Chapter 4

Display Lists

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand how clients and servers work together in a networked OpenGL system
- Understand how display lists can be used along with commands in immediate mode to improve performance
- Maximize performance by knowing how and when to use display lists

A display list is a group of OpenGL commands that have been stored for later execution. When a display list is invoked, the commands in it are executed in the order in which they were issued. Most OpenGL commands can be either stored in a display list or issued in immediate mode, which causes them to be executed immediately. You can freely mix immediate−mode programming and display lists within a single program. The programming examples you've seen so far have used immediate mode. This chapter discusses what display lists are and how best to use them. It has the following major sections:

- **"An Example of Using a Display List"** gives a brief example, showing the basic commands for using display lists.
- **"Display−List Design Philosophy"** explains when to use display lists.
- **"Creating and Executing a Display List"** discusses in detail the commands for creating and executing display lists.
- **"Managing Display Lists and Their Indices"** explains how to let OpenGL generate display−list indices for you automatically.
- **"Executing Multiple Display Lists"** shows how to execute several display lists in succession, using a small character set as an example.
- **"Encapsulating Mode Changes"** tells you how to use display lists to switch efficiently among different modes.

An Example of Using a Display List

A display list is a convenient and efficient way to name and organize a set of OpenGL commands. For example, suppose you want to draw a circle with 100 line segments. Without using display lists, you might write immediate−mode code like this:

```
drawCircle()
{
   GLint i;
   GLfloat cosine, sine;

   glBegin(GL_POLYGON);
      for(i=0;i<100;i++){
         cosine=cos(i*2*PI/100.0);
         sine=sin(i*2*PI/100.0);
         glVertex2f(cosine,sine);
      }
   glEnd();
}
```

This method is terribly inefficient because the trigonometry has to be performed each time the circle is rendered. Instead, you could save the coordinates in a table, and then pull the coordinates out of the table as needed:

```
drawCircle()
{
   GLint i;
   GLfloat cosine, sine;
   static GLfloat circoords[100][2];
   static GLint inited=0;

   if(inited==0){
      inited=1;
      for(i=0;i<100;i++){
         circcoords[i][0]=cos(i*2*PI/100.0);
         circcoords[i][1]=sin(i*2*PI/100.0);
      }
   }
   glBegin(GL_POLYGON);
      for(i=0;i<100;i++)
         glVertex2fv(&circcoords[i][0]);
   glEnd();
}
```

Even with this improved method, you still incur a slight penalty from incrementing and testing the variable *i*. What you really want to do is draw the circle once and have OpenGL remember how to draw it for later use. This is exactly what a display list is for, as shown in **Example 4−1**.

Example 4−1 Creating a Display List

```
#define MY_CIRCLE_LIST 1

buildCircle()
{
   GLint i;
   GLfloat cosine, sine;

   glNewList(MY_CIRCLE_LIST, GL_COMPILE);
      glBegin(GL_POLYGON);
         for(i=0;i<100;i++){
            cosine=cos(i*2*PI/100.0);
            sine=sin(i*2*PI/100.0);
            glVertex2f(cosine,sine);
         }
      glEnd();
   glEndList();
}
```

Note that the code for drawing a circle is bracketed by *glNewList()* and *glEndList()*. As you might have guessed, these commands define a display list. The argument MY_CIRCLE_LIST for *glNewList()* is an integer index that uniquely identifies this display list. You can execute the display list later with this *glCallList()* command:

```
glCallList(MY_CIRCLE_LIST);
```

A display list contains only OpenGL calls. Other calls—in **Example 4−1**, the C functions *cos()* and *sin()*—aren't stored in the display list. Instead, the coordinates and other variables (such as array contents) are evaluated and copied into the display list with the values they have when the list is compiled. After such a list has been compiled, these values can't be changed. You can delete a display list and create a new one, but you can't edit an existing display list.

Display−List Design Philosophy

OpenGL display lists are designed to optimize performance, particularly over networks, but never at the expense of performance on a local machine. A display list resides with the OpenGL server state, which in a networked environment might be on a different machine than the host (or client state). **"What Is OpenGL?"** discusses OpenGL's client−server model.

