Chapter 1

The First Steps

In an ideal world only a few lines of code would produce graphics right now. Alas, in the real world, some preliminary housekeeping is necessary. In this chapter you will learn how to make the necessary connection between Windows and OpenGL. The plan is to produce some graphics as soon as possible, so be patient with the necessary set up. In the spirit of getting to the interesting code as soon as possible, this chapter is fairly short and ends with actual graphics output. Although it is just a simple colored background, it shows that the code setting it up successfully did its job. Though trivial, this simple scene is not just a token reward for enduring a necessary evil. The code for producing it forms the basis for more advanced output.

DIRECTORY STRUCTURE

To use this book effectively you will be writing code along the way and will create and save a number of projects, so pick a drive for keeping these OpenGL projects. Create a directory and name it OpenGL. Under that directory create subdirectories named Chapter.1, Chapter.2, etc. Some chapters may produce several projects, and these each have their own subdirectories under the chapter directory. You can, of course, organize in some other way, but the book proceeds as if you use the recommended directory structure and names for your projects, so you must translate as you go.

PRELIMINARY CODE

Getting Ready to Start to Begin

OpenGL is intended to be fairly platform-independent, rather than just for Windows. Therefore OpenGL needs a link to Windows, using some special structures and API extensions in Windows to provide this link. In Delphi a good place for the connection is within a form's OnCreate event handler.

Create a new project and save it in a new directory under the OpenGL directory created earlier. Name the new directory “Chapter.1,” name the project “First.Dpr,” and name the main unit “First1.Pas”. Double-click the main form to set up the form's OnCreate event handler. Define a variable of type TPixelFormatDescriptor and fill it in. Defining the pixel format with this structure permits describing some properties

that the Windows GDI (graphics device interface) needs in order to work with OpenGL.

```
procedure TForm1.FormCreate(Sender: TObject);
var
   pfd: TPixelFormatDescriptor;
   FormatIndex: integer;
begin
   fillchar(pfd,SizeOf(pfd),0);
   with pfd do
   begin
     nSize      := SizeOf(pfd);
     nVersion   := 1; {The current version of the desccriptor is 1}
     dwFlags    := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL;
     iPixelType := PFD_TYPE_RGBA;
     cColorBits := 24; {support 24-bit color}
     cDepthBits := 32; {depth of z-axis}
     iLayerType := PFD_MAIN_PLANE;
   end; {with}
   FormatIndex := ChoosePixelFormat(Canvas.Handle,@pfd);
end; {FormCreate}
```

Inform the system of the desired properties by calling ChoosePixelFormat, passing it form's device context and a pointer to the descriptor. Depending on what the device context can support, the contents of the descriptor may be altered to reflect the best approximation of the request. Windows uses Device Contexts, and OpenGL uses Rendering Contexts. This code attempts to map them to each other.

DEVICE CONTEXT The use of a device context is not obvious in this code. In Delphi, the Canvas property is a wrapper for a window's device context, and Canvas.Handle is the handle to the device context. This code uses native Delphi where possible, leaving out error checking and other details for clarity. Be patient for a while; the code improves later.

DESCRIPTER FIELDS This descriptor has a number of fields that can remain zero or the equivalent, so explicit assignment statements were unnecessary. Here is a quick look at the rest of the contents of the descriptor to meet the current needs:

1.nSize. Windows structures often require the size of the structure as part of

the structure itself. This field follows that tradition.

2.nVersion. The version number of the descriptor structure is 1, so a 1 must be stored here.

3.dwFlags. The bits are set by or-ing together some pre-defined constants. PFD_DRAW_TO_WINDOW has an obvious meaning; you could be drawing to a bitmap in memory instead. PFD_SUPPORT_OPENGL is certainly a desired feature. Keep in mind that this code does not yet do anything with OpenGL; it just gets Windows ready for OpenGL. These are Windows structures and Windows API calls. Windows does not assume the code will work with OpenGL unless the code tells it. ChoosePixelFormat attempts to find a pixel format with the same flags set as those passed to it in the descriptor.

4.iPixelType. Use RGBA (red, green, blue, alpha) pixels. Explanation of Alpha comes later.

5.cColorBits. Support 24-bit color.

6.cDepthBits. Set the depth of the z-axis to 32. Explanation of depth comes later.

7.iLayerType. The current version only supports the main plane.

ChoosePixelFormat is a function that returns an integer, stored in FormatIndex. It returns zero to indicate an error, or a positive number as an index to the appropriate pixel format. SetPixelFormat sets the pixel format of the device context, using that index. Now the bottom of the OnCreate event handler looks like this:

```
   FormatIndex := ChoosePixelFormat(Canvas.Handle,@pfd);
   SetPixelFormat(Canvas.Handle,FormatIndex,@pfd);
end; {FormCreate}
```

The functions receive the handle to the window's device context and change the pixel format of the window. So far none of the code produces visible results other than a standard blank form, but keep coding. OpenGL will shine forth soon.

Starting to Begin

RENDERING CONTEXT Now that the code takes care of the pixel format, proceed to the rendering context. Add GLContext to the private section of the form:

```
type
```

```
   TForm1 = class (TForm)
     procedure FormCreate(Sender: TObject);
private
     GLContext : HGLRC;
public
     { Public declarations }
end;
```

The HGLRC type is a handle to an OpenGL Rendering Context type. Here is its declaration for the interface unit:

```
type
   HGLRC = THandle;
```

The new variable receives the result of wglCreateContext. This is one of several wgl (Windows-GL) functions for managing rendering contexts. Place this call at the bottom of the event handler:

```
   FormatIndex := ChoosePixelFormat(Canvas.Handle,@pfd);
   SetPixelFormat(Canvas.Handle,FormatIndex,@pfd);
   GLContext := wglCreateContext(Canvas.Handle);
end; {FormCreate}
```

As the name implies, this function creates the OpenGL rendering context needed by the window. Now make it current:

```
   FormatIndex := ChoosePixelFormat(Canvas.Handle,@pfd);
   SetPixelFormat(Canvas.Handle,FormatIndex,@pfd);
   GLContext := wglCreateContext(Canvas.Handle);
   wglMakeCurrent(Canvas.Handle,GLContext);
end; {FormCreate}
```

CLEAN UP You should always put away your tools when finished with them, and you should always put away windows resources when finished with them. Go to the events page of the object inspector for the form. Double-click the OnDestroy event. Fill in the event handler as follows:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
   wglMakeCurrent(Canvas.Handle,0);
   wglDeleteContext(GLContext);
end;
```

Passing a zero to wglMakeCurrent makes the previously current context no longer current. Now that GLContext is no longer current, delete it with wglDeleteContext.

Begin

USES CLAUSE So far the code is nothing but Windows and Delphi code. The time has arrived for some OpenGL code. Add OpenGL to a **uses** clause. Put it in the interface section in order to use an OpenGL type in the interface.

