of 48 triangle. Then a 196 triangle sphere and 784 triangle sphere. To better see the faces, we should enable lighting.

The original sphere is from the Modeling a Cube program. See Figure 4.

87a ⟨`display` *sphere by recursion* 87a⟩≡

```
void display(void) {
 int i;
 ⟨Define the vertices 88a⟩
 ⟨Define the triangles (never defined)⟩

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glMatrixMode(GL_MODELVIEW);
 glLoadIdentity();
 glColor3f(0.0, 0.0, 0.0);
 ⟨Draw a cube as an approximation (12 triangles) 89a⟩
 ⟨Subdivide the cube to 48 triangles 89b⟩
 ⟨Subdivide again to 196 triangles 89c⟩
 ⟨Subdivide again to 784 triangles 90a⟩
 glFlush();
}
```

The original sphere is from the Modeling a Cube program. See Figure 4.

The vertices are defined to be one unit from the origin. That means they are at $(\pm X, \pm X, \pm X)$ where $X = 1/\sqrt{3}$.

88a ⟨*Define the vertices* 88a⟩≡

```
#define X 0.5773502692  // 1 over the square root of 3

static GLfloat vertices[8][3] = {
  {-X, -X, -X},   {-X, -X, X}, {X, -X, -X}, {X, -X, X},
  {X, X, -X},   {X, X, X}, {-X, X, -X}, {-X, X, X}
};
```

The vertices can be defined by listing the indices into `vertices` that form the triangles.

88b ⟨*define the triangles* 88b⟩≡

```
static GLuint triIndices[12][3] = {
 {0, 3, 1}, {0, 2, 3}, // bottom face
 {2, 4, 5}, {2, 5, 3}, // right side face
 {1, 5, 7}, {1, 3, 5}, // front side face
 {1, 7, 6}, {1, 6, 0}, // left side face
 {0, 6, 4}, {0, 4, 2}, // back side face
 {4, 5, 7}, {4, 7, 6}, // top face
};
```

Now we draw the original cube using the `subdivide` function with a depth of 0, see below.

89a ⟨*Draw a cube as an approximation (12 triangles)* 89a⟩≡

```
glTranslatef(-2.0, 2.0, 0.0);
for (i = 0; i < 12; i++) {
  subdivide(&vertices[triIndices[i][0]][0],
            &vertices[triIndices[i][1]][0],
            &vertices[triIndices[i][2]][0], 0);
}
```

Subdividing each triangle into 4 yields a better approximation to the unit sphere.

89b ⟨*Subdivide the cube to 48 triangles* 89b⟩≡

```
glTranslatef(4.0, 0.0, 0.0);
for (i = 0; i < 12; i++) {
  subdivide(&vertices[triIndices[i][0]][0],
            &vertices[triIndices[i][1]][0],
            &vertices[triIndices[i][2]][0], 1);
}
```

Subdividing twice yields a 196 triangular approximation to the unit sphere.

89c ⟨*Subdivide again to 196 triangles* 89c⟩≡

```
glTranslatef(-4.0, -4.0, 0.0);
for (i = 0; i < 12; i++) {
  subdivide(&vertices[triIndices[i][0]][0],
            &vertices[triIndices[i][1]][0],
            &vertices[triIndices[i][2]][0], 2);
}
```

One last time gives a "pretty good sphere."

90a ⟨*Subdivide again to 784 triangles* 90a⟩≡

```
glTranslatef(4.0, -0.0, 0.0);
for (i = 0; i < 12; i++) {
  subdivide(&vertices[triIndices[i][0]][0],
            &vertices[triIndices[i][1]][0],
            &vertices[triIndices[i][2]][0], 3);
}
```

7.4. An Initialization function

All the `init` function will do is set the background color to black, set the shading model to (Gouraud) smooth, which is not necessary since it is the default, and set up an orthographic view in a $200 \times 200 \times 200$ volume centered at the origin.

90b ⟨*Subdivide initialization function* 90b⟩≡

```
void init(void) {
   glClearColor(1.0, 1.0, 1.0, 0.0);
   glMatrixMode(GL_PROJECTION);
   glEnable(GL_FLAT);
   glEnable(GL_DEPTH_TEST);
   glLoadIdentity();
   glOrtho(-4.0, 4.0, -4.0, 4.0, -4.0, 4.0);
}
```

7.5. Subdivision

The cube can be thought of as a very crude approximation to a sphere with radius 1. The approximation can be improved by subdividing each triangle in the model and *pushing* the newly created vertices out to radius 1.

A recursive subdivision algorithm for this takes in three triangle vertices and a subdivision `depth`. If the `depth` has not been reached, it calculates the midpoint of each edge, pushes the new vertices to be unit length and calls itself again for each of the four new triangles with `depth` reduced by 1. The geometry of the subdivided triangle is shown in Figure 6

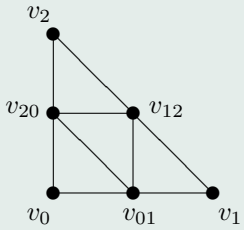

Figure 6: Subdivision of a triangle into four triangles.

The algorithm itself is simple. It takes 3 vertices and a depth parameter that controls the recursion. As shown in Figure 6, subdividing each triangle edge in the middle produces 4 sub-triangles and 3 new vertices.

If the recursive depth has been reached, the triangle is drawn. If not, new midpoints are computed, they are scaled to lie on the unit sphere, and the `subdivide()` routine is called for each of the 4 new triangle with the depth decreased by 1.

91a ⟨*Recursive subdivision function* 91a⟩≡

```
void subdivide(float *v0, float *v1, float *v2, long depth) {
  float v01[3], v12[3], v20[3];
  int i;
```

```
  ⟨Draw the triangle and return if depth reached 92a⟩
  ⟨Compute midpoints 92b⟩
  ⟨Push them out to lie on the sphere 93b⟩
  ⟨Call me again to draw the new triangles 93c⟩
}
```

Computer scientists must be perverted. A `depth` of zero signals a stop to the recursion.

92a ⟨*Draw the triangle and return if depth reached* 92a⟩≡

```
if (0 == depth) {
  drawTriangle(v0, v1, v2);
  return;
}
```

Midpoints are computed by averaging the original vertices.

92b ⟨*Compute midpoints* 92b⟩≡

```
for (i = 0; i < 3; i++) {
  v01[i] = (v0[i] + v1[i])/2.0;
  v12[i] = (v1[i] + v2[i])/2.0;
  v20[i] = (v2[i] + v0[i])/2.0;
}
```

Here's a simple function, called `normalize()`, that takes a triple and scales them so they form a unit length vector.

93a ⟨*Normalize a vector* 93a⟩≡

```
void normalize(float *v) {
  double d = sqrt((double) (v[0]*v[0]+v[1]*v[1]+v[2]*v[2]));
  if (0.0 == d) {
    printf("zero length vector");
    return;
  }
  v[0] /= d;
  v[1] /= d;
  v[2] /= d;
}
```

The function `normalize()` scales the vertices to be one unit from the origin, effectively placing them on the unit sphere. We want each of the newly formed vertices to lie on the unit sphere.

93b ⟨*Push them out to lie on the sphere* 93b⟩≡

```
normalize(v01);
normalize(v12);
normalize(v20);
```

As shown in Figure 4 there are now four triangles to draw. This is done recursively, decrementing the depth by one to eventually stop.

93c ⟨*Call me again to draw the new triangles* 93c⟩≡

```
subdivide(v0, v01, v20, depth-1);
subdivide(v01, v1, v12, depth-1);
subdivide(v12, v2, v20, depth-1);
subdivide(v01, v12, v20, depth-1);
```

7.6. Draw a Triangle

Now all that needs to be done is to draw each triangle.

94a ⟨*Draw a triangle* 94a⟩≡

```
void drawTriangle(float *v0, float *v1, float *v2) {
  glBegin(GL_TRIANGLES);
    glNormal3fv(v0);
    glVertex3fv(v0);
    glNormal3fv(v1);
    glVertex3fv(v1);
    glNormal3fv(v2);
    glVertex3fv(v2);
  glEnd();
}
```

7.7. Subdivision Source Code

The C program code is available at:

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/subdivide.c

8. Lines and Stipples

This program, from the Red book Woo et al. [1999], demonstrates geometric primitives and their attributes.

95a ⟨*lines* 95a⟩≡
⟨*Lines header files* 95b⟩
⟨*Lines* `display` *and redisplay* 97a⟩
⟨*Lines initialization function* 96a⟩
⟨*Lines* `reshape` *the window callback* 101a⟩
⟨*Lines* `keyboard` *event callback* 102a⟩
⟨*Lines* `main` *function* 103a⟩

8.1. Headers and routine variables

The GLUT library is used for window and I/O functions. A simple macro, `drawOneLine`, is defined as a shorthand notation to draw a single line. and numeric constant is defined to be `ESC` which signals that the *escape* key has been pressed to terminate the program.

95b ⟨*Lines header files* 95b⟩≡

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

#define drawOneLine(x1,y1,x2,y2)  glBegin(GL_LINES);  \
   glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();

#define OK 0
#define ESC 27  /* ASCII for the escape key */
```

8.2. An Initialization function

A simple initialization is still in use.

96a ⟨*Lines initialization function* 96a⟩≡

```
void init(void) {
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}
```

8.3. Displaying and Redisplaying

This `display` function clears the color buffer, sets the drawing color to white, enables line stippling, draws five rows of lines, disables line stippling, and flushes the buffers.

Figure 7 shows a picture of the five rows of lines each with its own stipple pattern.

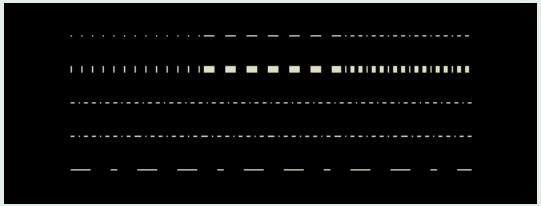

Figure 7: Five rows of lines, each with a different stipple pattern.

97a ⟨*Lines* `display` *and redisplay* 97a⟩≡

```
void display(void) {
 int i;
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glEnable(GL_LINE_STIPPLE);
    ⟨Lines first row, 3 lines, each with a different stipple 98a⟩
    ⟨Lines second row, 3 wide lines, each with different stipple 99a⟩
    ⟨Lines third row, 6 lines, with dash/dot/dash stipple 99b⟩
    ⟨Lines fourth row, 6 independent lines with same stipple 100a⟩
    ⟨Lines fifth row, 1 line, with dash/dot/dash stipple 100b⟩
    glDisable(GL_LINE_STIPPLE);
    glFlush();
}
```

8.3.1. Draw three lines each with a different stipple pattern

Stipple patterns are defined using

- `glLineStipple(GLint factor, GLushort pattern)` where `factor` determines how many times a bit in `pattern` is used.

In the examples below, a pattern of

- `0x0101` defines a short (16 bit) hexadecimal integer

  0000 0001 0000 0001

  that sets the stipple pattern to a dotted lines with 7 off pixels between each on pixel. if the factor were 2 there would be 14 off pixels between every 2 on pixels.

- `0x00ff` defines a dashed line with pixel pattern

  0000 0000 1111 1111.

- `0x1C47` defines a dash-dot-dash line with pixel pattern

  0001 1100 0100 0111.

98a ⟨*Lines first row, 3 lines, each with a different stipple* 98a⟩≡

```
glLineStipple(1, 0x0101);
drawOneLine(50.0, 125.0, 150.0, 125.0);
glLineStipple(1, 0x00FF);
drawOneLine(150.0, 125.0, 250.0, 125.0);
glLineStipple(1, 0x1C47);
drawOneLine(250.0, 125.0, 350.0, 125.0);
```

8.3.2. Draw three wide lines each with a different stipple pattern

This example repeats the above except the line width is set to 5 creating a wider line.

99a ⟨*Lines second row, 3 wide lines, each with different stipple* 99a⟩≡

```
glLineWidth(5.0);
glLineStipple(1, 0x0101);
drawOneLine(50.0, 100.0, 150.0, 100.0);
glLineStipple(1, 0x00FF);
drawOneLine(150.0, 100.0, 250.0, 100.0);
glLineStipple(1, 0x1C47);
drawOneLine(250.0, 100.0, 350.0, 100.0);
glLineWidth(1.0);
```

8.3.3. Draw six lines each with a dash/dot/dash stipple

99b ⟨*Lines third row, 6 lines, with dash/dot/dash stipple* 99b⟩≡

```
glLineStipple(1, 0x1C47);
glBegin(GL_LINE_STRIP);
for (i = 0; i < 7; i++) {
   glVertex2f(50.0 + ((GLfloat) i * 50.0), 75.0);
}
glEnd ();
```

8.3.4. Draw six lines each with a dash/dot/dash stipple

100a ⟨*Lines fourth row, 6 independent lines with same stipple* 100a⟩≡

```
for (i = 0; i < 6; i++) {
    drawOneLine(50.0 + ((GLfloat) i * 50.0), 50.0,
    50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
}
```

8.3.5. Draw one line with a stipple repeat factor of 5

100b ⟨*Lines fifth row, 1 line, with dash/dot/dash stipple* 100b⟩≡

```
glLineStipple(5, 0x1C47);
drawOneLine(50.0, 25.0, 350.0, 25.0);
```

8.4. Reshaping the window

Most of the initialization steps are moved to a `reshape` callback function that is registered with GLUT to be activated anytime a window move or resize event is detected.

The simplified function

- `gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)`

is used to set up a two dimensional orthographic parallel projection. This is equivalent to calling `glOrtho` with near = -1 and far = 1.

101a ⟨*Lines* `reshape` *the window callback* 101a⟩≡

```
void reshape (int w, int h) {
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
```

8.5. A Keyboard Event Callback function

The `keyboard` callback function is registered with GLUT to executes when a keyboard event is detected. If the *escape* key is pressed, the program exits with an `OK` status.

102a ⟨*Lines* `keyboard` *event callback* 102a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case ESC: {
         exit(OK);
         break;
      }
   }
}
```

8.6. A Main function

The differences between this `main` function and the `Hello, World main` functions are described. Consult the `Hello, World` program for additional instruction on what happens here.

Two new GLUT functions are introduced:

- `glutReshapeFunc(void (*func)(int width, int height))` registers a callback function that is executed when the window is moved or resized to a new width and height.
- `glutMouseFunc(void (*func)(int button, int state, int x, int y))` registers a callback function that is executed when a mouse event is detected. The left, middle, or right button is either up or down at window coordinates (x, y).

103a ⟨*Lines* `main` *function* 103a⟩≡

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(400, 150);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Line Drawings");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return OK;
}
```

8.7. Lines Source Code

The C program code is available at:

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/lines.c

9. Polygons

This program, from the Red book Woo et al. [1999], demonstrates geometric primitives and their attributes.

105a ⟨*polys* 105a⟩≡
⟨*Polys header files* 105b⟩
⟨*Polys* `display` *function* 106a⟩
⟨*Polys initialization function* 108a⟩
⟨*Polys* `reshape` *the window callback* 108b⟩
⟨*Polys* `keyboard` *event callback* 109a⟩
⟨*Polys* `main` *function* 110a⟩

9.1. Headers and routine variables

The GLUT library is used for window and I/O functions. Numeric constants are defined to be `OK` for a successful run, and `ESC` to signal that *escape* key has been pressed to terminate the program.

105b ⟨*Polys header files* 105b⟩≡

```
#include <GL/glut.h>
#define OK 0
#define ESC 27  /* ASCII for the escape key */
```

9.2. Displaying and Redisplaying

This `display` defines two stipple patterns to be used in filling rectangles.

- `glPolygonStipple(const GLubyte *mask)` sets the stipple mask (pattern) used in filling the polygon. Each polygon stipple pattern is a 32×32 bitmap. One stipple pattern is the outline of a *fly*, see the Red Book Woo et al. [1999], and the second alternates on/off to create a gray halftone.

- `glRect(x1, y1, x2, y2)` is introduced as a shorthand for

```
glBegin(GL_POLYGON);
  glVertex(x1, y1);
  glVertex(x2, y1);
  glVertex(x2, y2);
  glVertex(x1, y2);
glEnd();
```

106a ⟨*Polys* `display` *function* 106a⟩≡

```
void display(void)
{
   GLubyte fly[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
```

```
    0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
    0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
    0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
    0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
    0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
    0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};

GLubyte halftone[] = {
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
    0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55};

glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glRectf(25.0, 25.0, 125.0, 125.0);
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(fly);
glRectf(125.0, 25.0, 225.0, 125.0);
glPolygonStipple(halftone);
```

```
    glRectf(225.0, 25.0, 325.0, 125.0);
    glDisable(GL_POLYGON_STIPPLE);
    glFlush ();
}
```

9.3. An Initialization function

A simple initialization is still in use.

108a ⟨*Polys initialization function* 108a⟩≡

```
void init (void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
}
```

9.4. Reshaping the window

Most of the initialization steps are moved to a `reshape` callback function that is registered with GLUT to be activated anytime a window move or resize event is detected.

108b ⟨*Polys* `reshape` *the window callback* 108b⟩≡

```
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
```

9.5. A Keyboard Event Callback function

The `keyboard` callback function is registered with GLUT to executes when a keyboard event is detected. If the *escape* key is pressed, the program exits with an `OK` status.

109a ⟨*Polys* `keyboard` *event callback* 109a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case ESC: {
         exit(OK);
         break;
      }
   }
}
```

9.6. The Main Function

The only difference between this `main` and the `lines main` program is the title of the window.