To optimize performance, an OpenGL display list is a cache of commands rather than a dynamic database. In other words, once a display list is created, it can't be modified. If a display list were modifiable, performance could be reduced by the overhead required to search through the display list and to perform memory management. As portions of a modifiable display list were changed, memory allocation and deallocation might lead to memory fragmentation. Using display lists is typically at least as fast as using immediate mode. Display lists can substantially increase performance—particularly when you issue OpenGL routines across networks, since display lists reside with the server and network traffic is minimized.

Even locally, a display list might be more efficient since it can be processed as it's created into a form that's more compatible with the graphics hardware. The particular commands that are so optimized may vary from implementation to implementation. For example, a command as simple as *glRotate*()* might show a significant improvement if it's in a display list, since the calculations to produce the rotation matrix aren't trivial (they can involve square roots and trigonometric functions). In the display list, however, only the final rotation matrix needs to be stored, so a display−list rotation command can be executed as fast as the hardware can execute *glMultMatrix()*. A sophisticated OpenGL implementation might even concatenate adjacent transformation commands into a single matrix multiplication.

Although you're not guaranteed that your OpenGL implementation optimizes display lists for any particular uses, you know that execution of display lists isn't slower than executing the commands contained within them. There is some overhead, however, involved in jumping to a display list. If a particular list is small, this overhead could exceed any execution advantage. The most likely possibilities for optimization are listed below, with references to the chapters where the topics are discussed.

- Matrix operations (**Chapter 3**). Most matrix operations require OpenGL to compute inverses. Both the computed matrix and its inverse might be stored by a particular OpenGL implementation in a display list.
- Raster bitmaps and images (**Chapter 4**). The format in which you specify raster data isn't likely to be one that's ideal for the hardware. When a display list is compiled, OpenGL might transform the data into the representation preferred by the hardware. This can have a significant effect on the speed of raster character drawing, since character strings usually consist of a series of small bitmaps.
- Lights, material properties, and lighting models (**Chapter 6**). When you draw a scene with complex lighting conditions, you might change the materials for each item in the scene. Setting the materials can be slow, since it might involve significant calculations. If you put the material definitions in display lists, these calculations don't have to be done each time you switch materials, since only the results of the calculations need to be stored; as a result, rendering lit scenes might be faster. See **"Encapsulating Mode Changes"** for more details on using display lists to change such values as lighting conditions.
- Textures (**Chapter 9**). You might be able to maximize efficiency when defining textures by compiling them into a display list, since the hardware texture format might differ from the OpenGL format, and the conversion can be done at display−list compile time rather than during display.
- Polygon stipple patterns (**Chapter 2**).

Some of the commands to specify the properties listed here are context− sensitive, so you need to take this into account to ensure optimum performance. Most situations where this makes a difference involve pixel−transfer functions, lighting models, and texturing. Since all these topics haven't been introduced yet —they're covered in later chapters —the following example is a bit contrived. Although the specifics of this example are very unlikely, it illustrates an important principle that's discussed again in later chapters.

Imagine an implementation of OpenGL that's optimized to perform matrix transformations on vertices before storing them in a display list. If this were true, the time needed to perform the transformations would occur before rather than during display. Now suppose your code looked something like this:

```
glLoadMatrix(M);
glNewList(1, GL_COMPILE);
draw_some_geometric_objects();
glEndList();
```

The vertices in the objects would be compiled into the display list after having been transformed by matrix *M*. Suppose you invoke the display list as follows:

```
glLoadMatrix(N);
glCallList(1);
```

In this case, the geometric objects should be drawn using matrix *N*, but the data in the display list has been transformed by matrix *M* before it was stored. Thus, the display list has to save two copies of the original data (both the untransformed and the transformed vertices), thereby wasting memory. In addition, the vertices undergo two transformations when perhaps one would have sufficed. If instead you had defined the display list as follows:

```
glNewList(1, GL_COMPILE);
glLoadMatrix(M);
draw_some_geometry();
glEndList();
```

then no extra data would have to be stored, and full optimization would be possible. Of course, in this second case, you'd want to be sure that matrix *M* was really the transformation matrix you wanted.