```
uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
   Dialogs, OpenGL, Menus;
```

PAINT Again go to the events page of the form's object inspector. This time double-click the OnPaint event. This event fires every time the window (form) needs repainting, such as when the form has been partially (or fully) covered and then exposed again. It also fires when the form first shows. Make this event handler like the following:

```
procedure  TForm1.FormPaint(Sender: TObject);
begin
   {background}
   glClearColor(0.0,0.4,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);

   {error checking}
   errorCode := glGetError;
   if  errorCode<>GL_NO_ERROR then
      raise  Exception.Create('Error in Paint'#13+
         gluErrorString(errorCode));
end;
```

The given OpenGL commands have the following declaration:

```
procedure glClear(mask:GLbitfield); stdcall;
procedure glClearColor(red,green,blue,alpha:GLclampf); stdcall;
```

Their implementations are simply references to the DLL which contains the code and look like this:

```
procedure glClear; external opengl32;
procedure glClearColor; external opengl32;
```

ERROR CHECKING Notice this code introduces a little bit of error checking. This method is small enough (so far) that including the error checking code at this point does not obscure the main activity.

Use glClearColor to set the background color. Each color component ranges in floating point value from zero to one. In this case the color has no red, 0.4 units of green, and no blue. Set alpha to zero for now. The background color is a dark green (a little less than half the maximum intensity of green, which would be 1.0).

This is a good time to introduce an OpenGL type. Much of this graphics library, including color specifications, uses floating point numbers. To keep itself portable, OpenGL defines a type called glFloat. Under Windows on Intel processors and compatibles, glFloat is the same as a Delphi single, which is an IEEE (Institute for Electrical and Electronics Engineers) 32 bit floating point value. In the interface unit is a type declaration like this:

```
glFloat = single;
```

Next call glClear and pass it a bit mask that tells it what to clear. This clears the buffers enabled for writing colors. For error checking this segment of code introduces the function glGetError, which appears often. It returns a number of type GLenum, which is an alias for Cardinal. The following type declaration appear in the interface unit:

```
GLenum = Cardinal;
```

Store the result in errorCode, which you should add to the private section of the form declaration.

```
ErrorCode: GLenum;
```

Compare the result to GL_NO_ERROR, a pre-defined constant of obvious meaning. If there is an error, raise an exception. The application handles this exception simply by displaying the message. The message includes another new function, gluErrorString, from the OpenGL utility library. It returns a pointer to some text (PChar). The text represents the meaning of the error number passed to the function. Delphi's string concatenation knows how to handle that PChar.

The constant has this declaration:

```
  GL_NO_ERROR = 0;
```

Here is the declaration for the function:

```
function gluErrorString(errCode:GLenum):PChar; stdcall;
```

Since this function name begins with “glu,” its code is found in the other DLL:

```
function gluErrorString; external glu32;
```

Now, save the program (you have been warned!) and compile it. Do not run it from the Delphi IDE (Integrated Development Environment). OpenGL programs sometimes crash if launched from the IDE even though they work fine when run independently. That is why it was so important to save your work before starting, just in case you got stubborn and ran it from the IDE anyway. You might be lucky, but you might not. Use the Windows Run command, giving it the appropriate path. Behold! A dark green form!

MORE DEVICE CONTEXT Some of the code just written needs improving. The time has arrived to demonstrate why. Drag the form by its title bar left and right and up and down, so that parts are obscured and re-exposed repeatedly. Partially cover it with some other window, then uncover it. Do this a number of times until something strange happens. Eventually some part of the form fails to show green after exposure.

While Canvas.Handle is the handle to the device context, it is not reliable for these purposes, because it is generated on the fly, when referenced, then released. So the device context matched up with the rendering context at one point may not even exist at another point. The rendering context would not then have a valid association. The application needs a handle it can control. In the private section of the form define glDC:

```
type
   TForm1 = class (TForm)
      procedure FormCreate(Sender: TObject);
      procedure FormDestroy(Sender: TObject);
      procedure FormPaint(Sender: TObject);
   private
      GLContext: HGLRC;
      glDC: HDC;
      errorCode: GLenum;
   public
      { Public declarations }
   end ;
```

In the FormCreate method (OnCreate handler), call getDC and replace all references to Canvas.Handle with glDC:

```
   glDC := getDC(handle);
   FormatIndex := ChoosePixelFormat(glDC,@pfd);
   SetPixelFormat(glDC,FormatIndex,@pfd);
   GLContext := wglCreateContext(glDC);
   wglMakeCurrent(glDC,GLContext);
end; {FormCreate}
```

Save the code again, compile it, then run it (independently). Now it works much more reliably.

MORE ERROR HANDLING In the FormCreate method is the basic setup required to use OpenGL under Windows. Now that the code has appeared in its simplicity, it is time to complicate it with error checking. It is not exciting, but all good software requires some kind of error handling or reporting. With these learning programs, if something fails, at least some kind of clue should appear. Add a boolean to the form's private declaration. As a field of a TObject descendant it is initialized to false. If you are not familiar with this initialization, see Delphi's online help for the InitInstance method under TObject.

```
private
    GLContext: HGLRC;
    glDC: HDC;
    errorCode: GLenum;
    openGLReady: boolean;
```

In FormPaint test openGLReady because there is no point in calling OpenGL commands if they were not even set up.