110a ⟨*Polys* `main` *function* 110a⟩≡

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(350, 150);
   glutCreateWindow("Polygons");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc (keyboard);
   glutMainLoop();
   return OK;
}
```

9.7. Polys Source Code

The C program code is available at:

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/polys.c

10. Rendering a Lit Sphere

This program is from the Red book Woo et al. [1999]. It displays a sphere illuminated by a single light source. This is shown in Figure 8. It is in the initialization function where new

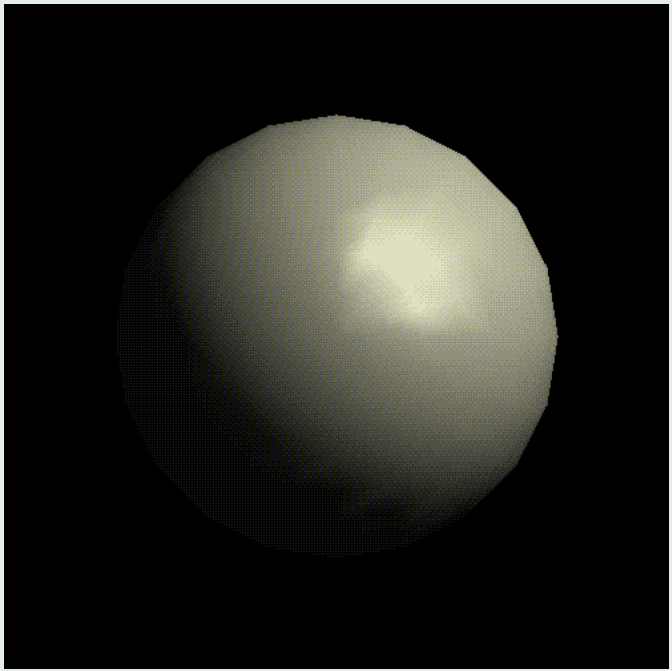

Figure 8: A lit sphere.

material on OpenGL is included. Here you will learn to set material and lighting properties that are then used in illuminating and rendering the sphere with a specular highlight.

112a ⟨*light* 112a⟩≡
⟨*Light header files* 112b⟩
⟨*Light initialization function* 114a⟩
⟨*Light* `display` *function* 117b⟩
⟨*Light* `reshape` *function* 118a⟩
⟨*Light* `keyboard` *function* 119a⟩
⟨*Light* `main` *function* 113a⟩

10.1. Header Inclusion

The GLUT header file is needed to define prototype functions and parameters. Global constants are defined as terminate (`ESC`) and return (`OK`) codes.

112b ⟨*Light header files* 112b⟩≡
```
#include <GL/glut.h>
#define OK 0
#define ESC 27
```

10.2. The `main` function

`main` performs the standard steps that have been explained in earlier programs. One thing that may be new is to initialize the window with depth buffering turned on to enable hidden surface elimination.

113a ⟨*Light* `main` *function* 113a⟩≡

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Rendering a Lit Sphere");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
}
```

Parameter Name	Default	Comment
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	material ambient color
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	material diffuse color
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	material specular color
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	material emissive color
GL_COLOR_INDICES	(0, 1, 1)	color in lookup table

Table 1: Parameters for Material Properties

10.3. Initialization for rendering

This initialization function sets material and light properties.

The function `glMaterial*(GLenum face, GLenum parameterName, TYPE parameter);` set material properties of the object to be rendered. The `face` can be either front, back, or both (`GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`), and determine the rendering of the named face(s). The `parameterName` can take on several values as shown in Table 1.

The light is also important in rendering. The function `glLight*(GLenum light, GLenum parameterName, TYPE parameter);` set light properties. There can be up to 8 `light` sources, named `GL_LIGHT0` through `GL_LIGHT7`. The `parameterName` can take on several values as shown in Table 2.

114a ⟨*Light initialization function* 114a⟩≡

```
void init(void)
{
  ⟨init local variables 115a⟩
  ⟨Set material properties 116a⟩
  ⟨Set the light properties 116b⟩
  ⟨Set the shading model 116c⟩
  ⟨Enable lighting and hidden surface removal 117a⟩
  glClearColor (0.0, 0.0, 0.0, 0.0);
```

Parameter Name	Default	Comment
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	light ambient intensity
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	light diffuse intensity
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	light specular intensity
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	light position
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight cutoff angle
GL_SPOT_CUTOFF	180	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant coefficient
GL_LINEAR_ATTENUATION	0.0	linear coefficient
GL_QUADRATIC_ATTENUATION	0.0	quadratic coefficient

Table 2: Parameters for Lights

```
}
```

Three arrays are used to set material and light properties.

115a ⟨init *local variables* 115a⟩≡

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[] = { 50.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
```

For this simple problem, we only have interest in the specular component of reflected light, which is set to white. The material Phong specular exponent controls the shininess of the object. It can be between 0.0 and 128.0. The higher the value the more the highlight is focused (small and bright).

116a ⟨*Set material properties* 116a⟩≡

```
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
```

Lights can be *directional* or *positional*. Directional light sources are *at infinity* which means that all rays from them are parallel. This increases the rendering efficiency. Positional light sources are a a local position in the scene, so rays from them change from one vertex to another.

The light in this sample code is *directional*, you can see this from the last component in `light_position` being 0.0. Explaining the mathematics behind this is beyond the scope of this presentation. The light rays are from direction (1.0, 1.0, 1.0).

116b ⟨*Set the light properties* 116b⟩≡

```
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Two shading models are supported by OpenGL: flat and smooth. Smooth shading is also called *Gouraud* shading, which uses bilinear color interpolation from vertices across the faces of polygons to make a faceted object look smooth.

116c ⟨*Set the shading model* 116c⟩≡

```
glShadeModel (GL_SMOOTH);
```

To cause OpenGL to calculate the lighting model, lighting must be enabled, and additionally, each light source must be enabled. Here only one light `GL_LIGHT0` has been used. Depth testing for hidden surface removal is also enabled.

117a ⟨*Enable lighting and hidden surface removal* 117a⟩≡

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);
```

10.4. The Display function

The display function clears the color buffer and depth buffer and draws a sphere with radius 1.0 with 20 meridians of longitude and 16 parallels of latitude.

117b ⟨*Light* `display` *function* 117b⟩≡

```
void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glutSolidSphere (1.0, 20, 16);
   glFlush ();
}
```

10.5. The Redisplay function

When the window is moved or reshaped, the viewport is set to match the size of the new window and new model-view and projection matrices are loaded to reflect this change. The model-view is just the identity, no translations, scales, or rotations are performed on the sphere.

An orthographic projection is defined with aspect ratio set equal to the aspect ratio of the viewport so that the sphere remains spherical.

118a ⟨*Light* `reshape` *function* 118a⟩≡

```
void reshape (int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
         1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h,
         1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

10.6. The Keyboard function

The `keyboard` callback function is registered with GLUT to execute when a keyboard event is detected. If the *escape* key is pressed, the program exits with an `OK` status.

119a ⟨*Light* `keyboard` *function* 119a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case ESC: {
         exit(OK);
         break;
      }
   }
}
```

10.7. Lit Sphere Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/light.c

10.8. Lighting Tutorial

Additional information on lighting in OpenGL is available through course notes for CSE 3280 Computer Graphics Programming.

11. Moving lights

This program is from the Red book Woo et al. [1999]. It draws a torus illuminated by a light. A mouse interaction allows the user to rotate the light around the x-axis.

This is shown in Figure 9. It is in the initialization function where new material on

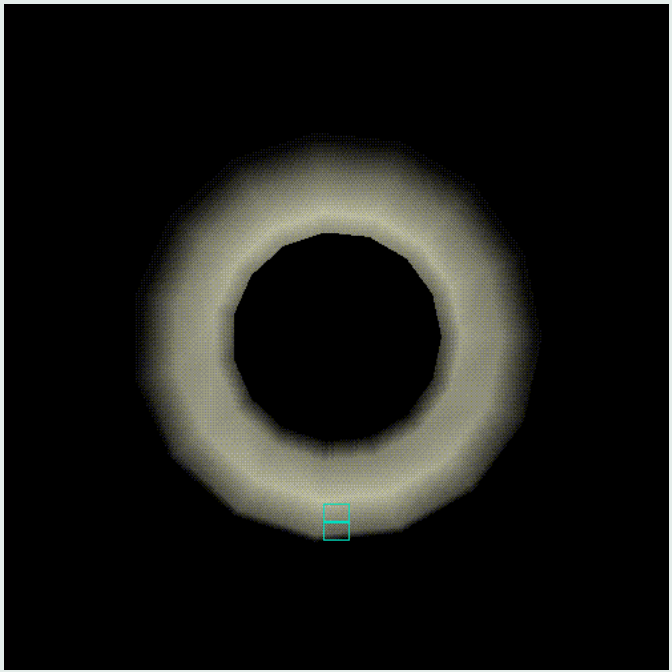

Figure 9: Moving lights.

OpenGL is included. Here you will learn to set material and lighting properties that are then used in illuminating and rendering the sphere with a specular highlight.

121a ⟨*movelight* 121a⟩≡
⟨*Moving light header files* 121b⟩
⟨*Moving light initialization function* 123a⟩
⟨*Moving light* `display` *function* 124a⟩
⟨*Moving light* `reshape` *function* 125a⟩
⟨*Moving light* `mouse` *function* 126a⟩
⟨*Moving light* `keyboard` *function* 127a⟩
⟨*Moving light* `main` *function* 122a⟩

11.1. Header Inclusion

The GLUT header file is needed to define prototype functions and parameters. Global constants are defined as terminate (`ESC`) and return (`OK`) codes. A global variable (`spin`) is used to rotate the torus and light source.

121b ⟨*Moving light header files* 121b⟩≡

```
#include <GL/glut.h>
#define OK 0
#define ESC 27

static int spin = 0;
```

11.2. The `main` function

`main` performs the standard steps that have been explained in earlier programs.

122a ⟨*Moving light* `main` *function* 122a⟩≡

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow ("Moving Lights");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
}
```

11.3. Initialization for rendering

Not much is done on initialization. Defaults are used.

123a ⟨*Moving light initialization function* 123a⟩≡

```
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
```

11.4. The Display function

/* Here is where the light position is reset after the modeling * transformation (glRotated) is called. This places the * light at a new position in world coordinates. The cube * represents the position of the light. */

124a ⟨*Moving light* `display` *function* 124a⟩≡

```
void display(void)
{
   GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);

   glTranslated (0.0, 0.0, 1.5);
   glDisable (GL_LIGHTING);
   glColor3f (0.0, 1.0, 1.0);
   glutWireCube (0.1);
   glEnable (GL_LIGHTING);
   glPopMatrix ();

   glutSolidTorus (0.275, 0.85, 8, 15);
   glPopMatrix ();
   glFlush ();
}
```

11.5. The Redisplay function

When the window is moved or reshaped, the viewport is set to match the size of the new window and new model-view and projection matrices are loaded to reflect this change. The model-view is just the identity, no translations, scales, or rotations are performed on the sphere.

An orthographic projection is defined with aspect ratio set equal to the aspect ratio of the viewport so that the sphere remains spherical.

125a ⟨*Moving light* `reshape` *function* 125a⟩≡

```
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

11.6. The Mouse function

When the left button is held down, the spin is increased by 30 degrees modulo 360.

126a ⟨*Moving light* `mouse` *function* 126a⟩≡

```
void mouse(int button, int state, int x, int y) {
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN) {
            spin = (spin + 30) % 360;
            glutPostRedisplay();
         }
         break;
      default:
         break;
   }
}
```

11.7. The Keyboard function

The `keyboard` callback function is registered with GLUT to execute when a keyboard event is detected. If the *escape* key is pressed, the program exits with an `OK` status.

127a ⟨*Moving light* `keyboard` *function* 127a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case ESC: {
         exit(OK);
         break;
      }
   }
}
```

11.8. Moving Lights Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/movelight.c

11.9. Lighting Tutorial

Additional information on lighting in OpenGL is available through course notes for CSE 3280 Computer Graphics Programming.

12. A Simple Checkerboard Texture

This program texture maps a checkerboard image onto two rectangles as shown in Figure ??.

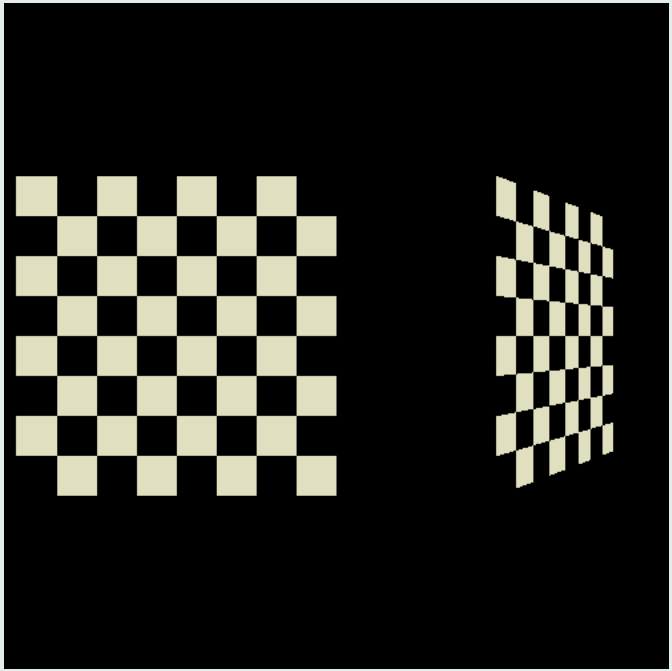

Figure 10: Two checkerboard textured rectangles.

128a ⟨*checker* 128a⟩≡
⟨*Checker header files* 129a⟩
⟨*Checker global variables* 129b⟩
⟨*Make the texture* 130a⟩
⟨*Checker* `display` *function* 134a⟩
⟨*Checker* `reshape` *function* 135a⟩
⟨*Checker initialization function* 132a⟩
⟨*Checker* `main` *function* 136a⟩

12.1. Checker header inclusion

We continue to use GLUT.

129a ⟨*Checker header files* 129a⟩≡

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

Here are some global constants and an array to hold red, green, and blue values for a 8×8 checker board where each checker square is an 8×8 pixel array.

129b ⟨*Checker global variables* 129b⟩≡

```
#define OK 0
#define checkImageWidth 64
#define checkImageHeight 64

GLubyte checkImage[checkImageWidth][checkImageHeight][3];
```

12.2. Defining a texture

Often one would use an existing image as a texture, but textures can be defined procedurally. What happens here is that integers i, $j = 0, 1, \ldots, 63$ are bitwise ANDed with hexadecimal $0x8 = 1000$. This masks off (sets to zero) all but the fourth bit of i and j. So, if the fourth bit of i or j is 1 the result will be 1, otherwise 0. Specifically, when $16k \leq i \leq 16k+7$ for $k = 0$, 1, 2, 3 the expression `i&0x8` is 0 and when $16k+8 \leq i \leq 16k+15$ for $k = 0$, 1, 2, 3 the expression `i&0x8` is 1

But then the EXCLUSIVE OR (`^`) of `i&0x8` and `j&0x8` is taken so when they are both 0 or 1 the final result is 0 and it is 1 otherwise.

For example, when $0 \leq i, j \leq 7$ each of `i&0x8` and `j&0x8` 0 so their exclusive or is 0 and the color is set to black. Then when $0 \leq i, \leq 7$ and $8 \leq j, \leq 15$ `i&0x8=0` and `j&0x8=1` so their exclusive or is 1 and the color is set to white. And so on.

130a ⟨*Make the texture* 130a⟩≡

```
void makeCheckImage(void) {
  int i, j, r, c;

  for (i = 0; i < checkImageWidth; i++) {
    for (j = 0; j < checkImageHeight; j++) {
      c = ((((i&0x8)==0)^((j&0x8))==0))*255;
          checkImage[i][j][0] = (GLubyte) c;
          checkImage[i][j][1] = (GLubyte) c;
          checkImage[i][j][2] = (GLubyte) c;
    }
  }
}
```

12.3. Checker initialization

After clearing the color buffer with `glClear()`, depth buffering is enabled. Then the comparison function for depth buffering is set using `glDepthFunc()`. The default test is `GL_LESS`, so the call made here is superfluous, but it illustrates how the depth function can be changed. In this case, when an incoming fragment has its z value (the distance from the fragment to the viewpoint) less than the value already stored in the depth buffer, the test is passed, and the incoming fragment is stored in the color buffer with its z stored in the depth buffer. Other functions can be set are: `GL_NEVER`, `GL_ALWAYS`, `GL_EQUAL`, `GL_LEQUAL`, `GL_GEQUAL`, `GL_GREATER`, and `GL_NOTEQUAL`.