Remember that display lists have some disadvantages. The *buildCircle()* example in **Example 4−1** requires storage for at least 200 floating−point numbers, whereas the object code for the original *drawCircle()* routine (in immediate mode) is probably a lot smaller than that. Another disadvantage is the immutability of the contents of a display list. To optimize performance, an OpenGL display list can't be changed, and its contents can't be read.

Creating and Executing a Display List

As you've already seen, *glNewList()* and *glEndList()* are used to begin and end the definition of a display list, which is then invoked by supplying its identifying index with *glCallList()*. In **Example 4−2**, a display list is created in the *makeList()* routine. This display list contains OpenGL commands to draw a red triangle. Then, in the *display()* routine, the display list is executed ten times. In addition, a line is drawn in immediate mode. Note that the display list allocates memory to store the commands and the values of any necessary variables.

Example 4−2 Using a Display List: list.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

GLuint listName = 1;
```

```
void myinit (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);
}

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

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 ();
}

void myReshape(GLsizei w, GLsizei 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();
}


int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 400, 50);
    auxInitWindow (argv[0]);
```

```
     myinit ();
     auxReshapeFunc (myReshape);
     auxMainLoop(display);
}
```

The *glTranslatef()* routine in the display list alters the position of the next object to be drawn. Without it, calling the display list twice would just draw the triangle on top of itself. The *drawLine()* routine, which is called in immediate mode, is also affected by the ten *glTranslatef()* calls that precede it. Thus, if you call transformation commands within a display list, don't forget to take the effect of those commands into account later in your program.

Only one display list can be created at a time. In other words, you must eventually follow *glNewList()* with *glEndList()* to end the creation of a display list before starting another one. As you might expect, calling *glEndList()* without having started a display list generates the error GL_INVALID_OPERATION.

void *glNewList* (GLuint *list*, GLenum *mode*);

Specifies the start of a display list. OpenGL routines that are called subsequently (until *glEndList()* is called to end the display list) are stored in a display list, except for a few restricted OpenGL routines that can't be stored. (Those restricted routines are executed immediately, during the creation of the display list.) The *list* parameter is a unique positive integer that identifies the display list. The possible values for the *mode* parameter are GL_COMPILE and GL_COMPILE_AND_EXECUTE. Use GL_COMPILE if you don't want the following OpenGL commands executed as they're placed in the display list; to cause the commands to be executed immediately as well as placed in the display list for later use, specify GL_COMPILE_AND_EXECUTE.

void *glEndList* (void);

Marks the end of a display list.

What's Stored in a Display List

When you're building a display list, only the values for expressions are stored in the list. Thus, if values in an array are subsequently changed, for example, the display−list values don't change. In the following code fragment, the display list contains a command to set the current color to black (0.0, 0.0, 0.0). The subsequent change of the value of the *color_vector* array to red (1.0, 0.0, 0.0) has no effect on the display list because the display list contains the values that were in effect when it was created.