```
procedure  TForm1.FormPaint(Sender: TObject);
begin
   if not openGLReady then
      exit;
   {background}
   glClearColor(0.0,0.4,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT);

   {error checking}
   error := glGetError;
   if  error<>GL_NO_ERROR then
      raise  Exception.Create('Error in Paint'#13+
         gluErrorString(errorCode));
end;
```

FormCreate also needs to test some function results. On failure call GetLastError which returns a Windows error code. If execution make it all the way to the end of the method, the reward is setting OpenGLReady to true.

```
   glDC := getDC(handle);
   FormatIndex := ChoosePixelFormat(glDC,@pfd);
   if FormatIndex=0 then
      raise Exception.Create('ChoosePixelFormat failed '+
          IntToStr(GetLastError));

   if not SetPixelFormat(glDC,FormatIndex,@pfd)   then
       raise Exception.Create('SetPixelFormat failed '+
          IntToStr(GetLastError));

GLContext := wglCreateContext(glDC);
```

```
 if  GLContext=0 then
   raise  Exception.Create('wglCreateContext failed '+
      IntToStr(GetLastError)');

 if not wglMakeCurrent(glDC,GLContext) then
     raise  Exception.Create('wglMakeCurrent failed '+
         IntToStr(GetLastError)');
 OpenGLReady := true;
end; {FormCreate}
```

Be sure to save your work. The next chapter starts with this code as the foundation.

DEFINITIONS

Term	Meaning
24-bit color	On typical PC video hardware colors have three components: red, green, and blue. Each color component can range in integer value from 0 (none) to 255 (maximum intensity), allowing an 8-bit (1-byte) integer to specify each intensity. You can specify 256*256*256 color combinations. These 16,777,216 possible colors are more than the human eye can distinguish. In a True Color or 24-bit color mode you can specify the color components directly, using at least a 24-bit number, but with a lesser "color depth" (number of bits to specify a color), the number you use to describe a color is just an index into a "palette" or pre-defined array of colors.
Background color	The color that shows through in a window when nothing else is drawn at a particular location. When a window is cleared, it is filled with the appropriate background color.
Current context	The active (currently being used) rendering context.
Device Context	A set of information about Windows drawing modes and commands.
GDI	Graphics Device Interface. A standardized way of specifying graphics operations under Windows independently of the hardware.
Pixel	Picture element. The smallest unit of drawing. A dot.
Pixel format	Those characteristics of a device context needed for setting up a rendering context.
Rendering Context	A set of information about OpenGL states and commands under Windows.
RGBA	Red, Green, Blue, Alpha. OpenGL's method of specifying color. Each of the three color

components are floating point values that range from 0.0 (none) to 1.0 (maximum intensity). The combination of the three “pure” colors produces the color intended for the viewer. The alpha component is reserved for a later chapter.

IDENTIFIER REFERENCE

Identifier	Description
ChoosePixelFormat	A Windows function that receives a handle to a device context and a pointer to a pixel format descriptor. It returns an integer index to Windows’ best attempt at a pixel format matching the specifications in the descriptor.
getDC	A Windows function that receives a handle to a window and returns a handle to that window’s device context.
GetLastError	A Windows function that returns an integer representing the most recent error.
glClear	An OpenGL command that erases a buffer, filling it with the most recently specified background color. If the buffer is the color buffer, then the window is filled with the background color.
glClearColor	An OpenGL command that receives red, blue, green, and alpha floating point values sets the background color to be used by glClear.
glEnum	An OpenGL numeric type that maps to Delphi cardinal;
glFloat	An OpenGL numeric type that maps to Delphi single.
glGetError	An OpenGL function that returns an integer value representing an error flag set by an OpenGL command, and clears that flag. If no error flags are set when called, the function returns the value of GL_NO_ERROR.
gluErrorString	An OpenGL utility function that receives and error number (such as returned by glGetError) and returns pointer to a human-readable string (null-terminated) describing the associated OpenGL error flag.
GL_NO_ERROR	An OpenGL constant to represent a successful command.

HDC	A Windows type. Handle to a device context.
HGLRC	A Windows/OpenGL type. Handle to a renedering context.
SetPixelFormat	A Windows function that specifies the pixel format to use with the given device context. While ChoosePixelFormat merely obtains an index to a pixel format, this function sets that pixel format as the one to use. The function receives a handle to the device context, an index to the desired pixel format (such as the one returned by ChoosePixelFormat) and a pointer to the pixel format descriptor (as modified by ChoosePixelFormat). It returns true on success or false on failure. Call GetLastError to learn more about the failure.
TPixelFormatDescriptor	A Windows type. A structure of this type contains the fields needed to select and set a pixel format.
wglCreateContext	A Windows function that creates a rendering context for the specified device context. It receives a handle to the device context and returns a handle to the rendering context.
wglDeleteContext	A Windows function that deletes the specified rendering context. It received the handle to the rendering context and returns true on success and false on failure.
wglMakeCurrent	A Windows function that makes the specified rendering context current (the one available for use) with the specified device context. It receives the handle to the device context and the handle to the rendering context and returns true on success and false on failure.

SUMMARY

Chapter 1 showed how to:

1. Link the Windows graphics device context with the OpenGL rendering context.
2. Use actual OpenGL commands to fill a window with a color.
3. Perform some basic error checking with OpenGL commands.

Chapter 2

Basics

OpenGL majors in three-dimensional graphics, but computer monitors operate with essentially flat, two-dimensional screens. Representing a three-dimensional space on a two-dimensional surface requires *projection*. Each point in the space projects mathematically to an appropriate point on the surface. This chapter introduces OpenGL commands to identify the part of the window to receive the projection, to set up the projection, and to draw an object. Included with the projection is a *clipping* volume, outside of which no drawing happens. These foundational concepts and commands permit drawing something besides a mere background. This chapter also brings another Delphi event handler into use.

Use the first chapter's final code as the starting point in this chapter. If you have not already done so, create a new directory under the "OPENGL" directory and name it "Chapter.2." In Delphi load the First.Dpr project and do a File|Save Project As in the new directory, naming it "Rect.Dpr." Do File|Save As for the "First1.Pas" unit. Name it "Rect1.Pas" in the new directory.

VIEW PORT

Event Handler

Drawing a green background is a wonderful accomplishment, but by itself, a green background is not very exciting. Drawing a shape ON the background is much more interesting. For now, drawing a rectangle is the next great advance in OpenGL knowledge. The place for drawing is the form's OnPaint event handler, but setting up the drawing area also needs the OnResize handler. Go to the events page of the form's object inspector and double-click OnResize. In the event handler place a call to glViewPort and a call to glOrtho, and a modicum of error checking:

```
procedure TForm1.FormResize(Sender: TObject);
begin
   if not openGLReady then
      exit;
   glViewPort(0,0,ClientWidth,ClientHeight);
   glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
   errorCode := glGetError;
```

```
   if errorCode<>GL_NO_ERROR then
      raise Exception.Create('FormResize:'+gluErrorString(errorCode));
end
```

Declaration

Remember that OpenGL is fairly platform independent, so it does not inherently know anything about the coordinate system of the operating environment. Use glViewPort to make a connection between the coordinates used by the windowing system (Microsoft Windows) and the coordinates used by OpenGL. Its declaration looks like this:

```
procedure glViewport(x,y:GLint; width,height:GLsizei); stdcall;
```

GLint and GLsizei both map to integer. The meaning of width and height are as expected, but x and y need more attention. Windows measures from the upper left corner of a region, but OpenGL measures from the lower left, as if it were relating to the first quadrant of a set of Cartesian coordinates. The main thing to note is that the y-direction runs opposite to the familiar Windows coordinates, but fits more naturally with coordinates commonly used in mathematics.

If you are building your own interface unit, the Listing 2-1 encompasses the declarations used thus far, plus a few more that are just waiting in the wings. Add other declarations when they appear.

Listing 2-1. Beginning of an Interface Unit.

```
unit OpenGL;

interface

uses
   Windows;

const
   GL_COLOR_BUFFER_BIT = $4000;
   GL_NO_ERROR = 0;
Type
   HGLRC = THandle;
   GLenum = cardinal;
   GLbitfield = cardinal;
```

```
   GLfloat = single;
   GLclampf = single;
   GLint = integer;
   GLsizei = integer;

{regular commands}
procedure glClear(mask:GLbitfield); stdcall;
procedure glClearColor(red,green,blue,alpha:GLclampf); stdcall;
function  glGetError: GLenum; stdcall;
procedure glViewport(x,y:GLint; width,height: GLsizei); stdcall;

{utility commands}
function  gluErrorString(errCode:GLenum):PChar; stdcall;

implementation

{regular commands}
procedure glClear; external opengl32;
procedure glClearColor; external opengl32;
function  glGetError; external opengl32;
procedure glViewport; external opengl32;

{utility commands}
function  gluErrorString; external glu32;

initialization
   Set8087CW($133F);
end.
```

Analogy

Think of the form as a wall. With glViewPort cut a hole in the wall through which to view a scene. In this case, cut a hole the size of the entire client area of the form, so all that is left is the frame. A smaller "opening" is possible:

```
   glViewPort(50,30,80,70);
```

The above example starts at 50 pixels from the left side of the form and 30 pixels up from the bottom of the form. It is 80 pixels wide and 70 pixels tall (going up from the bottom).

It may not be a good idea to specify such absolute coordinates without specifying a non-resizable border or limiting the minimum window size.

One side note: glViewPort limits the portion of the window in which the scene is *rendered*, but does not restrict the area in which the background is cleared. In the wall analogy think of the background color as something sprayed onto the stage for the scene. Not only does the spray go through the hole in the wall, but the spray also covers the entire wall! There are remedies for this problem, but that subject comes later.

In simplest terms rendering is drawing with OpenGL. To render a scene is to use OpenGL constructs and commands to cause pixels to illuminate in a pattern that represents the scene envisioned by the programmer.

ORTHOGRAPHIC PROJECTION

Meaning

The call to glOrtho defines a clipping volume, which is a region in space in which objects can be seen. Any part of a scene that lies outside of the region is not visible (clipped). Of course, though the rendered scenes may be conceptually three-dimensional, they must be projected onto a two-dimensional screen. The available projections are orthographic projection and perspective projection. You get one guess as to which kind glOrtho uses. In orthographic projection, also known as parallel projection, an object appears the same size whether it is near or far from the viewer. Showing a simple, flat rectangle certainly does not need anything more complicated. Here is the declaration:

```
procedure glOrtho(left,right,bottom,top,zNear,zFar: GLdouble); stdcall;
```

GLdouble is the same thing as a Delphi double, an IEEE 64 bit floating point value.

```
GLdouble = double;
```

Left, right, bottom, and top describe the same area as the glViewPort, but with programmer-defined values for the edges. The rendering commands use values relative to these. In the above FormResize method, -1.0 is the new value given to the left side of the view port, and 1.0 is the value for the right side. The bottom of the view port is now -1.0, and the top is 1.0. The z-values represent directions perpendicular to the screen. The zNear value represents the z-axis coordinate of the clipping region face nearest the viewer, while zFar designates the face farthest from the viewer, deepest into the screen. Those values represent in 3-dimensional space whatever the programmer wants them to mean.

Flexibility

Here is another example of glOrtho with glViewport:

```
glViewPort(0,0,ClientWidth,ClientHeight);
glOrtho(3.204,17.2,-11.0,1.45,5.025,41.0);
```

The call to glViewPort remains the same to isolate the effect, in this example, of changing the values given to glOrtho. For example, the clipping region now has a value of 17.2 for its right side. It is still located at the same place as before, which is the right side of the view port, at ClientWidth. The value of 17.2 was just convenient. Delphi’ s Object Pascal language has a similar concept for arrays:

```
var
   first: array[0..9] of integer;
   second: array[1..10] of integer;
   third: array[100..109] of integer;
```

All three of the above variables are equivalent arrays of ten integers, but with different starting points. Why not just use zero-based arrays? After all, that is all that is available in some programming languages. The answer is that they are declared in a manner convenient for the programmer. An alternate form may allow less typing in the coding of an algorithm, or may more obviously represent some real world concept.

Similarly the programmer has the freedom to assign arbitrary values to the orthographic clipping region. He may choose values for computational convenience or for better conceptual representation of the scene. There is a little bit of order imposed on the glOrtho universe, however. Negative z-values are in front of the view port in the window, and positive z-values are behind the view port. Placing the clipping region entirely in front of the view port strains the hole-in-the-wall analogy, but the comparison has already served its purpose.

DRAWING

Command Placement

At last the time has arrived to draw something: a rectangle. As expected, the place to draw the rectangle is in the OnPaint event handler. The drawing commands need to appear between glBegin and glEnd; The latter requires no parameters; the former takes a parameter to set the drawing mode:

```
procedure glBegin(mode:GLenum); stdcall;
procedure glEnd; stdcall;
```

Between glBegin and glEnd goes a list of vertices, and possibly a few commands related to the vertices. Each vertex is simply a point in space specified by the glVertex command. This command takes many forms, but only a few of them are appropriate here:

```
procedure glVertex2f (x,y: GLfloat); stdcall;
procedure glVertex3f (x,y,z: GLfloat); stdcall;
procedure glVertex4f (x,y,z,w: GLfloat); stdcall;
```

State Variables

In the FormPaint method (Listing 2-2), not only does glClearColor set the clear color (background), but glColor sets the foreground color for drawing. These calls set state variables which remain the same until similar calls explicitly change them. If the command says to draw in blue, all drawing is done in blue until another call changes the state. OpenGL's state variables are in DLLs, not directly accessible to the programmer. You read and write OpenGL's state variables via function and procedure calls. The glColor command is before glBegin in the current event handler method, but it is one of the limited set of commands that can appear between glBegin and glEnd, since color changes are often necessary in the midst of a drawing sequence.

Listing 2-2. Setting State Variables in FormPaint.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  if not openGLReady then
      exit;
    {green background}
    glClearColor(0.0,0.4,0.0,0.0);
```

```
    glClear(GL_COLOR_BUFFER_BIT);

    {red rectangle}
    glColor3f(0.5,0.0,0.0);
    glBegin(GL_POLYGON);
      glVertex2f(0.25,0.25);
      glVertex2f(0.75,0.25);
      glVertex2f(0.75,0.75);
      glVertex2f(0.25,0.75);
    glEnd;
    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
       raise Exception.Create('Error in Paint'#13+
          gluErrorString(errorCode));
end;
```

Here is the declaration of the glColor command used in the FormPaint method:

```
procedure glColor3f(red,green,blue:GLfloat); stdcall;
```

The new constant has this declaration:

```
  GL_POLYGON = 9;
```

Since glColor3f has arguments of 0.5, 0.0, and 0.0, OpenGL is drawing in half-intensity red. Alpha is not specified, so it gets 0.0 also. In the calls to glVertex2f, no z-value is passed, so all of the rectangle lies in the z = 0.0 plane. That means the rectangle lies in the same plane as the view port, neither in front nor behind. Recall that glOrtho received arguments of -1.0 for left and 1.0 for right. Since the x-values in glVertex are all positive, the rectangle is entirely on the right side of the viewing area. The same is true of the y-values, so the rectangle lies entirely in the upper half of the viewing area. Remember that all the drawing coordinates are relative to the values defining the clipping region, so changing the arguments to glOrtho could affect the size, shape, and position of the drawing.

Drawing Mode

```
procedure glBegin(mode:GLenum); stdcall;
```

Note the argument passed to glBegin. GL_POLYGON informs OpenGL that the subsequent vertex list establishes the corners of a polygon. The number of vertices between glEnd and glBegin are the number of vertices in the polygon, which in this case is a quadrilateral. The first side of the polygon is a line from the first vertex to the second. The second side of the polygon is a line from the second vertex to the third, and so on until the polygon closes with a line from the last vertex to the first. OpenGL knows to do all that simply by being given GL_POLYGON. It further knows to fill in the interior of the polygon with the color previously specified in glColor. What if a complete scene needs more than one type of figure? Make multiple glBegin, glEnd pairs, passing the appropriate mode arguments.

Another new command introduced here is glFlush. For the sake of efficiency, the graphics engine does not necessarily start drawing as soon as it receives its first command. It may continue to accumulate information in a buffer until given a little kick. The call to glFlush forces it to start drawing.

```
procedure glFlush; stdcall;
```

To illustrate another drawing mode, GL_LINES, Listing 2-3 is the code for the OnPaint event handler that produced Figure 2-1, including a couple of new commands needed for that purpose.