Next the `makeCheckImage()` function is called to fill out the texture array.

The call to `glPixelStore*()` is rather complex and I will not attempt a full explanation here. See the class notes on raster primitives for additional information. Suffice it to say that `glPixelStore*()` set the mode for texture storage, in this case to `GL_UNPACK-ALIGNMENT` so that pixel data starts on byte alignments (the parameter 1 indicates byte).

The call to `glTexImage2D()` specifies a 2D texture image, where the parameters are in order:

- `GLenum t` is the target either: `GL_TEXTURE_2D` or `GL_PROXY_TEXTURE_2D` (the proxy is used to query if the graphics system can accommodate a texture with the other parameters).
- `GLint l` specifies a level-of-detail for mip-mapping. Level 0 is the base texture.
- `GLint f` specifies the internal format of the texture. It must be 1, 2, 3 or 4, or one of about 40 supported symbolic constants. For example, a format `f = GL_R3_G3_B2` means three bits should be used for red and green and two bits for blue.
- `GLsizei w` is the width of the texture image. `w` must be of the form $2^n + 2b$ where `b` is the value of the border.

- `GLsizei h` is the height of the texture image. `h` must be of the form $2^m + 2b$ where `b` is the value of the border.
- `GLint b` is the width of the texture border and must be either 0 or 1.
- `GLenum m` and `GLenum t` describe the format and type of the texture data. The format `m` can be one of `GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`. The type `t` can be one of `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, and `GL_FLOAT`
- `const GLvoid *p` is a pointer to the start of the texture data, in this case the location of the array `checkImage`.

Then there are calls to `glTexParameter*()` that set various texture parameters. The `GL_TEXTURE_WRAP_S` and `T` parameter control how the s and t texture coordinates are handled when outside of their default range from 0 to 1. This program clamps the texture coordinates to the range [0.0, 1.0]. Other values for the parameter include:

- `GL_REPEAT` which causes the integer part of s to be discarded; a cyclic wrap.
- `GL_CLAMP_TO_EDGE` which causes the value s to be clamped to the range $[1/2N, 1 - 1/2N]$ where N is the size of the texture in the s direction.

Lastly, except for enabling texture mapping and setting the shading model, there is a call to `glTexEnv*()`, which sets texture environment parameters. In this case, the environment is set to `GL_DECAL`, which with the internal format set to `GL_RGB` (see above) the color assigned to the quadrangles is the color of the texture.

132a ⟨*Checker initialization function* 132a⟩≡

```
void myinit(void) {
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glEnable(GL_DEPTH_TEST);
```

```
  glDepthFunc(GL_LESS);

  makeCheckImage();
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glTexImage2D(GL_TEXTURE_2D, 0, 3, checkImageWidth,
      checkImageHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
      &checkImage[0][0][0]);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
  glEnable(GL_TEXTURE_2D);
  glShadeModel(GL_FLAT);
}
```

12.4. Checker `display` function

The `display` function draws two quadrangles, each time associating the corners of the checkerboard image to the corners of the quadrangles. The first quadrangle is viewed flat as a rectangle, but the second has been explicitly rotated by 45°.

134a ⟨*Checker* `display` *function* 134a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
  glEnd();
  glFlush();
}
```

12.5. Checker `reshape` function

The projection is reset for perspective viewing of the object. This is accomplished with `gluPerspective()` that sets a field of view angle (60°) in the y direction, an aspect ratio (w/h) and near (1.0) and far (30.0) clipping planes from the viewer.

135a ⟨*Checker* `reshape` *function* 135a⟩≡

```
void myReshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 30.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0, 0.0, -3.6);
}
```

12.6. Checker `main` function

The `main` function contains little that is new.

136a ⟨*Checker* `main` *function* 136a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH;
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow ("Two Checkered Rectangles");
  myinit();
  glutReshapeFunc(myReshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return OK;
}
```

12.7. Checkered Texture Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/checker.c

13. A Simple Display List

Here's a simple program from the Red book Woo et al. [1999] that utilizes display lists.

137a ⟨*list* 137a⟩≡
⟨*List header files* 137b⟩
⟨*List initialization function* 138a⟩
⟨*List* `drawline` *function* 139a⟩
⟨*List* `display` *function* 139b⟩
⟨*List* `reshape` *function* 140a⟩
⟨*List* `main` *function* 141a⟩

137b ⟨*List header files* 137b⟩≡

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

GLuint listName = 1;
```

13.1. Initialization function

This initialization function creates creates a new display list, named by `listName`. The list is created between `glNewList()`, `glEndList()` delimiters, and only *compiled*. This is, it is not immediately displayed (*executed*). The list contains one red trianble and a translation by 1.5 in the x direction.

Since there is only one display list in the program it is probably okay to avoid using `glGenLists()` to generate the starting index of a range of previously unused list indices.

138a ⟨*List initialization function* 138a⟩≡

```
void init (void) {
  glNewList (listName, GL_COMPILE);
    glColor3f (1.0, 0.0, 0.0);
    glBegin (GL_TRIANGLES);
      glVertex2f (0.0, 0.0);
      glVertex2f (1.0, 0.0);
      glVertex2f (0.0, 1.0);
    glEnd ();
    glTranslatef (1.5, 0.0, 0.0);
  glEndList ();
  glShadeModel (GL_FLAT);
}
```

13.2. Draw a vertical line

139a ⟨*List* `drawline` *function* 139a⟩≡

```
void drawLine (void) {
  glBegin(GL_LINES);
  glVertex2f(0.0, 0.5);
  glVertex2f(15.0, 0.5);
  glEnd();
}
```

13.3. The `display` function

The `display` function is where the action is. The display list is executed 10 times with the `glCallList()` command. Each time the red triangle stored in the display list is and the location for the next draw is "scooted" over 1.5 unit by the translate stored in the display list. This is followed by an immediate mode `drawline` command.

139b ⟨*List* `display` *function* 139b⟩≡

```
void display(void) {
  GLuint i;

  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (0.0, 1.0, 0.0);
  for (i = 0; i < 10; i++) glCallList(listName);
  drawLine ();
  glFlush ();
}
```

13.4. The reshape function

Nothing new is added here. Depending of the new width `w` and height `h` of the viewport, a new orthographic projection is set to maintain the aspect ratio.

140a ⟨*List* `reshape` *function* 140a⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h) {
    gluOrtho2D (0.0, 2.0, -0.5 * (GLfloat) h/(GLfloat) w,
                1.5 * (GLfloat) h/(GLfloat) w);
  }
  else {
    gluOrtho2D (0.0, 2.0 * (GLfloat) w/(GLfloat) h, -0.5, 1.5);
  }
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

13.5. The `main` function

Nothing new here.

141a ⟨*List* `main` *function* 141a⟩≡

```
int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(650, 50);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Display Lists");
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
}
```

13.6. Display List Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/list.c

14. A Stencil Buffer Program

Here's a simple program from the Red book Woo et al. [1999] that shows how to use the stencil buffer. The program draws two rotated tori in a window. A diamond in the center of the window masks out part of the scene. Within this mask, a different model (a sphere) is drawn in a different color. The result of the program is shown in Figure 11.

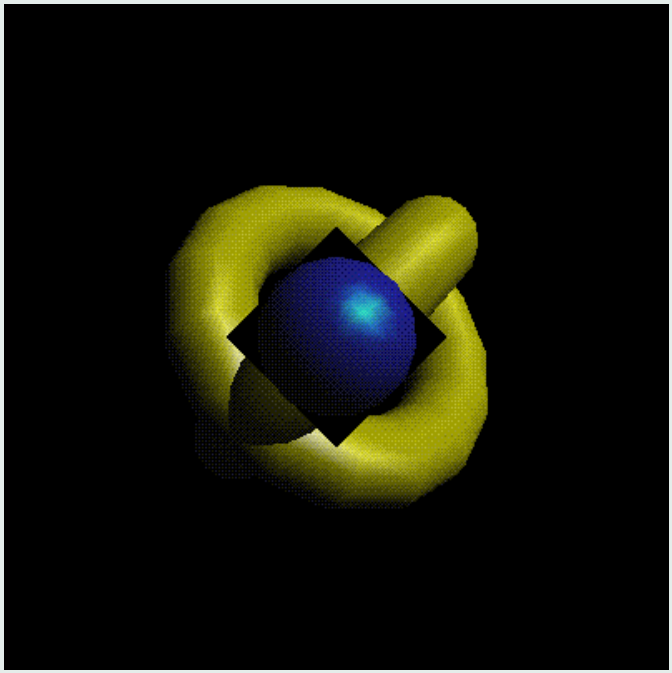

Figure 11: Tori with stenciled diamond containing sphere.

Stenciling is used to achieve special effects such as adding decals, outlining portions of a scene, and in some constructive solid geometry rendering.

143a ⟨*stencil* 143a⟩≡
⟨*Stencil header files* 143b⟩
⟨*Stencil initialization function* 144a⟩
⟨*Stencil* `display` *function* 150a⟩
⟨*Stencil* `reshape` *function* 149a⟩
⟨*Stencil* `main` *function* 151a⟩

14.1. Stencil header files

Two global constants are defined `YELLOWMAT` and `BLUEMAT` and used as indices for display lists. Each list holds state information that sets material properties for objects to be displayed.

143b ⟨*Stencil header files* 143b⟩≡

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

#define OK 0
#define YELLOWMAT 1
#define BLUEMAT   2
```

14.2. Stencil initialization

Five 4D arrays are assigned: 2 define yellow diffuse and specular material colors; 2 define blue diffuse and specular material colors; and one defines the position of a light source.

Two display list are created between `glNewList()`, `glEndList()` delimiters. One list defines yellow material properties and the other defines blue material properties with calls to `glMaterial*()`.

Next lighting is set up. Light zero is positioned with `glLight*()` and activated, and lighting calculations based on the illumination equation is enabled.

And then depth buffering is enabled. Prior to doing this, the comparison function is set using `glDepthFunc()`. The default test is `GL_LESS`, so the call made here is superfluous, but it illustrates how the depth function can be changed. In this case, when an incoming fragment has its z value (the distance from the fragment to the viewpoint) less than the value already stored in the depth buffer, the test is passed, and the incoming fragment is stored in the color buffer with its z stored in the depth buffer. Other functions can be set are: `GL_NEVER`, `GL_ALWAYS`, `GL_EQUAL`, `GL_LEQUAL`, `GL_GEQUAL`, `GL_GREATER`, and `GL_NOTEQUAL`.

The last step in `myinit()` is to set the value to be used when the the stencil buffer is cleared. The function `glClearStencil()` does this so that when `glClear()` is called with `GL_STENCIL_BUFFER_BIT` the stencil buffer will be set to all zeros.

144a ⟨*Stencil initialization function* 144a⟩≡

```
void myinit (void) {
  GLfloat yellow_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
  GLfloat yellow_specular[] = { 1.0, 1.0, 1.0, 1.0 };

  GLfloat blue_diffuse[] = { 0.1, 0.1, 0.7, 1.0 };
  GLfloat blue_specular[] = { 0.1, 1.0, 1.0, 1.0 };

  GLfloat position_one[] = { 1.0, 1.0, 1.0, 0.0 };

  glNewList(YELLOWMAT, GL_COMPILE);
```

```
    glMaterialfv(GL_FRONT, GL_DIFFUSE, yellow_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, yellow_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
  glEndList();

  glNewList(BLUEMAT, GL_COMPILE);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, blue_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, blue_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 45.0);
  glEndList();

  glLightfv(GL_LIGHT0, GL_POSITION, position_one);

  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);

  glClearStencil(0x0);
  glEnable(GL_STENCIL_TEST);
}
```

14.3. Stencil header files

The `reshape` function does its normal task of redefining the coordinate system when the window is changed in width and height, in particular, it sets up an orthographic projection to project the stencil onto the viewport.

146a ⟨*Parallel project stencil onto viewport* 146a⟩≡

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h) {
  gluOrtho2D(-3.0, 3.0, -3.0*(GLfloat)h/(GLfloat)w, 3.0*(GLfloat)h/(GLfloat)w);
}
else {
  gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h, 3.0*(GLfloat)w/(GLfloat)h, -3.0, 3.0);
}
```

Stencil Function	Pass Condition
GL_NEVER	Always fails
GL_LESS	(ref & mask) $<$ (stencil & mask)
GL_LEQUAL	(ref & mask) $\leq$ (stencil & mask)
GL_GREATER	(ref & mask) $>$ (stencil & mask)
GL_GEQUAL	(ref & mask) $\geq$ (stencil & mask)
GL_EQUAL	(ref & mask) $=$ (stencil & mask)
GL_NOTEQUAL	(ref & mask) $\neq$ (stencil & mask)
GL_ALWAYS	Always passes

Table 3: Stencil function and tests.

But now `reshape` performs a second task of defining and redrawing the stencil shape.

The stencil area is a diamond shaped region, defined as a quadrangle between `glBegin()` `glEnd()` delimiters, with corners out one unit on each of the positive and negative x and y axes.

A stencil buffer has some number of bit planes, call this value n. The stencil buffer is cleared with `glClear(GL_STENCIL_BUFFER_BIT)`. The function `glStencilFunc(GLenum func, GLint ref, GLuint mask)` sets the stencil function, the reference value, and the stencil mask. Both the reference value and the stencil value are integers in the range 0 to $2^n - 1$. Depending on the stencil function, the reference value, the mask value and the stencil value, the stencil test is passed (or not) and the pixel sent to the next rendering stage (or not). Table 3 shows how the stencil function interacts with these valued to determine if the test is passed or not. In the code below, the stencil test will always pass.

The `glStencilOp(GLenum fail, GLenum zfail, GLenum zpass)` function specifies the action to take dependent on the stencil test and depth test. In the code below the action is `GL_REPLACE` which causes the stencil buffer to be set to the reference value (hexadecimal 1) with out regard to whether the stencil or depth tests passed or failed. That is the diamond is drawn in all black an it becomes the stencil.

I won't try to explain all of the possibilities for `glStencilOp(GLenum fail, GLenum zfail, GLenum zpass)` here but refer you to the manual page for `glStencilOp()`, the Red Book Woo et al. [1999] and the Blue Book Shreiner [1999].

148a ⟨*Define the stencil and how it acts* 148a⟩≡

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

glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0x1, 0x1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glBegin(GL_QUADS);
  glVertex3f (-1.0, 0.0, 0.0);
  glVertex3f (0.0, 1.0, 0.0);
  glVertex3f (1.0, 0.0, 0.0);
  glVertex3f (0.0, -1.0, 0.0);
glEnd();
```

We reset the projection for perspective viewing of the object. This is accomplished here with `gluPerspective()` that sets a field of view angle ($45°$) in the y direction, an aspect ratio (w/h) and near (3.0) and far (7.0) clipping planes from the viewer.

148b ⟨*Perspective projection of the objects* 148b⟩≡

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 7.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -5.0);
```

The `myReshape` function is now just a few lines.

149a ⟨*Stencil* `reshape` *function* 149a⟩≡

```
void myReshape(int w, int h) {
  glViewport(0, 0, w, h);
  ⟨Parallel project stencil onto viewport 146a⟩
  ⟨Define the stencil and how it acts 148a⟩
  ⟨Perspective projection of the objects 148b⟩
}
```

14.4. Stencil `display` function

The `display` function draws a sphere in a diamond-shaped stencil in the middle of a window with 2 tori.

Specifically, after clearing the color and depth buffer, a blue sphere is drawn where the stencil is 1 and yellow tori are drawn where the stencil is not 1.

That is, the stencil function is set to test equality of $0x1 \& 0x1 = 0x1$ (the reference value bitwise ANDed with the mask value) with the stencil value (1 in the diamond, 0 outside) bitwise ANDed with the mask. Since the sphere has radius 0.5 all of it will be drawn.

Then, the stencil function is set to test for inequality, so now the tori will be drawn outside the stencil.

150a ⟨*Stencil* `display` *function* 150a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


  glStencilFunc(GL_EQUAL, 0x1, 0x1);
  glCallList(BLUEMAT);
  glutSolidSphere(0.5, 15, 15);

  glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);
  glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
  glPushMatrix();
    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 0.0, 1.0, 0.0);
    glCallList (YELLOWMAT);
    glutSolidTorus (0.275, 0.85, 15, 15);
    glPushMatrix();
      glRotatef (90.0, 1.0, 0.0, 0.0);
      glutSolidTorus (0.275, 0.85,15, 15);
```

```
    glPopMatrix();
  glPopMatrix();
}
```

14.5. Stencil `main` function

The `main` function contains little that is new other than to inform GLUT that the stencil buffer will be used.