```
GLfloat color_vector[3]={0.0,0.0,0.0};
glNewList(1,GL_COMPILE);
   glColor3fv(color_vector);
glEndList();
color_vector[0]=1.0;
```

Not all OpenGL commands can be stored and executed from within a display list. Generally, commands that pass parameters by reference or that return a value can't be stored in a display list, since the list might be called outside the scope of where the parameters are originally defined. If such commands are called when making a display list, they're executed immediately and aren't stored in the display list. Here are the OpenGL commands that aren't stored in a display list (also, note that *glNewList()* generates an error if it's called while you're creating a display list). Some of these commands haven't been described yet; you can look in the index to see where they're discussed.

glDeleteLists()	glIsEnabled()
glFeedbackBuffer()	glIsList()
glFinish()	glPixelStore()

glFlush() glReadPixels()

glGenLists() glRenderMode()

glGet*() glSelectBuffer()

To understand more clearly why these commands can't be stored in a display list, remember that when you're using OpenGL across a network, the client may be on one machine and the server on another. After a display list is created, it resides with the server, so the server can't rely on the client for any information related to the display list. If querying commands, such as *glGet*()* or *glIs*()*, were allowed in a display list, the calling program would be surprised at random times by data returned over the network. Without parsing the display list as it was sent, the calling program wouldn't know where to put the data. Thus, any command that returns a value can't be stored in a display list. Other routines —such as *glFlush()* and *glFinish()* —can't be stored in a display list because they depend on information about the client state. Finally, commands that change a state value maintained by the client can't be stored in a display list.

Executing a Display List

After you've created a display list, you can execute it by calling *glCallList()*. Naturally, you can execute the same display list many times, and you can mix calls to execute display lists with calls to perform immediate−mode graphics, as you've already seen.

void *glCallList* (GLuint *list*);

This routine executes the display list specified by *list*. The commands in the display list are executed in the order they were saved, just as if they were issued without using a display list. If *list* hasn't been defined, nothing happens.

Since a display list can contain calls that change the value of OpenGL state variables, these values change as the display list is executed, just as if the commands were called in immediate mode. The changes to OpenGL state persist after execution of the display list is completed. In **Example 4−2**, the changes to the current color and current matrix made during the execution of the display list remain in effect after it's been called, as shown in **Example 4−3**.

Example 4−3 Persistence of State Changes after Execution of a Display List

```
glNewList(listIndex,GL_COMPILE);
   glColor3f(1.0, 0.0, 0.0);
   glBegin(GL_POLYGON);
      glVertex2f(0.0,0.0);
      glVertex2f(1.0,0.0);
      glVertex2f(0.0,1.0);
   glEnd();
   glTranslatef(1.5,0.0,0.0);
glEndList();
```

Sometimes you want state changes to persist, but other times you want to save the values of state variables before executing a display list and then restore these values after the list has executed. Use *glPushAttrib()* to save a group of state variables and *glPopAttrib()* to restore the values when you're ready for them. (See **Appendix B** for more information about these commands.) To save and restore the current matrix, use *glPushMatrix()* and *glPopMatrix()* as described in **"Manipulating the Matrix Stacks."** To restore the state variables in **Example 4−3**, you might use the code shown in **Example 4−4**.

Example 4−4 Restoring State Variables within a Display List

```
glNewList(listIndex,GL_COMPILE);
```

```
   glPushMatrix();
   glPushAttrib(GL_CURRENT_BIT);
      glColor3f(1.0, 0.0, 0.0);
      glBegin(GL_POLYGON);
         glVertex2f(0.0,0.0);
         glVertex2f(1.0,0.0);
         glVertex2f(0.0,1.0);
      glEnd();
   glTranslatef(1.5,0.0,0.0);
   glPopAttrib();
   glPopMatrix();
glEndList();
```

Thus, if you used this kind of a display list that restores values, the code in **Example 4−5** would draw a green, untranslated line. With the display list in **Example 4−3** that doesn't save and restore values, the line drawn would be red, and its position would be translated.

Example 4−5 Using a Display List That Restores State Variables

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

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

You can call *glCallList()* from anywhere within a program, as long as its OpenGL context is still active. A display list can be created in one routine and executed in a different one, since its index uniquely identifies it. Also, there is no facility to save the contents of a display list into a data file, nor a facility to create a display list from a file. In this sense, a display list is designed for temporary use. Also, a display list is destroyed when its OpenGL context is destroyed.

Hierarchical Display Lists

You can create a *hierarchical display list*, which is a display list that executes another display list, by calling *glCallList()* between a *glNewList()* and *glEndList()* pair. A hierarchical display list is useful for an object that's made of components, especially if some of those components are used more than once. For example, this is a display list that renders a bicycle by calling other display lists to render parts of the bicycle:

```
glNewList(listIndex,GL_COMPILE);
   glCallList(handlebars);
   glCallList(frame);
   glTranslatef(1.0,0.0,0.0);
   glCallList(wheel);
   glTranslatef(3.0,0.0,0.0);
   glCallList(wheel);
glEndList();
```

To avoid infinite recursion, there's a limit on the nesting level of display lists; the limit is at least 64, but it might be higher, depending on the implementation. To determine the nesting limit for your implementation of OpenGL, call

```
glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data);
```

OpenGL allows you to create a display list that calls another list that hasn't been created yet. Nothing happens when the first list calls the second, undefined one.

You can use a hierarchical display list to approximate an editable display list by wrapping a list around several lower−level lists. For example, to put a polygon in a display list while allowing yourself to be able to easily edit its vertices, you could use the code in **Example 4−6** .

Example 4−6 Using a Hierarchical Display List

```
glNewList(1,GL_COMPILE);
   glVertex3f(v1);
glEndList();
glNewList(2,GL_COMPILE);
   glVertex3f(v2);
glEndList();
glNewList(3,GL_COMPILE);
   glVertex3f(v3);
glEndList();

glNewList(4,GL_COMPILE);
   glBegin(GL_POLYGON);
      glCallList(1);
      glCallList(2);
      glCallList(3);
   glEnd();
glEndList();
```

To render the polygon, call display list number 4. To edit a vertex, you need only recreate the single display list corresponding to that vertex. Since an index number uniquely identifies a display list, creating one with the same index as an existing one automatically deletes the old one. Keep in mind that this technique doesn't necessarily provide optimal memory usage or peak performance, but it's acceptable and useful in some cases.

Managing Display Lists and Their Indices

So far, we've used an arbitrary positive integer as a display−list index. This could be dangerous in practice because you might accidentally choose an index that's already in use, thereby overwriting an existing display list. To avoid accidental deletions, use *glGenLists()* to generate an unused index and *glIsList()* to determine whether a specific index is in use. You can explicitly delete a specific display list or a range of lists with *glDeleteLists()* .

GLuint *glGenLists* (GLsizei *range*);

Allocates *range* number of contiguous, previously unallocated display−list indices. The integer returned is the index that marks the beginning of a contiguous block of empty display−list indices. The returned indices are all marked as empty and used, so subsequent calls to *glGenLists()* don't return these indices until they're deleted. Zero is returned if the requested number of indices isn't available, or if *range* is zero.

GLboolean *glIsList* (GLuint *list*);

Returns TRUE if *list* is already used for a display list and FALSE otherwise.

In the following example, a single index is requested, and if it proves to be available, it's used to create a new display list:

```
listIndex=glGenLists(1);
if(listIndex!=0) {
   glNewList(listIndex,GL_COMPILE);
      ...
   glEndList();
}
```

The command *glDeleteLists()* deletes a contiguous group of display lists, thereby making their indices available again.

---

void *glDeleteLists* (GLuint *list*, GLsizei *range*);

Deletes *range* display lists, starting at the index specified by *list*. An attempt to delete a list that has never been created is ignored.

Executing Multiple Display Lists

OpenGL provides an efficient mechanism to execute several display lists in succession. This mechanism requires that you put the display−list indices in an array and call *glCallLists()*. An obvious use for such a mechanism occurs when display−list indices correspond to meaningful values. For example, if you're creating a font, each display−list index might correspond to the ASCII value of a character in that font. To have several such fonts, you would need to establish a different initial display−list index for each font. You can specify this initial index by using *glListBase()* before calling *glCallLists()*.

---

void *glListBase* (GLuint *base*);

Specifies the offset that's added to the display−list indices in *glCallLists()* to obtain the final display−list indices. The default display−list base is 0. The list base has no effect on *glCallList()*, which executes only one display list, or on *glNewList()*.

---

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

Executes *n* display lists. The indices of the lists to be executed are computed by adding the offset indicated by the current display−list base (specified with *glListBase()*) to the signed integer values in the array pointed to by *lists*.

The *type* parameter indicates the data type and the "stride" (or size) of each element in the array of indices. It's usually one of these constants: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, or GL_FLOAT. It can also be GL_2_BYTES, GL_3_BYTES, or GL_4_BYTES, in which case sequences of two, three, or four bytes are shifted and added together, byte by byte, to calculate the display−list offset, using this algorithm:

```
/* b = 2, 3, or 4; bytes are numbered 0, 1, 2, 3 in array */
 offset = 0;
 for (i = 0; i < b; i++) {
 offset = offset << 8;
 offset += byte[i];
 }
index = offset + listbase;
```

This means that for multiple−byte data, as bytes are taken from the array in order, the highest−order data comes first.

As an example of the use of multiple display lists, look at the program fragments in **Example 4−7** taken from the full program in **Example 4−8** . This program draws characters with a stroked font (a set of letters made from line segments). The routine *initStrokedFont()* sets up the display−list indices for each letter so they correspond with their ASCII values.

Example 4−7 Defining Multiple Display Lists

```
void initStrokedFont(void)
{
    GLuint base;

    base = glGenLists (128);
    glListBase(base);
    glNewList(base+'A', GL_COMPILE);
       drawLetter(Adata); glEndList();
    glNewList(base+'E', GL_COMPILE);
       drawLetter(Edata); glEndList();
    glNewList(base+'P', GL_COMPILE);
       drawLetter(Pdata); glEndList();
    glNewList(base+'R', GL_COMPILE);
       drawLetter(Rdata); glEndList();
    glNewList(base+'S', GL_COMPILE);
       drawLetter(Sdata); glEndList();
    glNewList(base+' ', GL_COMPILE);    /* space character */
       glTranslatef(8.0, 0.0, 0.0); glEndList();
}
```

The *glGenLists()* command allocates 128 contiguous display−list indices. The first of the contiguous indices becomes the display−list base. A display list is made for each letter; each display−list index is the sum of the base and the ASCII value of that letter. In this example, only a few letters and the space character are created.

After the display lists have been created, *glCallLists()* can be called to execute the display lists. For example, you can pass a character string to the subroutine *printStrokedString()* :

```
void printStrokedString(GLbyte *s)
{
    GLint len = strlen(s);
    glCallLists(len, GL_BYTE, s);
}
```

The ASCII value for each letter in the string is used as the offset into the display−list indices. The current list base is added to the ASCII value of each letter to determine the final display−list index to be executed. The output produced by **Example 4−8** is shown in **Figure 4−1** .

Figure 4−1 Example of a Stroked Font That Defines the Characters A, E, P, R, S

Example 4–8 Using Multiple Display Lists to Define a Stroked Font: stroke.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

#define PT 1
#define STROKE 2
#define END 3

typedef struct charpoint {
    GLfloat   x, y;
    int    type;
} CP;

CP Adata[] = {
    { 0, 0, PT}, {0, 9, PT}, {1, 10, PT}, {4, 10, PT},
    {5, 9, PT}, {5, 0, STROKE}, {0, 5, PT}, {5, 5, END}
};

CP Edata[] = {
    {5, 0, PT}, {0, 0, PT}, {0, 10, PT}, {5, 10, STROKE},
    {0, 5, PT}, {4, 5, END}
};

CP Pdata[] = {
    {0, 0, PT}, {0, 10, PT},  {4, 10, PT}, {5, 9, PT},
    {5, 6, PT}, {4, 5, PT}, {0, 5, END}
};

CP Rdata[] = {
    {0, 0, PT}, {0, 10, PT},  {4, 10, PT}, {5, 9, PT},
    {5, 6, PT}, {4, 5, PT}, {0, 5, STROKE}, {3, 5, PT},
    {5, 0, END}
};

CP Sdata[] = {
    {0, 1, PT}, {1, 0, PT}, {4, 0, PT}, {5, 1, PT}, {5, 4, PT},
    {4, 5, PT}, {1, 5, PT}, {0, 6, PT}, {0, 9, PT}, {1, 10, PT},
    {4, 10, PT}, {5, 9, END}
};

void drawLetter(CP *l)
{
    glBegin(GL_LINE_STRIP);
    while (1) {
        switch (l->type) {
            case PT:
                glVertex2fv(&l->x);
                break;
            case STROKE:
                glVertex2fv(&l->x);
                glEnd();
                glBegin(GL_LINE_STRIP);
```

```
                break;
            case END:
                glVertex2fv(&l->x);
                glEnd();
                glTranslatef(8.0, 0.0, 0.0);
            return;
        }
    l++;
    }
}

void myinit (void)
{
    GLuint base;

    glShadeModel (GL_FLAT);

    base = glGenLists (128);
    glListBase(base);
    glNewList(base+'A', GL_COMPILE); drawLetter(Adata);
        glEndList();
    glNewList(base+'E', GL_COMPILE); drawLetter(Edata);
        glEndList();
    glNewList(base+'P', GL_COMPILE); drawLetter(Pdata);
        glEndList();
    glNewList(base+'R', GL_COMPILE); drawLetter(Rdata);
        glEndList();
    glNewList(base+'S', GL_COMPILE); drawLetter(Sdata);
        glEndList();
    glNewList(base+' `, GL_COMPILE);
        glTranslatef(8.0, 0.0, 0.0); glEndList();
}

char *test1 = "A SPARE SERAPE APPEARS AS";
char *test2 = "APES PREPARE RARE PEPPERS";

void printStrokedString(char *s)
{
    GLsizei len = strlen(s);
    glCallLists(len, GL_BYTE, (GLbyte *)s);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0, 30.0, 0.0);
    printStrokedString(test1);
    glPopMatrix();
    glPushMatrix();
    glScalef(2.0, 2.0, 2.0);
    glTranslatef(10.0, 13.0, 0.0);
```

```
    printStrokedString(test2);
    glPopMatrix();
    glFlush();
}


int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 440, 120);
    auxInitWindow (argv[0]);
    myinit ();
    auxMainLoop(display);
}
```