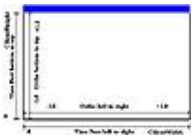

Figure 2-1.glViewPort and glOrtho

Listing 2-3. Using GL_LINES.

```
{white background}
glClearColor(1.0,1.0,1.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);

{top of the form}
glColor3f(0.0,0.0,1.0);
glRectf(-300.0,300.0,300.0,345.0);

glLineWidth(2.0);
glColor3f(0.0,0.0,0.0);
glBegin(GL_LINES);
    {left side}
    glVertex2f(-300.0,-300.0);
    glVertex2f(-300.0,300.0);

    {bottom}
    glVertex2f(-300.0,-300.0);
    glVertex2f(300.0,-300.0);

    {right side}
    glVertex2f(300.0,-300.0);
    glVertex2f(300.0,300.0);
```

```
glEnd;

glLineWidth(1.0);
glBegin(GL_LINES);
    {view port measure left to right}
    glVertex2f(-300.0,-340.0); {line}
    glVertex2f(300.0,-340.0);

    glVertex2f(-300.0,-320.0); {end}
    glVertex2f(-300.0,-360.0);

    glVertex2f(-300.0,-340.0); {end}
    glVertex2f(-290.0,-330.0);

    glVertex2f(-300.0,-340.0); {arrow}
    glVertex2f(-290.0,-350.0);

    glVertex2f(300.0,-320.0); {arrow}
    glVertex2f(300.0,-360.0);

    glVertex2f(300.0,-340.0); {arrow}
    glVertex2f(290.0,-330.0);

    glVertex2f(300.0,-340.0); {arrow}
    glVertex2f(290.0,-350.0);

    {view port measure bottom to top}
    glVertex2f(-325.0,-300.0); {line}
    glVertex2f(-325.0,300.0);

    glVertex2f(-312.0,-300.0); {end}
    glVertex2f(-337.0,-300.0);

    glVertex2f(-312.0,300.0); {end}
    glVertex2f(-337.0,300.0);

    glVertex2f(-325.0,-300.0); {arrow}
    glVertex2f(-332.0,-285.0);
```

```
glVertex2f(-325.0,-300.0); {arrow}
glVertex2f(-318.0,-285.0);

glVertex2f(-325.0,300.0); {arrow}
glVertex2f(-332.0,285.0);

glVertex2f(-325.0,300.0); {arrow}
glVertex2f(-318.0,285.0);

{ortho measure left to right}
glVertex2f(-300.0,-260.0); {line}
glVertex2f(300.0,-260.0);

glVertex2f(-300.0,-260.0); {arrow}
glVertex2f(-288.0,-272.0);

glVertex2f(-300.0,-260.0); {arrow}
glVertex2f(-290.0,-250.0);

glVertex2f(300.0,-260.0); {arrow}
glVertex2f(290.0,-250.0);

glVertex2f(300.0,-260.0); {arrow}
glVertex2f(288.0,-272.0);

{ortho measure bottom to top}
glVertex2f(-275.0,-300.0); {line}
glVertex2f(-275.0,300.0);

glVertex2f(-275.0,-300.0); {arrow}
glVertex2f(-282.0,-285.0);

glVertex2f(-275.0,-300.0); {arrow}
glVertex2f(-268.0,-285.0);

glVertex2f(-275.0,300.0); {arrow}
glVertex2f(-282.0,285.0);
```

```
    glVertex2f(-275.0,300.0); {arrow}
    glVertex2f(-268.0,285.0);
glEnd;
```

When the drawing mode is GL_LINES, each pair of calls to glVertex defines a line segment. The declaration for the new mode is:

```
GL_LINES = 1;
```

Rectangles are so common that OpenGL has a special command just for drawing them. The first two parameters define one vertex, and the next two define the opposite vertex. The rectangle assumes z = 0. The float version just introduced is one of many variations. Also introduced is a command for setting the width of the lines in pixels. The default width is 1.0.

```
procedure glRectf(x1,y1,x2,y2:GLfloat); stdcall;
procedure glLineWidth(width:GLfloat); stdcall;
```

ADDITIONAL CONSIDERATIONS

Naming Convention

You may have noticed by now that all the OpenGL command names begin with "gl." The graphics library has a well-defined naming convention, which this book mentions from time to time. So far the "gl" prefix has dominated, which does apply to all OpenGL commands. A "glu" prefix has also made a cameo appearance. It is the standard for all commands of the utility library. Chapter 1 presented the "wgl" prefix for the extensions to Microsoft Windows that support OpenGL rendering. Similarly the X Window System (for Unix) extensions use a "glx" prefix, and OS/2 Presentation Manager extensions use "pgl."

The naming convention does not stop with prefixes. The glVertex commands, for example, have a number suffix, such as 2, 3, or 4, indicating the number of parameters the command requires. A later chapter explains the fourth (w) parameter, but for now, if it receives no explicit value, it receives a value of 1.0. In the two-parameter version, even the z-value goes without assignment, so it gets a 0.0 value by default. The "f" suffix indicates these versions of glVertex take GLfloat parameters. Other paramter types are possible, so as many as 24 different versions of the glVertex command are available.

The FormPaint method used the glColor3f command. The “3” indicates that this version of the command takes three parameters, but there is another version of the command that takes four parameters. The “f” indicates that the parameters are of type glFloat.

Error Handling in Depth

Chapter 1 introduced glGetError, but gave it only superficial treatment. Checking it every time a command could possibly produce an error would have a performance penalty. Notice that FormPaint performs a single call to glGetError at the end of the method. Fortunately the graphics library is capable of storing multiple error codes. If several error codes have accumulated, multiple calls to glGetError are required to clear them all. Accordingly this application replaces its default exception handler. Place this at the end of the private section of the form's type declaration:

```
procedure ExceptionGL(Sender:TObject;E:Exception);
```

The application's OnException event handler activates whenever an exception climbs all the way up the stack without being handled along the way. The default handler simply displays the corresponding error message. To replace the default one with one more suitable for OpenGL, place this at the beginning of the FormCreate method:

```
Application.OnException := ExceptionGL;
```

Finally, here is the body of the new exception handler. By calling ShowException it does what the old exception handler would have done, then in the loop it calls glGetError and shows all the pending OpenGL errors until they have all been cleared:

```
procedure TForm1.ExceptionGL(Sender:TObject;E:Exception);
begin
  ShowException(Sender,E);
   repeat
     errorCode := glGetError;
     if errorCode<>GL_NO_ERROR then
        showMessage(gluErrorString(errorCode));
   until errorCode=GL_NO_ERROR;
end
```

Order of Events

Now, compile the program and launch it with the Windows "Run" command. Resize it several ways. Watch the shape of the rectangle as you do the resizing. Careful observation reveals that if the form only shrinks, the shape of the rectangle does not change. If the form enlarges either horizontally or vertically or both, the shape of the rectangle changes appropriately. An expected event is not firing. This calls for further investigation.

In the variable declaration section add:

```
    EventFile: TextFile;
```

So that an "Event.Log" text file opens for writing at program start, and closes at program end, make the end of the unit look like this:

```
initialization
   assignFile(EventFile,' Event.Log' );
   rewrite(EventFile);
finalization
   closeFile(EventFile);
end.
```

To record each firing of the OnPaint event, at the beginning of FormPaint add the following:

```
   writeln(EventFile, 'Paint  '+IntToStr(TimeGetTime));
```

TimeGetTime is a Windows API call that returns the number of milliseconds since Windows last loaded. To access it the unit needs MMSystem in the uses clause. Now, to log each firing of the OnResize event, at the beginning of FormResize add this line:

```
  writeln(EventFile, 'Resize '+IntToStr(TimeGetTime));
```

Now compile and run as before. Wait about one second. Shrink the form from the right side. Wait another second. Shrink the form from the bottom. Wait another second. Enlarge the form a little at the bottom. Now close the form. Examine the contents of the Event.Log file in the current (project) directory. At the top of the file are "Resize" and "Paint" with nearly identical time stamps, only 10 or 20 milliseconds apart. They fired at program start. About 1000 milliseconds later (depending on the accuracy of the "one second" wait) is another "Resize" line. That fired when the form shrank from the right side. Another 1000 milliseconds later is another "Resize" entry, which fired when the form shrank from the bottom. Still 1000 millisconds later is another "Resize" followed shortly by a "Paint."