151a ⟨*Stencil* `main` *function* 151a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH | GLUT_STENCIL);
  glutInitWindowSize(400, 400);
  glutInitWindowPosition(100, 100);
  glutCreateWindow ("Stencil Example");
  myinit();
  glutReshapeFunc(myReshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return OK;
}
```

14.6. Stencil Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/stencil.c

15. Alpha Blending

This programs draws two overlapping triangles (a left one and a right one) to demonstrate basic blending concepts. The order in which the triangles is drawn is controlled by keyboard toggling. That is, by default the left triangle is drawn first, but typing 't' or 'T' *toggles* the order of drawing.

152a ⟨*alpha* 152a⟩≡
⟨*Alpha header files* 152b⟩
⟨*Alpha triangle shape functions* 153a⟩
⟨*Alpha* `init` *function* 155a⟩
⟨*Alpha* `display` *function* 156a⟩
⟨*Alpha* `reshape` *function* 157a⟩
⟨*Alpha* `keyboard` *function* 158a⟩
⟨*Alpha* `main` *function* 159a⟩

15.1. Alpha header files

The `glut.h` header is included, `OK` is defined and the ASCII value of the escape key is set to the symbolic constant `ESC`.

152b ⟨*Alpha header files* 152b⟩≡

```
#include <GL/glut.h>
#define OK 0
#define ESC 27
```

15.2. Left and right side triangle

Boolean variable `leftFirst` is used to control the order in which yellow left triangle and cyan right triangles are drawn.

153a ⟨*Alpha triangle shape functions* 153a⟩≡

```
GLboolean leftFirst = GL_TRUE;

static void drawLeftTriangle(void) {
  glBegin(GL_TRIANGLES);
    glColor4f(1.0, 1.0, 0.0, 0.75);
    glVertex3f(0.1, 0.9, 0.0);
    glVertex3f(0.1, 0.1, 0.0);
    glVertex3f(0.7, 0.5, 0.0);
  glEnd();
}

static void drawRightTriangle(void) {
  glBegin(GL_TRIANGLES);
    glColor4f(0.0, 1.0, 1.0, 0.75);
    glVertex3f(0.9, 0.9, 0.0);
    glVertex3f(0.3, 0.5, 0.0);
    glVertex3f(0.9, 0.1, 0.0);
  glEnd();
}
```

15.3. Alpha initialization

Upon initialization,

- blending is enabled
- *source blending factors* are set to (A_s, A_s, A_s, A_s).
- *destination blending factors* are set to $(1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s)$.
- Flat shading is enabled.
- And transparent black is set as the color for clearing the color buffer.

So what happens? When the yellow left triangle is drawn first it is blended as the source, with alpha factor $A_s = 0.75$, with the destination transparent background black, with one minus alpha factor $1 - A_s = 0.25$, generating pixel colors

$$\begin{aligned} R &= 0.75 \times 1.0 + 0.25 \times 0.0 = 0.75 \\ G &= 0.75 \times 1.0 + 0.25 \times 0.0 = 0.75 \\ B &= 0.75 \times 0.0 + 0.25 \times 0.0 = 0.00 \\ A &= 0.75 \times 0.75 + 0.25 \times 0.0 = 0.5625 \end{aligned}$$

over the triangle. That is a yellow triangle is rendered with some of the black background showing through.

Next, when the right cyan triangle is drawn, a similar calculation occurs over the background, but in the overlapping region the color generated is a blend of the cyan source with the yellow-black destination. The calculation is

$$\begin{aligned} R &= 0.75 \times 0.0 + 0.25 \times 0.75 = 0.1875 \\ G &= 0.75 \times 1.0 + 0.25 \times 0.75 = 0.9375 \end{aligned}$$

$$\begin{aligned} B &= 0.75 \times 1.0 + 0.25 \times 0.0 = 0.75 \\ A &= 0.75 \times 0.75 + 0.25 \times 0.5625 = 0.703125 \end{aligned}$$

155a ⟨*Alpha* `init` *function* 155a⟩≡

```
void init(void) {
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glShadeModel(GL_FLAT);
  glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

15.4. Alpha `display` function

The `display` function either draws a left side triangle and then an overlapping right side triangle or vice versa, dependent upon the value of `leftFirst`.

156a ⟨*Alpha* `display` *function* 156a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT);

  if (leftFirst) {
    drawLeftTriangle();
    drawRightTriangle();
  }
  else {
    drawRightTriangle();
    drawLeftTriangle();
  }
  glFlush();
}
```

15.5. Alpha reshape function

The **reshape** function does its normal tasks of resizing the projection to maintain the aspect ratio of the new viewport.

157a ⟨*Alpha* **reshape** *function* 157a⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
      gluOrtho2D(0.0, 1.0, 0.0, 1.0*(GLfloat)h/(GLfloat)w);
  else
      gluOrtho2D(0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0, 1.0);
}
```

15.6. Alpha keyboard function

158a ⟨*Alpha* keyboard *function* 158a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
  switch(key) {
    case 't':
    case 'T':
      leftFirst = !leftFirst;
      glutPostRedisplay();
      break;
    case ESC:
      exit(0);
      break;
    default:
      break;
  }
}
```

15.7. Alpha `main` function

The `main` function does the usual stuff: initialize GLUT, create a window of a given size at some position, perform any application initialization (set up blending), and register the `display`, `reshape` and `keyboard` callback functions prior to starting the graphics rendering loop.

159a ⟨*Alpha* `main` *function* 159a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(200, 200);
  glutCreateWindow("Alpha Blending");
  init();
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(display);
  glutMainLoop();
  return OK;
}
```

15.8. Alpha Source Code

The C program code is available at

http://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/alpha.c

16. Using the Depth Buffer in Alpha Blending

Data stored in the depth buffer can be used to blend transparent (translucent) objects with opaque objects. This program demonstrates how to intermix opaque and alpha blended polygons in the same scene, by using

`glDepthMask(GLboolean flag)`.

The depth buffer tracks the distance from the viewpoint and the closest object, rendered so far, occupying a given pixel. A simplified version of the depth-buffer code is:

160a ⟨*depth-buffer algorithm* 160a⟩≡

```
public void zbuffer() {
  for all polygons P {
    for all pixels (x, y) covered by P {
      if (z < zBuffer[x,y]) {
        colorBuffer[x,y] = color of P at (x, y);
        zBuffer[x,y] = z;
      }
    }
  }
}
```

When an object is rendered and found to occupy a pixel it's drawn only if it's closer to the viewpoint than whatever was there before. If it is closer the depth buffer is updated with this new closer value. Objects that are obscured (farther away) are not drawn and thus not used in blending; even if they are obscured by translucent objects.

To render both opaque and translucent objects correctly, draw all opaque objects first, then all translucent objects. And enable depth buffering when rendering opaque objects, but disable depth buffering, or better, make the depth buffer read-only, when rendering translucent objects.

When the opaque objects are rendered with depth buffering enabled, hidden objects are not drawn to the color buffer, and after all the opaque objects have been processed, the closest depth values are stored in the depth buffer.

Then, by making the depth buffer read-only, when translucent objects are processed, their depth values are still compared to determine if they are visible (closer than the opaque background) or not, and if they are their color can be blended with that of the opaque object.

Read/write of the depth buffer is controlled using

```
glDepthMask(GLboolean flag)
```

where the flag is either `GL_TRUE` (allow read/writes of the depth buffer) or `GL_FALSE` (read-only depth buffer).

To illustrate these concepts, this program will draw an translucent cube and a opaque sphere. Initially, the cube is behind the sphere, but typing 'a' or 'A' starts an animation sequence in which the cube moves forward through the sphere.

The structure of the program follows the familiar outline:

161a ⟨*alpha3D* 161a⟩≡
- ⟨*Alpha3D header files* 162a⟩
- ⟨*Alpha3D* `init` *function* 163a⟩
- ⟨*Alpha3D* `animate` *function* 164a⟩
- ⟨*Alpha3D* `keyboard` *function* 165a⟩
- ⟨*Alpha3D* `display` *function* 166a⟩

⟨*Alpha3D* `reshape` *function* 167a⟩

⟨*Alpha3D* `main` *function* 168a⟩

16.1. Alpha3D header files

There are several global constants and variable used in the code.

- `MAXZ` sets the initial depth of the translucent cube.
- `MINZ` sets the initial depth of the opaque cube.
- `ZINC` is added to `MINZ` and subtracted from `MAXZ` to move the cube forward and the sphere back. Since my computer does not support double buffering this is *jerky* on my machine, so I've set `ZINC=16` to make the move in one step. The original source codes had `ZINC=0.4`.

162a ⟨*Alpha3D header files* 162a⟩≡

```
#include <GL/glut.h>
#include <stdlib.h>
#define OK 0
#define ESC 27
#define MAXZ 8.0
#define MINZ -8.0
#define ZINC 16.0
static float solidZ = MAXZ;
static float transparentZ = MINZ;
static GLuint sphereList, cubeList;
```

16.2. Alpha3D initialization

A translucent specular white and Phong shininess exponent are defined for material properties of both the sphere and cube. A position $(0.5, 0.5, 1.0, 0.0)$ for light 0 is defined. (Recall setting the homogeneous coordinate to 0.0 makes the light rays parallel.)

Lighting calculations, `LIGHT0`, and depth buffering are enabled. And two display lists are defined: One to create a solid sphere and another to create a solid cube.

163a ⟨*Alpha3D* `init` *function* 163a⟩≡

```
void init(void) {
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 0.15 };
  GLfloat mat_shininess[] = { 100.0 };
  GLfloat position[] = { 0.5, 0.5, 1.0, 0.0 };

  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glLightfv(GL_LIGHT0, GL_POSITION, position);

  glEnable (GL_LIGHTING);
  glEnable (GL_LIGHT0);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);

  sphereList = glGenLists(1);
  glNewList(sphereList, GL_COMPILE);
    glutSolidSphere(0.4, 16, 16);
  glEndList();

  cubeList = glGenLists(1);
  glNewList(cubeList, GL_COMPILE);
    glutSolidCube(0.6);
  glEndList();
```

```
}
```

16.3. Alpha3D `animate` function

Here's the `animate` function that is used as an *idle* function. When `solidZ` has been decremented to `MINZ` or below or `transparentZ` has been incremented to `MAXZ` or above the animation stops. The scene can by reset by typing 'r' or 'R'.

164a ⟨*Alpha3D* `animate` *function* 164a⟩≡

```
void animate(void) {
  if (solidZ <= MINZ || transparentZ >= MAXZ) { glutIdleFunc(NULL); }
  else {
    solidZ -= ZINC;
    transparentZ += ZINC;
    glutPostRedisplay();
  }
}
```

16.4. Alpha3D keyboard function

Typing 'a' or 'A' starts an animation sequence where a translucent cube moves through an opaque sphere. Typing 'r' or 'R' resets the scene.

165a ⟨*Alpha3D* keyboard *function* 165a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
  switch(key) {
    case 'a':
    case 'A':
      solidZ = MAXZ;
      transparentZ = MINZ;
      glutIdleFunc(animate);
      break;
    case 'r':
    case 'R':
      solidZ = MAXZ;
      transparentZ = MINZ;
      glutPostRedisplay();
      break;
    case ESC:
      exit(0);
      break;
    default:
      break;
  }
}
```

16.5. Alpha3D `display` function

The sphere is drawn as an opaque object: Its diffuse color is yellowish with an alpha component of 1.0, and it emits a black light with alpha 1.0.

The cube is drawn as a translucent object: Its diffuse color is cyanish with an alpha component of 0.6, and it emits a dark cyan light with alpha 0.6.

The opaque sphere is drawn first with depth buffer already enabled by the `init()` routine. Before the translucent cube is drawn, blending is enabled and the depth buffer is set to read-only. Afterwards, the depth buffer is reset to read/write and blending is disabled.

166a ⟨*Alpha3D* `display` *function* 166a⟩≡

```
void display(void) {
  GLfloat mat_solid[] = { 0.75, 0.75, 0.0, 1.0 };
  GLfloat mat_zero[] = { 0.0, 0.0, 0.0, 1.0 };
  GLfloat mat_transparent[] = { 0.0, 0.8, 0.8, 0.6 };
  GLfloat mat_emission[] = { 0.0, 0.3, 0.3, 0.6 };

  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glPushMatrix ();
    glTranslatef (-0.15, -0.15, solidZ);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_zero);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_solid);
    glCallList(sphereList);
  glPopMatrix ();

  glPushMatrix();
    glTranslatef(0.15, 0.15, transparentZ);
    glRotatef(15.0, 1.0, 1.0, 0.0);
    glRotatef(30.0, 0.0, 1.0, 0.0);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_transparent);
```

```
    glEnable(GL_BLEND);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glCallList(cubeList);
    glDepthMask(GL_TRUE);
    glDisable(GL_BLEND);
  glPopMatrix();

  glutSwapBuffers();
}
```

16.6. Alpha3D reshape function

Nothing new here.

167a ⟨*Alpha3D* reshape *function* 167a⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, (GLint) w, (GLint) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h) {
    glOrtho(-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w, 1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
  }
  else {
    glOrtho(-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h,-1.5, 1.5, -10.0, 10.0);
  }
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

16.7. Alpha3D `main` function

Nothing new here.

168a ⟨*Alpha3D* `main` *function* 168a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutCreateWindow("3D Alpha Blending");
  init();
  glutKeyboardFunc(keyboard);
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return OK;
}
```

16.8. Alpha Source Code

The C program code is available at

ftp://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/alpha3D.c

17. Antialiasing Lines

This program draws shows how to draw anti-aliased (smoothed) lines and compares them with aliased (un-smoothed) lines. On the top of the screen it draws two anti-aliased diagonal lines to form an X. On the bottom of the screen it draws two aliased diagonal lines to form an X. When 'r' is typed in the window, the lines are rotated in opposite directions. Several things may be noticable:

1. When the lines are almost horizontal or almost vertical the jaggies of the aliased lines should be apparent.
2. The anti-aliased lines may not appear as bright or as wide as the aliased lines.
3. There may be more variation in color of the anti-aliased lines as they rotate.

Here's the standard organization of our programs:

169a ⟨*alias* 169a⟩≡
⟨*Alias header files* 170a⟩
⟨*Alias* `init` *function* 171a⟩
⟨*Alias* `display` *function* 172a⟩
⟨*Alias* `reshape` *function* 175a⟩
⟨*Alias* `keyboard` *function* 176a⟩
⟨*Alias* `main` *function* 177a⟩

17.1. Alias header files

A global variable `rotAngle` is defined to control the slopes of the intersection lines forming the X.

170a ⟨*Alias header files* 170a⟩≡

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

#define OK 0
#define ESC 27

static float rotAngle = 0.;
```

17.2. Alias initialization function

On initializaion, the implementation specific line width granularity and width range is printed out. For my implementation the `GL_LINE_WIDTH_GRANULARITY` value is 0.1 and the `GL_LINE_WIDTH_RANGE` values are 0.1 100.0. This means lines widths can be set from as small as 0.1 to as large as 100.0, and the width can be incremented in jumps of 0.1.

Line smoothing and and blending are enabled (we disable them when to draw aliased lines). The blending that occurs to filter the anti-aliased lines is to compute

$$(A_2R_2 + (1 - A_2)R_1,\ A_2G_2 + (1 - A_2)G_1,\ A_2B_2 + (1 - A_2)B_1,\ A_2A_2 + (1 - A_2)A_1)$$

where $(R_2,\ G_2,\ B_2, A_2)$ is the (source) color of the (new) line and $(R_1,\ G_1,\ B_1, A_1)$ is the (destination) color of the (old) background that was already in the color buffer.

171a ⟨*Alias* `init` *function* 171a⟩≡

```
void init(void) {
  GLfloat values[2];
  glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);
  printf ("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n", values[0]);

  glGetFloatv (GL_LINE_WIDTH_RANGE, values);
  printf ("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n", values[0], values[1]);

  glEnable (GL_LINE_SMOOTH);
  glEnable (GL_BLEND);
  glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
  glLineWidth (1.5);

  glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

17.3. Alias `display` function

Now we draw two sets of intersecting lines. In the lower portion of the view port draw aliased lines. In the upper portion draw anti-aliased lines.