Encapsulating Mode Changes

You can use display lists to organize and store groups of commands to change various modes or set various parameters. When you want to switch from one group of settings to another, using display lists might be more efficient than making the calls directly, since the settings might be cached in a format that matches the requirements of your graphics system.

Display lists are likely to be more efficient when you're switching between multiple texture maps, for example. (Texture mapping is described in **Chapter 9**.) Suppose you have two different textures that are fairly large, as textures tend to be, but that both of them fit into texture memory. Without display lists, you would have to load the data for the first texture, use it to draw some objects, wait for the second texture to be loaded into memory, and then use it for drawing. When you want to switch back to the first texture, it would have to be loaded into memory again rather than being plucked out of texture memory. There's no way for OpenGL to know that it's already been stored without the display−list mechanism to provide a "handle" to identify it. With display lists, both textures can be loaded into texture memory once and then used as often as necessary without having to be reloaded.

Another case where display lists are likely to be more efficient than immediate mode is for switching among various lighting, lighting−model, and material−parameter settings. (These topics are discussed in **Chapter 6**.) You might also use display lists for stipple patterns, fog parameters, and clipping−plane equations. In general, you're guaranteed that executing display lists is at least as fast as making the relevant calls directly, but remember that some overhead is involved in jumping to a display list.

Example 4−9 shows how to use display lists to switch among three different line stipples. First, you call *glGenLists()* to allocate a display list for each stipple pattern and create a display list for each pattern. Then, you use *glCallList()* to switch from one stipple pattern to another.

Example 4−9 Using Display Lists for Mode Changes

```
GLuint offset;
offset = glGenLists (3);

glNewList (offset, GL_COMPILE);
    glDisable (GL_LINE_STIPPLE);
glEndList ();

glNewList (offset+1, GL_COMPILE);
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x0F0F);
```

```
glEndList ();

glNewList (offset+2, GL_COMPILE);
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x1111);
glEndList ();


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

glCallList (offset);
drawOneLine (50.0, 125.0, 350.0, 125.0);

glCallList (offset+1);
drawOneLine (50.0, 100.0, 350.0, 100.0);

glCallList (offset+2);
drawOneLine (50.0, 75.0, 350.0, 75.0);
```