After careful consideration, this makes sense. When the form shrinks, Windows only sends a Resize message. Why should Windows send a Paint message just for the covering of some already painted territory? Only form enlargement in one or both directions exposes new area that needs painting. Windows does not know about the special need to adjust the

interior of the form. Since the Windows Resize and Paint messages are what trigger the Delphi OnResize and OnPaint events, a form shrinkage does not trigger the necessary adjustment of the rectangle.

The way to remedy this little problem is to call FormPaint on those occasions when it would otherwise be neglected. FormResize is called for every OnResize event, so that is a good place to check whether either dimension enlarged. If not, then no OnPaint event happens, so you must call FormPaint directly. After cleaning out the text file code, add these fields to the private section of the form type declaration:

```
   oldw,
   oldh: integer;
```

The fields preserve the previous width and height for comparison with new values. Initialize them at the end of the FormCreate method:

```
   yldw := ClientWidth;
   oldh := ClientHeight;
```

Finally, this code at the end of FormResize insures the FormPaint method always fires when needed:

```
if (ClientWidth<=oldw) and (ClientHeight<=oldh) then
   FormPaint(Sender);
oldh := ClientHeight;
oldw := ClientWidth;
```

Save, compile and run (from Windows). Resize to your heart's content.

Matrices and Command Placement

At this point the program is ready for a little experimentation with glOrtho. Change left from – 1.0 to 0.0, then save, compile and run as before.

```
    glOrtho(0.0,1.0,-1.0,1.0,-1.0,1.0);
```

Remember that the values given to the vertices of the rectangle are relative to the clipping volume. Since the range of horizontal numbers in the call to glOrtho is half the former range, the rectangle should be twice as wide as before. Program execution shows that to be the case. Now, resize the form exactly one time. Something is wrong! The rectangle jumps to the left side. Resize the form again. The rectangle is gone, never to be seen again!

This experience exposes another important concept. OpenGL relies heavily on *matrices*. A matrix is a two-dimensional array of numbers for which a special set of mathematical operations apply. For every call to glOrtho, OpenGL creates a matrix from the parameters and multiplies it by the projection matrix. Each successive call changes the projection matrix. The symmetrical numbers passed to glOrtho just masked the potential strange behavior, but the unbalanced numbers for this version exposed the problem. One solution is to move the call to glOrtho to the last line of the FormCreate method. Save, compile and run. Resize the form a few times. Now the expected behavior manifests itself.

Go ahead and experiment some more with passing different values to glOrtho, leaving everything else the same. These variations should give a sense of the relationship between the values in the projection matrix and the values of the vertices.

View Port Manipulation

The view port is the next experimental subject. The orthographic projection can change some of its boundary numbers, but still have the same boundary (the view port). Just the arbitrary meaning of the numbers change. The view port boundary numbers have a real meaning, which is window coordinates. Changing these numbers changes the actual boundary of the view port.

Restore the call to glOrtho to its symmetrical arrangement, but leave it at the bottom of FormCreate.

```
   glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
end; {FormCreate}
```

In FormResize change the vertical extent in the call to glViewPort.

```
   glViewPort(0,0,ClientWidth,ClientHeight div 2);
```

Compile and run as before. From this point on using the Windows Run command rather than running from the IDE is taken for granted and needs no further mention. Now the view port only covers the lower half of the window. Remember that OpenGL coordinates normally start from the bottom of the window, even when using "window coordinates." The red rectangle is still in the upper right quadrant of the view port, but since the view port itself is confined to the lower half if the window, the rectangle is now in the lower half of the window. Since the view port is only half as tall, the rectangle is half as tall as it was.

Now, try it the other way.

```
   glViewPort(0,0,ClientWidth div 2,ClientHeight);
```

Compile and run. This time everything lies on the left side of the screen. The rectangle is still in the right side of the view port, but it lies in the left side of the window. Notice that with these restrictions on the size of the view port, the background color still covers the whole window. This experiment should be enough to give a good feel for glViewport, so take out the div operators to restore the original appearance.

Mixing Windows Components

One of the really neat features of OpenGL is that it does not prevent using regular Windows components. This opens many possibilities for user interaction with your graphics constructs. The coming chapters will explore a number of different ways users can give information to your applications. Since these are illustrations and exercises, they are not the ultimate in slick, state-of-the-art front ends; they are just enough to demonstrate a technique. For the first such demonstration, drop a MainMenu component on the form. Right-Click the component and select the Menu Designer. Give it one menu item with a caption of "&Z-Value." Close the Menu Designer, click the menu item, and edit the event handler as follows:

```
procedure TForm1.Zvalue1Click(Sender: TObject);
var
   zString: string[7];
begin
   zString := InputBox('Z-Value','float:','0.0');
end;
```

The first argument gives the title of the input box. The second is the prompt. "float:" serves as a reminder to enter a floating point value. The third argument provides a default value to return.

Position the form near the center of the screen or set the form's Position property to poScreenCenter. Save, compile and run. Click the menu item. Close the input box. The menu showed properly. The input box showed properly. Since the form shows at or near the center of the screen, and the input box shows near the center of the screen, the input box has to obscure part of the form. Since there was no hole left in the form when the input box closed, then the OnPaint event must have fired. The only adverse effect is the fact that the view port is a little shorter, since the presence of the menu reduced the ClientHeight.