172a ⟨*Alias* `display` *function* 172a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f (0.0, 1.0, 0.0);
  ⟨Draw aliased lines in lower portion of viewport 172b⟩
  ⟨Draw aliased lines in upper portion of viewport 173a⟩
  glFlush();
}
```

To draw the aliased lines disable smoothing and blending, move down a bit and draw the lines.

172b ⟨*Draw aliased lines in lower portion of viewport* 172b⟩≡

```
glDisable(GL_LINE_SMOOTH);
glDisable(GL_BLEND);
glPushMatrix();
  glTranslatef(0.0, -1.5, 0.0);
  ⟨Draw crossing lines 174a⟩
glPopMatrix();
```

To draw the anti-aliased lines re-enable smoothing and blending, move up a bit and draw the lines.

173a ⟨*Draw aliased lines in upper portion of viewport* 173a⟩≡

```
glEnable(GL_LINE_SMOOTH);
glEnable(GL_BLEND);
glPushMatrix();
  glTranslatef(0.0, 1.5, 0.0);
  ⟨Draw crossing lines 174a⟩
glPopMatrix();
```

Now draw the crossing lines. The first drawn is green, the coordinates: $(-0.5, 0.5)$ to $(0.5, -0.5)$ forming the X are rotated by `-rotAngle` about the z axis.

The second drawn is blue, and its vertices are rotated by `rotAngle` about the z axis.

174a ⟨*Draw crossing lines* 174a⟩≡

```
glColor3f (0.0, 1.0, 0.0);
glPushMatrix();
glRotatef(-rotAngle, 0.0, 0.0, 0.1);
glBegin (GL_LINES);
   glVertex2f (-0.5, 0.5);
   glVertex2f (0.5, -0.5);
glEnd ();
glPopMatrix();

glColor3f (0.0, 0.0, 1.0);
glPushMatrix();
glRotatef(rotAngle, 0.0, 0.0, 0.1);
glBegin (GL_LINES);
   glVertex2f (0.5, 0.5);
   glVertex2f (-0.5, -0.5);
glEnd ();
glPopMatrix();
```

17.4. Alias `reshape` function

Nothing new here.

175a ⟨*Alias* `reshape` *function* 175a⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
     gluOrtho2D (-1.0, 1.0,
        -2.0*(GLfloat)h/(GLfloat)w, 2.0*(GLfloat)h/(GLfloat)w);
  else
     gluOrtho2D (-1.0*(GLfloat)w/(GLfloat)h,
        1.0*(GLfloat)w/(GLfloat)h, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

17.5. Alias keyboard function

Typing r or R increments the rotation angle by 20 degrees.

176a ⟨*Alias* keyboard *function* 176a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
  switch (key) {
    case 'r':
    case 'R':
      rotAngle += 20.;
      if (rotAngle >= 360.) rotAngle = 0.;
      glutPostRedisplay();
      break;
    case ESC:  /*  Escape Key  */
      exit(0);
      break;
    default:
      break;
  }
}
```

17.6. Alias `main` function

Nothing new here.

177a ⟨*Alias* `main` *function* 177a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (200, 400);
  glutCreateWindow (argv[0]);
  init();
  glutReshapeFunc (reshape);
  glutKeyboardFunc (keyboard);
  glutDisplayFunc (display);
  glutMainLoop();
  return OK;
}
```

17.7. Antialias Source Code

The C program code is available at

ftp://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/alias.c

18. Fog

The quest for visual reality has led to blending of colors for translucent and other effects, and antialiasing to make objects appear more smooth. Now we'll show how to incorporate atomospheric effects into a graphics scene. The general term for this is *fog*, but it can be used to simulate haze, smoke, mist, pollution and other particulates that obscure visibility.

The program illustrates various forms of

`glFog()`.

Five red teapots are drawn, each at a different z distance from the eye, and in in different types of fog: exponential, exponential squared, and linear. These fog types select equations used in modeling fog. Fog color is blended with the incoming object color using a fog blending factors f:

- Exponential (`GL_EXP`): $f = e^{-dz}$
- Exponential squared (`GL_EXP2`): $f = e^{-(dz)^2}$
- Linear (`GL_LINEAR`): $f = \frac{e-z}{e-s}$

where

- z is the eye-coordinate distance between the viewpoint and the object.
- d is a *density* defined in the `glFog()` call.
- e is an *end* value defined in the `glFog()` call.
- s is a *start* value defined in the `glFog()` call.

Specifically, the calling sequence is

```
glFog*(GLenum name, TYPE parameter)
```

Parameter Name	
GL_FOG_MODE	
GL_FOG_DENSITY	
GL_FOG_START	
GL_FOG_END]	& [[parameter is an integer or floating-point value that specifies end, the far di
GL_FOG_INDEX]	& [[parameter specifies the index used to identify fog color in color index mode

Table 4: Fog parameters

where the parameter *name* can be one of the values listed in Table 4.

In RGBA mode fog color C_f and object color C_o are mixed as an affine combination with factor f, that is, the final color is

$$C = fC_o + (1 - f)C_f.$$

Here's the standard organization of our programs:

180a ⟨*fog* 180a⟩≡
- ⟨*Fog header files* 180b⟩
- ⟨*The* `cycleFog` *function* 181a⟩
- ⟨*Fog* `init` *function* 182a⟩
- ⟨*Display a red teapot* 184a⟩
- ⟨*Fog* `display` *function* 185a⟩
- ⟨*Fog* `reshape` *function* 186a⟩
- ⟨*Fog* `main` *function* 187a⟩

18.1. Fog header files

The global variable `fogMode` controls the type of fog used in the program.

180b ⟨*Fog header files* 180b⟩≡
```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

#define OK 0

GLint fogMode;
```

18.2. Cycling through the fog

Pressing a left mouse button chooses between 3 types of fog: exponential, exponential squared, and linear. The fog model starts as exponential: One click switches it to exponential squared, a second click switches it to linear.

Pressing the right mouse button exits the program.

181a ⟨*The* `cycleFog` *function* 181a⟩≡

```
void cycleFog (int button, int state, int x, int y) {
  if (GLUT_LEFT_BUTTON == button && GLUT_DOWN == state) {
      if (fogMode == GL_EXP) {
        fogMode = GL_EXP2;
        printf("Fog mode is GL_EXP2\n");
      }
      else if (fogMode == GL_EXP2 &&  GLUT_DOWN == state) {
        fogMode = GL_LINEAR;
        printf("Fog mode is GL_LINEAR\n");
      }
      else if (fogMode == GL_LINEAR) {
        fogMode = GL_EXP;
        printf("Fog mode is GL_EXP\n");
      }
      glFogi(GL_FOG_MODE, fogMode);
      glutPostRedisplay();
    }
  else if (GLUT_RIGHT_BUTTON == button) {
      exit(OK);
  }
  else { }
}
```

18.3. Fog initialization

Upon initialization the z-buffer is enabled, lighting is enabled, and light zero is positioned and enabled. The function

glLightModel*()

is used to control how specular reflection angles are computed. Recall, specular reflections is dependent on the position of the viewer, the position of the light source, and the orientation of the object. In the code below, we set the viewer *at infinity* so the angle between the viewer and reflected specular light is constant, increasing the efficiency of the rendering.

Also, the orientation of polygons is set to clockwise, and normals are generated automatically with renormalization of lengths after transformations.

Finally, fog is enabled. In particular, exponential fog (the default) is set, the fog color is set to opaque gray, and the density is set to 0.35. When the linear fog model is enabled by the mouse, the start and end values are 1.0 and 5.0 respectfully.

182a ⟨*Fog* `init` *function* 182a⟩≡

```
void init(void) {
  GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };
  GLfloat local_view[] = { GL_FALSE };

  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);

  glLightfv(GL_LIGHT0, GL_POSITION, position);
  glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

  glFrontFace(GL_CW);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);
```

```
  glEnable(GL_FOG);
  {
    GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

    fogMode = GL_EXP;
    glFogi(GL_FOG_MODE, fogMode);
    glFogfv(GL_FOG_COLOR, fogColor);
    glFogf(GL_FOG_DENSITY, 0.35);
    glHint(GL_FOG_HINT, GL_DONT_CARE);
    glFogi(GL_FOG_START, 1.0);
    glFogi(GL_FOG_END, 5.0);
    glClearColor(0.5, 0.5, 0.5, 1.0);
  }
}
```

18.4. Fog initialization

This function renders a teapot at position (x, y, z). Material colors for the teapot are set to a reddish tone. The numbers appear as magic and I would like to track them down.

184a ⟨*Display a red teapot* 184a⟩≡

```
void renderRedTeapot(GLfloat x, GLfloat y, GLfloat z) {
  float mat[4];

  glPushMatrix();
  glTranslatef (x, y, z);
  mat[0] = 0.1745; mat[1] = 0.01175; mat[2] = 0.01175; mat[3] = 1.0;
  glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
  mat[0] = 0.61424; mat[1] = 0.04136; mat[2] = 0.04136;
  glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
  mat[0] = 0.727811; mat[1] = 0.626959; mat[2] = 0.626959;
  glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
  glMaterialf (GL_FRONT, GL_SHININESS, 0.6*128.0);
  auxSolidTeapot(1.0);
  glPopMatrix();
}
```

18.5. Fog `display` function

The `display()` function draws 5 teapots in a row each at increasing z distances. That is the teapots on the right will be more obscured by the fog.

185a ⟨*Fog* `display` *function* 185a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  renderRedTeapot(-4.0, -0.5, -1.0);
  renderRedTeapot(-2.0, -0.5, -2.0);
  renderRedTeapot(0.0, -0.5, -3.0);
  renderRedTeapot(2.0, -0.5, -4.0);
  renderRedTeapot(4.0, -0.5, -5.0);
  glFlush();
}
```

18.6. Fog reshape function

Nothing new here.

186a ⟨*Fog* reshape *function* 186a⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= (h*3))
    glOrtho (-6.0, 6.0, -6.0*((GLfloat) h)/(GLfloat) w,
            6.0*((GLfloat) h)/(GLfloat) w, 0.0, 10.0);
  else
    glOrtho (-2.0*(GLfloat) w/((GLfloat) h),
            2.0*(GLfloat) w/((GLfloat) h), -2.0, 2.0, 0.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity ();
}
```

18.7. Fog `main` function

187a ⟨*Fog* `main` *function* 187a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(450, 150);
  glutCreateWindow("Teapots in Fog");
  init();
  glutMouseFunc(cycleFog);
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return OK;
}
```

19. Bitmaps and Image Data

A *bitmap* is a rectangular array with a single *bit* of data per pixel. Bitmaps are typically used for *fonts* and are similar to *masks* in that they overlay (are written on top of) another image.

Image data, sometimes called *pixmaps*, are also rectangular arrays, but typically contain multiple pieces of information about the pixel. For example, red, green, blue, and alpha data can be stored in a pixmap.

Notice that these are new data types that are different from the geometric primitives: points, lines, polygons, that have been the topic of rendering until now.

20. Pixel Storage

The reading and writing of pixel rectangle and texture data is controlled by the pixel store state. When you want to draw or read a pixel image, you pass a pointer to a rectangle in OpenGL, together with a width and height in pixels and a format that specifies the number of bytes per pixel. This sounds straight forward, but consider: Your application may need to extract a sub-rectangle; for performance the rows of pixels may need to begin on some regular byte alignment; and the byte order differs from machine to machine (Intel and DEC are little-endian while Sun, SGI, and Motorola are big-endian).

Figure 12 shows an array of pixels together with an subimage that is to be extracted. Parameters that determine how to find this subimage are indicated in Figure 12. Bytes in your application's address space get unpacked into or packed from the graphics processor memory. The function

```
glPixelStore*(GLenum name, TYPE parameter)
```

controls how data is packed (written) or unpacked (read) from OpenGL memory.

Some of the parameter names that can be set with `glPixelStore*()` are:

- `GL_PACK_ROW_LENGTH` and `GL_UNPACK_ROW_LENGTH` specify the number of pixels in each rectangle row of the larger image from which the subimage is extracted. The default is 0, in which case the row length is the same as the width used in `glReadPixels()`, `glDrawPixels()`, or `glCopyPixels()`.
- `GL_PACK_SKIP_PIXELS` and `GL_UNPACK_SKIP_PIXELS` specify the number of pixels (columns) to skip before the start of the subimage. By default this parameter is 00 so you start on the left hand edge.
- `GL_PACK_SKIP_ROWS` and `GL_UNPACK_SKIP_ROWS` specify the number of rows to skip before the start of the subimage. By default this parameter is 00 so you start on the bottom edge.

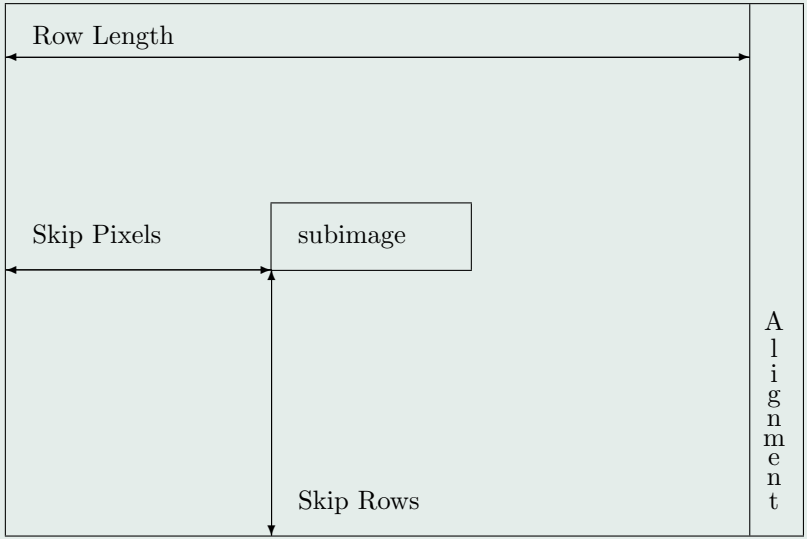

Figure 12: Image layout for pixel storage

- `GL_PACK_ALIGNMENT` and `GL_UNPACK_ALIGNMENT` specify byte alignment of the image. Hardware is often optimized for moving data that has a particular byte alignment, a four byte alignment would be common. If you image data is 3 bytes (one for each of red, green, and blue), and W pixels wide, then $3W$ bytes of data is needed per row. If the first row and each successive row begins on a 4 byte boundary. Then $4 - 3W \bmod 4$ bytes will be padded (wasted) in memory storage for each row.

A common programming mistake is to assume that pixel data is tightly packed at byte alignment (set to 1). The default alignment is 4. Let B, W, and H be the bytes per pixel, image width, and image height respectively. The space required to store an image with 4 byte alignment is *not* $B \times W \times H$. Instead it is

$$(((B * W + 3)/4) * 4) * H.$$

20.1. Current Raster Position

The function

`glRasterPos*()`

is used to set the screen position where the next bitmap or pixmap will be drawn. The coordinates passed to `glRasterPos*()` can be 2, 3, or 4 dimensional; listed explicitly or as a pointer to a vector; and of type: short, integer, float, or double. The coordinates are transformed exactly as vertex coordinates are, that is, by the model-view and projection matrices. After these transformations, the current raster position may be valid (inside the viewport) or invalid. Often, you'll want to set the current raster position in screen coordinates. To do so, make certain you've set up simple 2D rendering prior to setting the position and drawing the image.

191a ⟨*Simple 2D Rendering* 191a⟩≡

```
glMatrixMode(GL_PROJECTION);
glLoadIndentity();
gluOrtho2D(0.0, width, 0.0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIndentity();
```

⟨*Set the current raster position* (never defined)⟩
⟨*Draw the image* (never defined)⟩

20.2. Bitmap Drawing

The function

`glBitmap()`

draws a bitmap at the current raster position. This function takes several arguments. In order they are:

1. `GLsizei width` width of the bitmap in pixels
2. `GLsizei height` height of the bitmap in pixels
3. `GLfloat xorig` the x origin of the bitmap, relative to the current raster position
4. `GLfloat yorig` the y origin of the bitmap, relative to the current raster position
5. `GLfloat xmove` x increment to the current raster position after the bitmap is rasterized
6. `GLfloat ymove` y increment to the current raster position after the bitmap is rasterized
7. `const GLubyte *bitmap` a pointer to the bitmap.

To illustrate the use of `glPixelStore*()`, `glRasterPos*()` and `glBitmap()` consider the "draw F" code from the Redbook Woo et al. [1999].

193a ⟨*drawF* 193a⟩≡
⟨*Draw F header* 194a⟩
⟨*Draw F initialzation* 195a⟩
⟨*Draw F* `display` 196a⟩
⟨*Draw F* `reshape` 197a⟩
⟨*Draw F* `main` 197b⟩

20.3. Draw F Header

The interesting thing here is the `rasters` array of 24 bytes that defines the letter F. You should think of it as 16 bits wide (every two bytes define a row of pixel storage and 12 rows high. The F is defined from its lower left corner: Its stem is six rows high and defined by `0xc0, 0x00` or in binary `1100 0000`. The middle horizontal bar in the F is defined by two rows of `0xff, 0x00` (binary `1111 0000`) and the top bar of the F is defined by two rows of `0xff, 0xc0` (binary `1111 1100`).

194a ⟨*Draw F header* 194a⟩≡

```
#include <GL/glut.h>
#include <GL/glu.h>
#include <stdlib.h>

#define OK 0

GLuint fontOffset;
GLubyte rasters[24] = {
    0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
    0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
    0xff, 0xc0, 0xff, 0xc0};
```

20.4. Draw F Initialization

The initialization routine specifies that the unsigned byte array above should be unpacked into the graphics processor one byte at a time, this is with no padding.

195a ⟨*Draw F initialzation* 195a⟩≡

```
void init(void) {
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

20.5. Draw F Initialization

The state variable GL_CURRENT_RASTER_COLOR (and GL_CURRENT_RASTER_INDEX for color table look-up) is set by glColor (glIndex). But, these state variables are set when glRasterPos*() is called. So in the code below, the color of the bitmap is white, even though the color is set to red prior to calling glBitmap() three times.

Each time glBitmap() is called, an F is rendered and the current raster position is incremented by 12 pixels in x.

The raster color

196a ⟨*Draw F* display 196a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);
  glRasterPos2i (20.5, 20.5);
  glColor3f(1.0, 0.0, 0.0);
  glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
  glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
  glBitmap(10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
  glFlush();
}
```

20.6. Draw F Initialization

The reshape function sets up a simple 2D display so that the coordinated defining the F map directly to screen coordinates.

197a ⟨*Draw F* `reshape` 197a⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, w, 0, h, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
}
```

20.7. Draw F `main`

Nothing new here.

197b ⟨*Draw F* `main` 197b⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
  glutCreateWindow ("Drawing F");
  init();
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return OK;
}
```

20.8. Draw F Source Code

The C program code for Draw F is available at

ftp://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/drawF.c

21. Fonts

A common use for bitmaps is to define fonts. The glyphs in a font can be referenced as an index, for example, Unicode uses 65 for Latin A, etc. Display lists are useful in managing fonts. To draw a glyph we can map name display lists by the indices of the glyph and then call the appropriate display list to render the character. In particular, the command

```
glCallLists(GLsizei n, GLenum type, const GLvoid *lists)
```

draws n display lists, where each lists contains data of the given `type` (usually `GL_BYTE`) and `lists` is a pointer to an array of display list names.

There is a slight problem in this interpretation. We want to use the same index, 65 for A, no matter what the font: be it roman, bold, italic, whatever. This is fixed by the function

```
glListBase(GLunit offset)
```

that defines an `offset` to be added to the list names prior to their call. For example, roman font can have an offset of 1000, bold font and offset of 2000, etc. Then to get the correct glyph, set the offset and call the list of characters to be rendered.

Another potential problem is obtaining a contiguous list of unused display list numbers. But the

```
glGenLists(GLsizei range)
```

returns a block of unused `range` display list names.

The `font` program below shows how to define a complete array of characters.

199a ⟨*font* 199a⟩≡
⟨*Font header* 201a⟩
⟨*Make the font* 205a⟩
⟨*Font initialzation* 206a⟩
⟨*Font* `printString` 206b⟩

⟨*Font* `display` 207a⟩
⟨*Font* `reshape` 208a⟩
⟨*Font* `main` 208b⟩

21.1. Font Header

The trick of this program is to define the printable ASCII characters as a bitmap. This data is stored in the `rasters` array below. The characters in the font are stored in separate display lists, indexed by their ASCII code plus a `fontOffset`.

201a ⟨*Font header* 201a⟩≡

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

#define OK 0

GLuint fontOffset;

GLubyte rasters[][13] = {
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36, 0x36, 0x36, 0x36},
{0x00, 0x00, 0x00, 0x66, 0x66, 0xff, 0x66, 0x66, 0xff, 0x66, 0x66, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x7e, 0xff, 0x1b, 0x1f, 0x7e, 0xf8, 0xd8, 0xff, 0x7e, 0x18},
{0x00, 0x00, 0x0e, 0x1b, 0xdb, 0x6e, 0x30, 0x18, 0x0c, 0x76, 0xdb, 0xd8, 0x70},
{0x00, 0x00, 0x7f, 0xc6, 0xcf, 0xd8, 0x70, 0x70, 0xd8, 0xcc, 0xcc, 0x6c, 0x38},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x1c, 0x0c, 0x0e},
{0x00, 0x00, 0x0c, 0x18, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x18, 0x0c},
{0x00, 0x00, 0x30, 0x18, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x18, 0x30},
{0x00, 0x00, 0x00, 0x00, 0x99, 0x5a, 0x3c, 0xff, 0x3c, 0x5a, 0x99, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0xff, 0xff, 0x18, 0x18, 0x18, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x60, 0x60, 0x30, 0x30, 0x18, 0x18, 0x0c, 0x0c, 0x06, 0x06, 0x03, 0x03},
{0x00, 0x00, 0x3c, 0x66, 0xc3, 0xe3, 0xf3, 0xdb, 0xcf, 0xc7, 0xc3, 0x66, 0x3c},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x78, 0x38, 0x18},
```

```
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0xe7, 0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0x07, 0x03, 0x03, 0xe7, 0x7e},
{0x00, 0x00, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0xff, 0xcc, 0x6c, 0x3c, 0x1c, 0x0c},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x03, 0x03, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x03, 0x7f, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x1c, 0x1c, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x06, 0x0c, 0x18, 0x30, 0x60, 0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06},
{0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x06, 0x0c, 0x18, 0x30, 0x60},
{0x00, 0x00, 0x18, 0x00, 0x00, 0x18, 0x18, 0x0c, 0x06, 0x03, 0xc3, 0xc3, 0x7e},
{0x00, 0x00, 0x3f, 0x60, 0xcf, 0xdb, 0xd3, 0xdd, 0xc3, 0x7e, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x18},
{0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xfc},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xff},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x7e},
{0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06},
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff, 0xff, 0xe7, 0xc3},
{0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe3},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xe7, 0x7e},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x66, 0x3c},
```

```
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7, 0x7e},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3},
{0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x03, 0xff},
{0x00, 0x00, 0x3c, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x3c},
{0x00, 0x03, 0x03, 0x06, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0x30, 0x30, 0x60, 0x60},
{0x00, 0x00, 0x3c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x3c},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc3, 0x66, 0x3c, 0x18},
{0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x38, 0x30, 0x70},
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0x7f, 0x03, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x7e, 0xc3, 0xc0, 0xc0, 0xc0, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x03, 0x03, 0x03, 0x03, 0x03},
{0x00, 0x00, 0x7f, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30, 0x33, 0x1e},
{0x7e, 0xc3, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x00, 0x00, 0x18, 0x00},
{0x38, 0x6c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x00, 0x00, 0x0c, 0x00},
{0x00, 0x00, 0xc6, 0xcc, 0xf8, 0xf0, 0xd8, 0xcc, 0xc6, 0xc0, 0xc0, 0xc0, 0xc0},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x78},
{0x00, 0x00, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xfe, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xfc, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x7c, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x7c, 0x00, 0x00, 0x00, 0x00},
{0xc0, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0x00, 0x00, 0x00, 0x00},
{0x03, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x00, 0x00, 0x00, 0x00},
```

```
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe0, 0xfe, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xfe, 0x03, 0x03, 0x7e, 0xc0, 0xc0, 0x7f, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x1c, 0x36, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30, 0x00},
{0x00, 0x00, 0x7e, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0xe7, 0xff, 0xdb, 0xc3, 0xc3, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xc3, 0x66, 0x3c, 0x18, 0x3c, 0x66, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0xc0, 0x60, 0x60, 0x30, 0x18, 0x3c, 0x66, 0x66, 0xc3, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0xff, 0x60, 0x30, 0x18, 0x0c, 0x06, 0xff, 0x00, 0x00, 0x00, 0x00},
{0x00, 0x00, 0x0f, 0x18, 0x18, 0x18, 0x38, 0xf0, 0x38, 0x18, 0x18, 0x18, 0x0f},
{0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18},
{0x00, 0x00, 0xf0, 0x18, 0x18, 0x18, 0x1c, 0x0f, 0x1c, 0x18, 0x18, 0x18, 0xf0},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x8f, 0xf1, 0x60, 0x00, 0x00, 0x00}
};
```

21.2. Making the font

The routine `makeRasterFont` creates the font as a collection of display lists. 128 such lists are created: One for each symbol in a 7 bit alphabet. Data is stored in graphics memory one byte at a time (see `glPixelStore*()`)

To make the raster font as a collection of displays, 128 contiguous display list names are generated using `glGenLists(GLsizei range)`. Then in a loop over the printable characters (index 32 [space] to 126 [tilde]) a new display list is compiled to contain a bitmap that is 8×13 with an orgin at $(0, 2)$ relative to the current raster position, and an increment of $(10, 0)$ for updating the current raster position.

205a ⟨*Make the font* 205a⟩≡

```
void makeRasterFont(void) {
  GLuint i;
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

  fontOffset = glGenLists (128);
  for (i = 32; i < 127; i++) {
      glNewList(i+fontOffset, GL_COMPILE);
      glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, rasters[i-32]);
      glEndList();
  }
}
```

21.3. Making the font

The fonts are created upon initialization.

206a ⟨*Font initialzation* 206a⟩≡

```
void init(void) {
  glShadeModel(GL_FLAT);
  makeRasterFont();
}
```

21.4. Making the font

To print a string all we need do is call the display lists that define the letters in the string.

This might be the first time you've seen the `glPushAttrib(GLbitfield mask)` and `glPopAttrib(void)` functions. The store and restore state variables to and from the *attribute stack*. In this particular call, the `GL_LIST_BASE` setting is pushed and popped from the attribute stack. This protects the call to `glListBase()` that changed the base offset in calls to the display lists. That is, if other parts of the code use other offsets we don't want to have this part of the code interfere with their work.

Notice how conveniently the string `s` itself serves as an array of display list indices.

Everything above could be in a library that defines a font. Of course, to make it work, you've got to call `makeRasterFont()` before you start making calls to `printString()`.

206b ⟨*Font* `printString` 206b⟩≡

```
void printString(char *s) {
  glPushAttrib (GL_LIST_BIT);
  glListBase(fontOffset);
  glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
  glPopAttrib ();
}
```

21.5. Making the font

The `display` routine first renders each of the characters in `rasters`, and then prints the *pangram* "The quick brown fox jumps over a lazy dog."

207a ⟨*Font* `display` 207a⟩≡

```
void display(void) {
  GLfloat white[3] = { 1.0, 1.0, 1.0 };
  int i, j;
  char teststring[33];

  glClear(GL_COLOR_BUFFER_BIT);
  glColor3fv(white);
  for (i = 32; i < 127; i += 32) {
      glRasterPos2i(20, 200 - 18*i/32);
      for (j = 0; j < 32; j++)
          teststring[j] = (char) (i+j);
      teststring[32] = 0;
      printString(teststring);
  }
  glRasterPos2i(20, 100);
  printString("The quick brown fox jumps");
  glRasterPos2i(20, 82);
  printString("over a lazy dog.");
  glFlush ();
}
```

21.6. Making the font

Map coordinates directly to the screen.

208a ⟨*Font* `reshape` 208a⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho (0.0, w, 0.0, h, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
}
```

21.7. Making the font

Nothing new here.

208b ⟨*Font* `main` 208b⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
  glutCreateWindow ("A Complete Font");
  init();
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return OK;
}
```

21.8. Font Source Code

The C program code for Draw F is available at

ftp://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/font.c

22. Image Data

Images are similar to bitmaps but they contain multiple pieces of information about each pixel, usually, red, green, blue, and alpha values. There are three basic routines for manipulating images.

1. `glReadPixels()` reads pixels from the framebuffer and packs them in application memory.
2. `glDrawPixels()` unpacks pixels from application memory into the framebuffer at the current raster position
3. `glCopyPixels()` copies pixels from one part of the framebuffer to another.

This all sounds straightforward, but is complicated by multiple kinds of framebuffer data, multiple ways to store pixel data in application memory, and operations that might be performed on the data as it is copies, read, or written.

Another complication is deciding which buffer in the framebuffer will be read or written: The front or back color buffer, or some other buffer? The call

```
glReadBuffer(GLenum buffer)
```

selects the buffer to be read. The call

```
glDrawBuffer(GLenum buffer)
```

selects the buffer to be written. The `buffer` can be one of `GL_FRONT`, `GL_BACK`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`, `GL_LEFT`, `GL_RIGHT`, or `GL_AUX`i provided that all of these buffers are supported by your OpenGL implementation.

22.1. Reading Pixels

The call to read pixels from the framebuffer to application memory is

```
glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
             GLenum format, GLenum type, GLvoid *pixels)
```

where (x, y) is the lower left corner of the rectangle with stated `width` and `height` that is to be read into memory pointed to by `pixels`. The `format` specifies what kind of data to be read, and the `type` specifies the data type. For example, the format might be RGB of type floating-point. Allowable formats and types are listed in Table 6 and Table **??**, respectively. Not all data types are listed in Table **??**, see the Red Book Woo et al. [1999] for a complete list.

Format Constant	Meaning
GL_COLOR_INDEX	a single color table index
GL_STENCIL_INDEX	a single color table index
GL_DEPTH_COMPONENT	a single depth value
GL_RED	a single red component
GL_GREEN	a single red component
GL_BLUE	a single red component
GL_ALPHA	a single red component
GL_RGB	red, followed by green, followed by blue
GL_RGBA	red, green, blue, and alpha
GL_BGR	blue followed by green, followed by red
GL_BGRA	blue, green, red, and alpha
GL_LUMINANCE	a single luminance component
GL_LUMINANCE_ALPHA	a luminance followed by alpha

Table 5: Pixel formats

Type Constant	Meaning
GL_UNSIGNED_BYTE	an unsigned byte
GL_BYTE	an signed byte
GL_BITMAP	single bits in unsigned bytes
GL_UNSIGNED_SHORT	an unsigned 16-bit integer
GL_SHORT	a signed 16-bit integer
GL_UNSIGHED_INT	an unsigned 32-bit integer
GL_INT	a signed 32-bit integer
GL_FLOAT	a single precision floating point number
GL_UNSIGNED_BYTE_3_3_2	3 bits red, 3 bits green, 2 bits blue packed into unsigned byte
GL_UNSIGNED_BYTE_2_3_3_REV	packed into unsigned byte

Table 6: Data Types for Pixel

22.2. Writing Pixels

The call to write pixels from application memory to the framebuffer is

```
glDrawPixels(GLsizei width, GLsizei height, GLenum format, GLenum type,
             GLvoid *pixels)
```

The rectangle is drawn at the current raster position.

22.3. Copying Pixels

The call to copy pixels from one part of the framebuffer to another is

```
glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height,
             GLenum buffer)
```

The `buffer` is either `GL_COLOR`, `GL_STENCIL`, or `GL_DEPTH`.

The `image` program below illustrates the use of some of the functions described above, and some others.

213a ⟨*image* 213a⟩≡
⟨*Image header files* 213b⟩
⟨*Image global variables* 214a⟩
⟨*Make the image* 215a⟩
⟨*Image* `motion` *function* 217a⟩
⟨*Image* `keyboard` *function* 218a⟩
⟨*Image* `display` *function* 216a⟩
⟨*Image* `reshape` *function* 216b⟩
⟨*Image initialization function* 215b⟩
⟨*Image* `main` *function* 219a⟩

22.4. Image header inclusion

We continue to use GLUT.

213b ⟨*Image header files* 213b⟩≡
```
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
```

Here are some global constants and an array to hold red, green, and blue values for a 8×8 checker board where each checker square is an 8×8 pixel array.

A `zoomFactor` is used to scale the size of the image (note bitmaps cannot be scaled, but images can). The global variable `height` is set in `reshape` as the height of the viewport and used in `motion` to fix the lower left corner of the checkerboard as the mouse is moved around the window.

214a ⟨*Image global variables* 214a⟩≡

```
#define OK 0
#define ESC 27
#define checkImageWidth 64
#define checkImageHeight 64

GLubyte checkImage[checkImageWidth][checkImageHeight][3];

static GLdouble zoomFactor = 1.0;
static GLint height;
```

22.5. Defining an Image

This code also appears in the checkerboard texture program. It makes a 8×8 checker board that is composed of alternating black and white squares, each square an 8×8 array of pixels.

215a ⟨*Make the image* 215a⟩≡

```
void makeCheckImage(void) {
  int i, j, r, c;

  for (i = 0; i < checkImageWidth; i++) {
    for (j = 0; j < checkImageHeight; j++) {
      c = ((((i&0x8)==0)^((j&0x8))==0))*255;
          checkImage[i][j][0] = (GLubyte) c;
          checkImage[i][j][1] = (GLubyte) c;
          checkImage[i][j][2] = (GLubyte) c;
    }
  }
}
```

22.6. Image `init` function

Initializations set the clear color to black, creates the checkerboard image, and sets pixel storage to unsigned bytes.

215b ⟨*Image initialization function* 215b⟩≡

```
void init(void) {
  glClearColor(0.0, 0.0, 0.0, 0.0);
  makeCheckImage();
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}
```

22.7. Image `display` function

The display function sets the raster position to the lower left corner and draws the checkerboard.

216a ⟨*Image* `display` *function* 216a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glRasterPos2i(0,0);
  glDrawPixels(checkImageWidth, checkImageHeight, GL_RGB,
               GL_UNSIGNED_BYTE, checkImage);
  glFlush();
}
```

22.8. Image `reshape` function

Not much new here. The transformations are set up to draw directly to screen coordinates.

216b ⟨*Image* `reshape` *function* 216b⟩≡

```
void reshape(int w, int h) {
  glViewport(0, 0, w, h);
  height = (GLint) h;
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

22.9. Image `motion` function

The `motion` function is set as a callback by `glutMotionFunction(void (*func)(int x, int y))` The `motion()` function is called when the mouse is moving inside the window with a mouse button down.

What happens is that a static integer `screeny` records the location of the lower left corner of the checkerboard. Notice that `x` and `y` are in window coordinates, in particular, `y` is close to 0 at the top of the screen and close to `height` at the bottom of the screen.

The function

```
glPixelZoom(GLfloat xscale, GLfloat yscale)
```

scales the image by factors in `x` and `y`. The 64×64 image at the lower left corner of the window is copied to the current raster position set by `glRasterPos()`.

217a ⟨*Image* `motion` *function* 217a⟩≡

```
void motion(int x, int y) {
  static GLint screeny;
  screeny = height-(GLint)y;
  glRasterPos2i(x, screeny);
  glPixelZoom(zoomFactor, zoomFactor);
  glCopyPixels(0,0,checkImageWidth, checkImageHeight, GL_COLOR);
  glPixelZoom(1.0, 1.0);
  glFlush();
}
```

22.10. Image keyboard function

The keyboard function controls the size of the zoom, resetting the image size to its original value and exiting the application.

218a ⟨*Image* keyboard *function* 218a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
  switch (key) {
    case 'r':
    case 'R':
      zoomFactor = 1.0;
      glutPostRedisplay();
      printf("zoom factor reset to 1.0\n");
      break;
    case 'z':
      zoomFactor += 0.5;
      if (3.0 <= zoomFactor) { zoomFactor = 3.0; }
      printf("zoom factor is now %4.1f\n", zoomFactor);
      break;

    case 'Z':
      zoomFactor -= 0.5;
      if (0.5 >= zoomFactor) { zoomFactor = 0.5; }
      printf("zoom factor is now %4.1f\n", zoomFactor);
      break;
    case ESC:
      exit (0);
      break;
    default: break;
  }
}
```

22.11. Image `main` function

The `main` function contains little that is new.

219a ⟨*Image* `main` *function* 219a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(250, 250);
  glutInitWindowPosition(100, 100);
  glutCreateWindow ("Image Zooms");
  init();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMotionFunc(motion);
  glutMainLoop();
  return OK;
}
```

22.12. Pixel Transfer Operations

As pixels are transferred to or from the framebuffer operations can be performed on them.

23. Mip Mapping

Lance Williams proposed mipmaps in 1983 Williams [1983] as means antialias textures. The problem is this:

1. Dynamic objects in a scene can move relative to the view position.
2. As a textured object moves away from the viewpoint, the texture should shrink as the object decreases in size.
3. As a textured object moves toward from the viewpoint, the texture should expand as the object increases in size.

The graphics API must handle these situations, and there should be no unpleasant visual artifacts as the textured objects move: sparkles, twinkles, flashing, etc. That is, the graphics API must select the appropriate *level of detail* when select a texture to map to the object.

Mipmapping means to specify multiple levels of detail for a texture image [1]. The algorithm Williams describes filters an initial texture image by averaging adjacent texels, but we will not describe this here. A course on graphics algorithms might explain how the algorithm works.

OpenGL allows the programmer to specify each level of detail or for the levels of detail to be generated automatically based on Williams' algorithm.

The first program, *mipmap* from the Redbook Woo et al. [1999], explicitly creates each level of the mipmap. Mark Kilgard kilg:00 identifies *not* setting all mipmap levels as one of the 16 common OpenGL pitfall. All sizes of the texture map between the largest image and a 1×1 map must be specified. Texture widths and heights must be powers of 2: The the largest dimension is 2^n, then images of dimension $2^{n-1}, 2^{n-2}, \ldots, 2, 1$ must be given. More explicitly, suppose your largest image is 16×8, then images of size 8×4, 4×2, 2×1, and 1×1 must be provided, or generated automatically by the API.

[1] Mip is an acronym for the Latin *multum in parvo*, which means: "many things in a small place."

220a ⟨*mipmap* 220a⟩≡
⟨*Mipmap header files* 222a⟩
⟨*Mipmap* `makeImages` *function* 223a⟩
⟨*Mipmap* `init` *function* 226a⟩
⟨*Build mipmap checkerboard* 233a⟩
⟨*Mipmap* `display` *function* 229a⟩
⟨*Mipmap* `reshape` *function* 230a⟩
⟨*Mipmap* `keyboard` *function* 235a⟩
⟨*Texture coordinate generation* 237a⟩
⟨*Mipmap* `main` *function* 245a⟩

23.1. Mipmap header files

There are 6 global mipmap images declared starting with a largest one that is 32×32.

222a ⟨*Mipmap header files* 222a⟩≡

```
#include <GL/glut.h>
#include <GL/glu.h>
#include <stdlib.h>

#define OK 0
#define ESC 27

GLubyte mipmapImage32[32][32][3];
GLubyte mipmapImage16[16][16][3];
GLubyte mipmapImage8[8][8][3];
GLubyte mipmapImage4[4][4][3];
GLubyte mipmapImage2[2][2][3];
GLubyte mipmapImage1[1][1][3];

GLfloat xWidth = 2.0;
GLfloat zDepth = 0.0;
```

23.2. Mipmap `makeImages` function

The 6 mipmap images are fleshed out here by making the largest image yellow, the next size down magenta, the next red, and so on down to the smallest image which is white.

This is, perhaps, not typical of what would be done. Normally images at one level would be averages of images at higher levels. (In OpenGL, the initial image is said to be level 0, the next [smaller] image is level 1, and so on).

223a ⟨*Mipmap* `makeImages` *function* 223a⟩≡

```
void makeImages(void) {
  int i, j;

  for (i = 0; i < 32; i++) {
    for (j = 0; j < 32; j++) {
      mipmapImage32[i][j][0] = 255;
      mipmapImage32[i][j][1] = 255;
      mipmapImage32[i][j][2] = 0;
    }
  }
  for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
      mipmapImage16[i][j][0] = 255;
      mipmapImage16[i][j][1] = 0;
      mipmapImage16[i][j][2] = 255;
    }
  }
  for (i = 0; i < 8; i++) {
    for (j = 0; j < 8; j++) {
      mipmapImage8[i][j][0] = 255;
      mipmapImage8[i][j][1] = 0;
      mipmapImage8[i][j][2] = 0;
   }
  }
```

```
 for (i = 0; i < 4; i++) {
   for (j = 0; j < 4; j++) {
     mipmapImage4[i][j][0] = 0;
     mipmapImage4[i][j][1] = 255;
     mipmapImage4[i][j][2] = 0;
   }
 }
 for (i = 0; i < 2; i++) {
   for (j = 0; j < 2; j++) {
     mipmapImage2[i][j][0] = 0;
     mipmapImage2[i][j][1] = 0;
     mipmapImage2[i][j][2] = 255;
   }
 }
 mipmapImage1[0][0][0] = 255;
 mipmapImage1[0][0][1] = 255;
 mipmapImage1[0][0][2] = 255;
}
```

23.3. Mipmap `init` files

The `init` function uses `glTexImage2D()` to identify the above color arrays as 2D texture images at levels 0, 1, . . . , 5.

Then various texture parameters are set with `glTexParameter*()`. In particular,

- The texture is wrapped in the s (horizontal) and t (vertical) texture direction by *repeating* it when texture coordinates extend beyond the 0 to 1 boundaries. That is, the fractional part, $s - \lfloor s \rfloor$ of texture coordinate s (or t) is used to map into the texture. Another option is to *clamp* texture coordinates to 0.0 and 1.0
- Texture *magnification* occurs when a single texture element maps to multiple pixels: *minification* occurs when multiple texture elements map to a single pixel. Whether either of these occur is dependent on the texture coordinates, the size of the surface, and viewing conditions. Exactly which texture elements become involved in setting a pixel color can be controlled by the magnification and minification filters.

There are several ways to specify magnification and minification filters. Table 7 lists these options.

Filter	Value
GL_TEXTURE_MAG_FILTER	GL_NEAREST GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST GL_LINEAR GL_NEAREST_MIPMAP_NEAREST GL_NEAREST_MIPMAP_LINEAR GL_LINEAR_MIPMAP_NEAREST GL_LINEAR_MIPMAP_LINEAR

Table 7: Magnification and minifaction filtering methods.

The GL_NEAREST (*point sampling*) option for filtering selects the texture element nearest the center of the pixel being colored. The GL_LINEAR (*bilinear sampling*) option computes a weighted linear average of the 2×2 texture array that lies nearest the center of the pixel (for 1D textures and 3D textures 2×1 and $2 \times 2 \times 2$ arrays are used, respectively).

For mipmapping, more options are available for minification. The GL_NEAREST_MIPMAP_NEAREST option selects the nearest texture element in the mipmap nearest in size to the polygon being rendered. The GL_LINEAR_MIPMAP_NEAREST computes an average of texture elements in the 2×2 texture array nearest the pixel.

The next two filtering options avoid sudden transitions between the selection of the mipmap texture to use. GL_NEAREST_MIPMAP_LINEAR uses linear interpolation between the nearest texture element in the two mipmaps that best over- and under-estimate the size of the polygon. GL_LINEAR_MIPMAP_LINEAR (*trilinear sampling*) does a linear weighted average in the two surrounding mipmaps and then linearly interpolates these values.

The function

```
glTexEnv*(GLenum target, GLenum parameter, TYPE value)
```

sets the drawing mode for texture mapping. The `target` must be GL_TEXTURE_ENV. Two options exist for the `paramter`: GL_TEXTURE_ENV_MODE and GL_TEXTURE_ENV_COLOR.

If GL_TEXTURE_ENV_MODE is set, the `value` can be one of: GL_MODULATE, GL_DECAL, GL_BLEND, or GL_REPLACE. If GL_TEXTURE_ENV_COLOR is set, the `value` is an four tuple representing R, G, B, and A values; and these are used only if the GL_BLEND texture function has been set.

The most simple drawing mode is GL_REPLACE, where the color of the texture replaces the color of the pixel. The exact way in which this occurs depends on the internal format of the texture. For internal formats: GL_RGB and GL_RGBA, the computed color `C` and alpha value `A` are shown in Table 8. Colors C_t and C_f or alpha values A_t and A_f refer to the texture and current pixel color or alpha values, respectively. Colors C_c refer to the color set by GL_TEXTURE_ENV_COLOR.

Format	Replace	Decal	Blend	Modulate
RGB	$C = C_t$	$C = C_t$	$C = C_f(1 - C_t) + C_c C_t$	$C = C_f C_t$
	$A = A_f$	$A = A_f$	$A = A_f$	$A = A_f$
RGBA	$C = C_t$	$C = C_f(1 - A_t) + C_t A_t$	$C = C_f(1 - C_t) + C_c C_t$	$C = C_f C_t$
	$A = A_t$	$A = A_f$	$A = A_f A_t$	$A = A_f A_t$

Table 8: Texture functions

226a ⟨*Mipmap* `init` *function* 226a⟩≡

```
void basicinit(void) {
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  glShadeModel(GL_FLAT);

  makeImages();
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glTexImage2D(GL_TEXTURE_2D, 0, 3, 32, 32, 0,
               GL_RGB, GL_UNSIGNED_BYTE, &mipmapImage32[0][0][0]);
  glTexImage2D(GL_TEXTURE_2D, 1, 3, 16, 16, 0,
               GL_RGB, GL_UNSIGNED_BYTE, &mipmapImage16[0][0][0]);
  glTexImage2D(GL_TEXTURE_2D, 2, 3, 8, 8, 0,
               GL_RGB, GL_UNSIGNED_BYTE, &mipmapImage8[0][0][0]);
  glTexImage2D(GL_TEXTURE_2D, 3, 3, 4, 4, 0,
               GL_RGB, GL_UNSIGNED_BYTE, &mipmapImage4[0][0][0]);
  glTexImage2D(GL_TEXTURE_2D, 4, 3, 2, 2, 0,
               GL_RGB, GL_UNSIGNED_BYTE, &mipmapImage2[0][0][0]);
  glTexImage2D(GL_TEXTURE_2D, 5, 3, 1, 1, 0,
               GL_RGB, GL_UNSIGNED_BYTE, &mipmapImage1[0][0][0]);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

```
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST_MIPMAP_NEAREST);

// try this using:
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
}
```

23.4. Mipmap `display` files

The `display` function creates a quadrangle that recedes in the distance, so it looks almost like a triangle when viewed under a perspective projection. The texture coordinates ranging from 0.0 8.0 in s and t are mapped to corners of the quadrangle, guaranteeing that 64 copies of the texture map must be used to tile the quadrangle.

A scale factor (ρ) between texture size and polygon size determines which mipmap level is selected in rendering a pixel. The actual selection is based on another parameter $\lambda = \lg \rho$, which selects the *level* of the mipmap. If $\lambda \leq 0.0$, then $\rho \leq 1$ and the texture is smaller than the polygon, so a magnification filter is use. If $\lambda > 0.0$, a minification filter is used.

Suppose the bounding box for the polygon has size $P_x \times P_y$ in pixels, and texture sizes are $T_s \times T_t$. Then scale factor ρ is

$$\rho = \max\{T_s/P_x,\, T_t/P_y\}$$

For example, if the texture is 64×8 and the polygon size is 8×4, the level is $\lambda = 8$.

229a ⟨*Mipmap* `display` *function* 229a⟩≡

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 1.0);
    glTexCoord2f(0.0, 8.0); glVertex3f(-2.0, 1.0, 1.0);
    glTexCoord2f(8.0, 8.0); glVertex3f(xWidth, 1.0, zDepth);
    glTexCoord2f(8.0, 0.0); glVertex3f(xWidth, -1.0, zDepth);
  glEnd();
  glFlush();
}
```

23.5. Mipmap `reshape` files

The perspective projection another feature that controls the mapping of the texture to the polygon. By printing it out we can see exactly what the map it and thus how texture coordinates map to pixel coordinates.

230a ⟨*Mipmap* `reshape` *function* 230a⟩≡

```
void reshape(int w, int h) {
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 30000.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -3.6);

   ⟨Print the quadrangle coordinates 231a⟩
}
```

The coordinates of the quadrangle are multiplied by model-view matrix then the projection matrix. Since the model view matrix is the identity, we only need to know the projection matrix. With this we can better see how the texture is mapped to the quadrangle.

231a ⟨*Print the quadrangle coordinates* 231a⟩≡

```
{
  GLfloat matrix[16], out[16];
  GLfloat in[16] = { -2.0, -2.0,  2000.0,  2000.0,
                     -1.0,  1.0,     1.0,    -1.0,
                      1.0,  1.0, -6000.0, -6000.0,
                      1.0,  1.0,     1.0,     1.0 };

  GLfloat sum = 0.0;

  int i, j, k;

  glGetFloatv(GL_PROJECTION_MATRIX, matrix);
  printf("The projection tranformation is\n");
  for (i = 0; i < 4; i++) {
    printf("\t[");
    for (j = 0; j < 4; j++) {
      printf("%f\t",matrix[4*i+j]);
    }
    printf("]\n");
  }

  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      sum = 0.0;
      for (k = 0; k < 4; k++) {
        sum += matrix[4*i+k]*in[4*k+j];
      }
      out[4*i+j] = sum;
```

```
      }
    }

//    glGetIntegerv(GL_MAX_VIEWPORT_DIMS, viewport);
//    printf("Maximum viewport size x = %d, y = %d\n", viewport[0], viewport[1]);

    printf("The transformed coordinates are\n");
    for (i = 0; i < 4; i++) {
      printf("\t[");
      for (j = 0; j < 4; j++) {
        printf("%f\t", out[4*i+j]/out[12+j]);
      }
      printf("]\n");
    }
  }
```

23.6. Building a mipmap

The routine `gluBuild*DMipmaps()` automatically constructs a series of mipmaps and calls `glTexImage*()` to load them. We'll illustrate its use with the checkerboard pattern.

233a ⟨*Build mipmap checkerboard* 233a⟩≡

```
void checkerinit() {
  GLubyte checkerboard[64][64][3];
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LESS);
  glShadeModel(GL_FLAT);

  glTranslatef(0.0, 0.0, -3.6);
  ⟨Make checkerboard 234a⟩
  gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, 64, 64, GL_RGB, GL_UNSIGNED_BYTE, &checkerboard[
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// try this using:
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
//  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
}
```

234a ⟨*Make checkerboard* 234a⟩≡

```
{
  int i, j, r, c;

  for (i = 0; i < 64; i++) {
    for (j = 0; j < 64; j++) {
      c = ((((i&0x8)==0)^((j&0x8))==0))*255;
          checkerboard[i][j][0] = (GLubyte) c;
          checkerboard[i][j][1] = (GLubyte) c;
          checkerboard[i][j][2] = (GLubyte) c;
    }
  }
}
```

23.7. Mipmap keyboard files

235a ⟨*Mipmap* keyboard *function* 235a⟩≡

```
void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case 'x':
          xWidth += 1.0;
          glutPostRedisplay();
          printf("x=%f\n", xWidth);
          break;
      case 'X':
          xWidth -= 1.0;
          glutPostRedisplay();
          printf("x=%f\n", xWidth);
          break;
      case 'z':
          zDepth += 1.0;
          glutPostRedisplay();
          printf("z=%f\n", zDepth);
          break;
      case 'Z':
          zDepth -= 1.0;
          glutPostRedisplay();
          printf("z=%f\n", zDepth);
          break;
      case ESC:
         exit(OK);
         break;
      default:
         break;
   }
}
```

23.8. Automatic texture coordinates

Computing good texture coordinates is problematic. Textures are rectangular arrays that are powers of 2 in dimension, and have coordinates that range for 0 to 1. To avoid distortions in the texture, aspect ratio of the polygon should match the aspect ratio of the texture, or you should only use a portion of the texture that matches the aspect ratio of the polygon. The Red Book Woo et al. [1999] offers some advice on how to specify texture coordinates.

An alternative is to let OpenGL generate texture coordinates for you. The function

```
glTexGen*(GLenum coord, GLenum parameter, TYPE (*)value)
```

specifies how texture coordiantes are generated. The `coord` can be one of `GL_S`, `GL_T`, `GL_R` or `GL_Q`, identifying which coordinate is to be generated. The `parameter` can be one of `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE` or `GL_EYE_PLANE`.

When `GL_TEXTURE_GEN_MODE` is the parameter, you select the function to be used in coordinate generation by setting the value to one of `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR` or `GL_SPHERE_MAP`. When either of the other parameters are selected, the `value` is a pointer to an array of values specifying parameters for the texture generation funciton.

`GL_OBJECT_LINEAR` is used when the texture remains fixed on a moving object, for example a brick wall. The texture coordinate, say s, is a linear combination of the object coordinates:

$$s = p_0 x_o + p_1 y_o + p_2 z_o + p_3 w_o$$

where the parameters (p_0, p_1, p_2, p_3) are set as the `vector` in `glTexGen*v(GL_S, GL_OBJECT_PLANE, vector)`. Notice that the texture coordinate is calculated using the model's object coordinates. When the parameters (p_0, p_1, p_2, p_3) are normalized to unit length, the formula

$$s = p_0 x_o + p_1 y_o + p_2 z_o + p_3 w_o$$

gives the distance from the object point (x_o, y_o, z_o, w_o) to the plane $p_0 x + p_1 y + p_2 z + p_3 w = 0$.

GL_EYE_LINEAR is used when the texture moves relative to a fixed object. Texture coordinate calculation is similar to the above, but now

$$s = p'_0 x_e + p'_1 y_e + p'_2 z_e + p'_3 w_e$$

where (x_e, y_e, z_e, w_e) are eye coordinates and

$$(p'_0, p'_1, p'_2, p'_3) = (p_0, p_1, p_2, p_3)M^{-1}$$

where M is the model-view matrix.

To illustrate how this works, consider the **texgen** program from the Red Book Woo et al. [1999].

237a ⟨*Texture coordinate generation* 237a⟩≡
- ⟨*Define the* **stripeImage** 237b⟩
- ⟨*Make a stripe image* 238a⟩
- ⟨*Define reference plane* 238b⟩
- ⟨*Texture generation* **init** 239a⟩
- ⟨*Texture generation* **display** 241a⟩
- ⟨*Texture generation* **reshape** 242a⟩
- ⟨*Texture generation* **keyboard** 243a⟩

The first thing to do is define a 32 element array **stripeImage** that contains a RGBA color in each element. A red stripe 4 pixels wide is followed by a green stripe that is 28 pixels wide.

237b ⟨*Define the* **stripeImage** 237b⟩≡
```
#define stripeImageWidth 32
GLubyte stripeImage[4*stripeImageWidth];

static GLuint texName;
```

238a ⟨*Make a stripe image* 238a⟩≡

```
void makeStripeImage(void) {
  int j;

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

Thre are two reference planes used in the sample program. One is $x = 0$ and the other is $x + y + z = 0$.

A teapot, sitting on the xy plane is drawn with one of these reference planes defining the contour of the red-green striped image. Initially, the `equalzero` plane is used and the texture coordinates `s` equals the object coordinate `x`. The red stripe will appear vertical along the teapot even as the teapot rotates.

Using the keyboard, the reference plane can be set to `slanted` and the `s` texture coordinate will be determined by `s=x+y+z`.

238b ⟨*Define reference plane* 238b⟩≡

```
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
static GLfloat *currentCoeff;
static GLenum currentPlane;
static GLint currentGenMode;
static GLfloat tiltangle;
```

23.8.1. Automatic texture `init` function

Upon initialization, we set the clear color to black, enable depth testing and smooth shading, and create the one dimensional red-green striped image. A unique texture name or id is generated using `glGenTextures()`. This is texture name is then bound, using `glBindTextures()`, creating a new texture object whose parameters values are set by subsequent calls to `glTexParamater*()`. The advantage of this is the multiple textures can be created, and when bound, using the `glBindTextures()` again, their properties become the current state for texture mapping.

239a ⟨*Texture generation* `init` 239a⟩≡

```
void texgeninit(void) {
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_SMOOTH);

  makeStripeImage();
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_1D, texName);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
   glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, stripeImageWidth, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);
   glTexImage1D(GL_TEXTURE_1D, 0, 4, stripeImageWidth, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, stripeImage);

   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
   currentCoeff = xequalzero;
   currentGenMode = GL_OBJECT_LINEAR;
   currentPlane = GL_OBJECT_PLANE;
```

```
    tiltangle = 45.0;
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
    glTexGenfv(GL_S, currentPlane, currentCoeff);

    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_1D);
    glEnable(GL_CULL_FACE);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glFrontFace(GL_CW);
    glCullFace(GL_BACK);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);
}
```

23.8.2. Automatic texture `display` function

The `display` function makes the texture initialized the texture used, and sets it right before calling the routine to render the teapot.

241a ⟨*Texture generation* `display` 241a⟩≡

```
void texgendisplay(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix ();
   glRotatef(tiltangle, 0.0, 0.0, 1.0);
   glBindTexture(GL_TEXTURE_1D, texName);
   glutSolidTeapot(2.0);
   glPopMatrix ();
   glFlush();
}
```

23.8.3. Automatic texture reshape function

242a ⟨*Texture generation* reshape 242a⟩≡

```
void texgenreshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho (-3.5, 3.5, -3.5*(GLfloat)h/(GLfloat)w,
               3.5*(GLfloat)h/(GLfloat)w, -3.5, 3.5);
   else
      glOrtho (-3.5*(GLfloat)w/(GLfloat)h,
               3.5*(GLfloat)w/(GLfloat)h, -3.5, 3.5, -3.5, 3.5);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

23.8.4. Automatic texture keyboard function

The keyboard function controls the generation of texture coordinates.

243a ⟨*Texture generation* keyboard 243a⟩≡

```
void texgenkeyboard (unsigned char key, int x, int y)
{
   switch (key) {
      case 'e':
      case 'E':
         currentGenMode = GL_EYE_LINEAR;
         currentPlane = GL_EYE_PLANE;
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 'o':
      case 'O':
         currentGenMode = GL_OBJECT_LINEAR;
         currentPlane = GL_OBJECT_PLANE;
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 'r':
      case 'R':
         tiltangle += 10.0;
         glutPostRedisplay();
         break;
      case 's':
      case 'S':
         currentCoeff = slanted;
         glTexGenfv(GL_S, currentPlane, currentCoeff);
```

```
            glutPostRedisplay();
            break;
         case 'x':
         case 'X':
            currentCoeff = xequalzero;
            glTexGenfv(GL_S, currentPlane, currentCoeff);
            glutPostRedisplay();
            break;
         case 27:
            exit(0);
            break;
         default:
            break;
      }
   }
```

23.9. Mipmap `main` files

245a ⟨*Mipmap* `main` *function* 245a⟩≡

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH) ;
  glutInitWindowSize (500, 500);
  glutCreateWindow ("Mipmapping Example");
//  basicinit();  // for the first simple texture map program.
//  checkerinit(); // for the checkerboard - build texture maps
  texgeninit(); // for the texture generation example.
//  glutKeyboardFunc(keyboard);
  glutKeyboardFunc(texgenkeyboard);
//  glutReshapeFunc(reshape);
//  glutDisplayFunc(display);
  glutReshapeFunc(texgenreshape);
  glutDisplayFunc(texgendisplay);
  glutMainLoop();
  return OK;
}
```

23.10. Mipmap Source Code

The C program code is available at

ftp://www.cs.fit.edu/wds/classes/prog-graphics/src/redBook/mipmap.c

References

Brian W. Kernighan and Dennis M. Ritchie. *The* `C` *Programming Language*. Prentice Hall, second edition, 1988. [Beginning].

Mark J. Kilgard. *Programming OpenGL for the X Window System*. Addison-Wesley, 1996. ISBN 0-201-48359-9.

Dave Shreiner, editor. *OpenGL Reference Manual*. Addison–Wesley, 3rd edition, 1999. 0-201-65765-1.

Lance Williams. Pyramidal parametrics. *Proceedings of SIGGRAPH '83*, 17(3):1–11, 1983.

Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison–Wesley, 3rd edition, 1999. 0-201-46138-2.

23.11. Index to Chuncks

Home Page
Title Page
Contents
Page 247 of 254
Go Back
Full Screen
Close
Quit

⟨*Double* `reshape` *the window callback* 23a⟩
⟨*Buffer support* 35a⟩
⟨*Clip support* 37b⟩
⟨*Convolution filters* 40b⟩
⟨*Current values* 41a⟩
⟨*Display list support* 38b⟩
⟨*Implementation-dependent states* 33a⟩
⟨*Initialize the graphics system* 30b⟩
⟨*Light support* 37a⟩
⟨*Lighting state variables* 43a⟩
⟨*Line information* 39c⟩
⟨*Local variables* 31a⟩
⟨*Point information* 39b⟩
⟨*Polynomial evaluation* 40a⟩
⟨*Rasterization state variables* 45a⟩
⟨*Stack support* 38a⟩
⟨*state* 29a⟩
⟨*State Header files* 29b⟩
⟨*State* `main` *function* 30a⟩
⟨*Transformation state variables* 42a⟩
⟨*Version information* 34a⟩
⟨*Viewport size* 39a⟩
⟨*Display GLUT models* 62a⟩
⟨*Display line loop* 55a⟩
⟨*Display line strips* 54a⟩
⟨*Display lines* 53a⟩
⟨*Display points* 52a⟩
⟨*Display polygon* 61a⟩
⟨`display` *primitives* 50a⟩

Florida Tech
Home Page
Title Page
Contents
◀◀ ▶▶
◀ ▶
Page 249 of 254
Go Back
Full Screen
Close
Quit

⟨*Define the vertices* 80b⟩
⟨`display` *cube* 80a⟩
⟨*Draw its faces as triangles* 82a⟩
⟨*Call me again to draw the new triangles* 93c⟩
⟨*Compute midpoints* 92b⟩
⟨*Define the triangles* (never defined)⟩
⟨*define the triangles* 88b⟩
⟨*Define the vertices* 88a⟩
⟨`display` *sphere by recursion* 87a⟩
⟨*Draw a cube as an approximation (12 triangles)* 89a⟩
⟨*Draw a triangle* 94a⟩
⟨*Draw the triangle and return if depth reached* 92a⟩
⟨*Normalize a vector* 93a⟩
⟨*Push them out to lie on the sphere* 93b⟩
⟨*Recursive subdivision function* 91a⟩
⟨*subdivide* 84a⟩
⟨*Subdivide again to 196 triangles* 89c⟩
⟨*Subdivide again to 784 triangles* 90a⟩
⟨*Subdivide header files* 84b⟩
⟨*Subdivide initialization function* 90b⟩
⟨*Subdivide* `main` *function* 86a⟩
⟨*Subdivide the cube to 48 triangles* 89b⟩
⟨*lines* 95a⟩
⟨*Lines* `display` *and redisplay* 97a⟩
⟨*Lines fifth row, 1 line, with dash/dot/dash stipple* 100b⟩
⟨*Lines first row, 3 lines, each with a different stipple* 98a⟩
⟨*Lines fourth row, 6 independent lines with same stipple* 100a⟩
⟨*Lines header files* 95b⟩
⟨*Lines initialization function* 96a⟩

Florida Tech
Home Page
Title Page
Contents
◀◀ ▶▶
◀ ▶
Page 250 of 254
Go Back
Full Screen
Close
Quit

⟨*Lines* `keyboard` *event callback* 102a⟩
⟨*Lines* `main` *function* 103a⟩
⟨*Lines* `reshape` *the window callback* 101a⟩
⟨*Lines second row, 3 wide lines, each with different stipple* 99a⟩
⟨*Lines third row, 6 lines, with dash/dot/dash stipple* 99b⟩
⟨*polys* 105a⟩
⟨*Polys* `display` *function* 106a⟩
⟨*Polys header files* 105b⟩
⟨*Polys initialization function* 108a⟩
⟨*Polys* `keyboard` *event callback* 109a⟩
⟨*Polys* `main` *function* 110a⟩
⟨*Polys* `reshape` *the window callback* 108b⟩
⟨*Enable lighting and hidden surface removal* 117a⟩
⟨`init` *local variables* 115a⟩
⟨*light* 112a⟩
⟨*Light* `display` *function* 117b⟩
⟨*Light header files* 112b⟩
⟨*Light initialization function* 114a⟩
⟨*Light* `keyboard` *function* 119a⟩
⟨*Light* `main` *function* 113a⟩
⟨*Light* `reshape` *function* 118a⟩
⟨*Set material properties* 116a⟩
⟨*Set the light properties* 116b⟩
⟨*Set the shading model* 116c⟩
⟨*movelight* 121a⟩
⟨*Moving light* `display` *function* 124a⟩
⟨*Moving light header files* 121b⟩
⟨*Moving light initialization function* 123a⟩
⟨*Moving light* `keyboard` *function* 127a⟩

Florida Tech
Home Page
Title Page
Contents
◀◀ ▶▶
◀ ▶
Page 253 of 254
Go Back
Full Screen
Close
Quit

⟨*Fog* `init` *function* 182a⟩
⟨*Fog* `main` *function* 187a⟩
⟨*Fog* `reshape` *function* 186a⟩
⟨*The* `cycleFog` *function* 181a⟩
⟨*Draw F* `display` 196a⟩
⟨*Draw F header* 194a⟩
⟨*Draw F initialzation* 195a⟩
⟨*Draw F* `main` 197b⟩
⟨*Draw F* `reshape` 197a⟩
⟨*Draw the image* (never defined)⟩
⟨*drawF* 193a⟩
⟨*font* 199a⟩
⟨*Font* `display` 207a⟩
⟨*Font header* 201a⟩
⟨*Font initialzation* 206a⟩
⟨*Font* `main` 208b⟩
⟨*Font* `printString` 206b⟩
⟨*Font* `reshape` 208a⟩
⟨*image* 213a⟩
⟨*Image* `display` *function* 216a⟩
⟨*Image global variables* 214a⟩
⟨*Image header files* 213b⟩
⟨*Image initialization function* 215b⟩
⟨*Image* `keyboard` *function* 218a⟩
⟨*Image* `main` *function* 219a⟩
⟨*Image* `motion` *function* 217a⟩
⟨*Image* `reshape` *function* 216b⟩
⟨*Make the font* 205a⟩
⟨*Make the image* 215a⟩

⟨*Set the current raster position* (never defined)⟩
⟨*Simple 2D Rendering* 191a⟩
⟨*Build mipmap checkerboard* 233a⟩
⟨*Define reference plane* 238b⟩
⟨*Define the* `stripeImage` 237b⟩
⟨*Make a stripe image* 238a⟩
⟨*Make checkerboard* 234a⟩
⟨*mipmap* 220a⟩
⟨*Mipmap* `display` *function* 229a⟩
⟨*Mipmap header files* 222a⟩
⟨*Mipmap* `init` *function* 226a⟩
⟨*Mipmap* `keyboard` *function* 235a⟩
⟨*Mipmap* `main` *function* 245a⟩
⟨*Mipmap* `makeImages` *function* 223a⟩
⟨*Mipmap* `reshape` *function* 230a⟩
⟨*Print the quadrangle coordinates* 231a⟩
⟨*Texture coordinate generation* 237a⟩
⟨*Texture generation* `display` 241a⟩
⟨*Texture generation* `init` 239a⟩
⟨*Texture generation* `keyboard` 243a⟩
⟨*Texture generation* `reshape` 242a⟩