Pushing the (X,Y,Z) Limits

Add zvalue as a new field to the private section of the form declaration.

```
  private
     GLContext: HGLRC;
     glDC: HDC;
     errorCode: GLenum;
     openGLReady: boolean;
     oldw,
     oldh: integer;
```

```
    zvalue: glFloat;
    procedure ExceptionGL(Sender:TObject;E:Exception);
```

Edit the Zvalue1Click event handler.

```
procedure TForm1.Zvalue1Click(Sender: TObject);
var
  zString: string[7];
begin
  zString := InputBox('Z-Value','float:','0.0');
  zvalue := StrToFloat(zString);
end;
```

Change the calls to glVertex from the two-parameter version to the three-parameter version inside FormPaint, using the freshly-minted zvalue as the third argument.

```
  glBegin(GL_POLYGON);
      glVertex3f(0.25,0.25,zvalue);
      glVertex3f(0.75,0.25,zvalue);
      glVertex3f(0.75,0.75,zvalue);
      glVertex3f(0.25,0.75,zvalue);
  glEnd;
```

Save, compile and run. Select the menu item and enter 0.5 in the input box (and press the Ok button, of course). Do it again, but enter 1.0 as the value. Again, but use 1.01. Something different happened! The rectangle disappeared! Every time the input box covers part of form and closes, the form must re-paint. The results of each new zvalue appear as soon as the box closes. As long as zvalue is between the z-axis limits of the clipping volume (the glOrtho projection), the rectangle is visible. The side of the clipping volume toward the viewer is at z = 1.0. The side of the clipping volume away from the viewer, into the screen is at z = -1.0. As long as the rectangle lies between – 1.0 and 1.0 on the z-axis, it is visible. Values of 0.5, 1.0, -0.5, and -1.0 kept the figure in sight, but 1.01 and – 1.01 made it vanish.

Use the Menu Designer to add two more main menu items. The captions are &Y-Value and &X-Value. Add the following event handlers:

```
procedure TForm1.YValue1Click(Sender: TObject);
var
  yString: string[7];
```

```
begin
  yString := InputBox('Y-Value','float:','0.0');
  yvalue := StrToFloat(yString);
end;
procedure TForm1.XValue1Click(Sender: TObject);
var
  xString: string[7];
begin
  xString := InputBox('X-Value','float:','0.0');
  xvalue := StrToFloat(xString);
end;
```

Add xvalue and yvalue to private section of the form declaration:

```
  private
    GLContext: HGLRC;
    glDC: HDC;
    errorCode: GLenum;
    openGLReady: boolean;
    oldw,
    oldh: integer;
    xvalue,
    yvalue,
    zvalue: glFloat;
    procedure ExceptionGL(Sender:TObject;E:Exception);
```

Use the new fields in the drawing:

```
glBegin(GL_POLYGON);
   glVertex3f(xvalue+0.25,yvalue+0.25,zvalue);
   glVertex3f(xvalue+0.75,yvalue+0.25,zvalue);
   glVertex3f(xvalue+0.75,yvalue+0.75,zvalue);
   glVertex3f(xvalue+0.25,yvalue+0.75,zvalue);
glEnd;
```

Save, compile and run. Try 0.5 and – 1.5 for yvalue, then go back to 0.0. Try the same numbers for xvalue. Now the meaning of a clipping volume should be quite clear. If any part of the figure goes outside the boundaries of the clipping volume, that part is clipped off, but the rest remains visible.

Figure 2-2.Clipping with yvalue = 0.5

Naturally, good code uses a more reliable method of insuring a re-paint than just having an input box go away, but for this little clipping demonstration, it was sufficient.

DEFINITIONS

Clipping	The act of not drawing the part of a scene that lies beyond a specified plane.
Clipping volume	A volume enclosed by clipping planes. The parts of a scene lying outside the volume are not drawn.
Foreground color	The current drawing color.
Matrix	A two-dimensional array of numbers for which special mathematical operations have been defined.
Orthographic projection	Parallel projection.
Parallel projection	A projection that does not make size correction for distance.
Polygon	A closed two-dimensional figure consisting of three or more vertices with sides connecting one vertex to the next, and finally a side connecting the last vertex to the first.
Projection	The mathematical transformation of points in a three-dimensional scene onto a flat surface, such as the

	computer screen.
Quadrilateral	A four-sided polygon.
Render	To convert drawing elements into an image. To draw with OpenGL.
State variables	Much OpenGL information is contained in variables whose values remain the same until deliberately changed. This kind of information is a state. The varaibles that hold such information are state variables.
Vertex	A point in three-dimensional space, usually representing an endpoint of a line segment or a corner of a polygon.
View port	The portion of a window's client area in which to draw the clipping area.

IDENTIFIER REFERENCE

glBegin	An OpenGL command designating the start of a list of vertices. It receives a mode parameter that tells OpenGL what to do with those vertices.
GLbitfield	An OpenGL type that maps to Delphi cardinal.
GLclampf	An OpenGL type equivalent to Delphi single.
glColor	An OpenGL command that sets the foreground color.
GLdouble	An OpenGL type equivalent to Delphi double.
glEnd	An OpenGL command that terminates a list of vertices started with glBegin.
GLenum	An OpenGL type that maps to Delphi cardinal.
GLfloat	An OpenGL type that maps to Delphi single.
glFlush	An OpenGL command that forces drawing to the screen.
GLint	An OpenGL type that maps to Delphi integer.
glLineWidth	An OpenGL command that specifies the width, in pixels, of all subsequent lines drawn.
glOrtho	An OpenGL command that specifies an orthographic projection and clipping volume.
glRect	An OpenGL command for drawing a rectangle.

GLsizei	An OpenGL type that maps to Delphi integer.
glVertex	An OpenGL command for specifying the coordinates of a vertex.
glViewport	An OpenGL command that specifies the viewport in window coordinates.
GL_COLOR_BUFFER_BIT	An OpenGL constant that identifies the color buffer.
GL_LINES	An OpenGL constant to pass to glBegin. Each pair of vertices then specifies a line segment, separate from any of the other line segments.
GL_POLYGON	An OpenGL constant to pass to glBegin. All the vertices listed before glEnd are connected to form a polygon.
TimeGetTime	A Windows function that returns the number of milliseconds since Windows was started.

SUMMARY

Chapter 2 presented the following concepts:

1. Perform drawing in the window's OnPaint event handler.
2. Drawing occurs within the bounds of the view port.
3. The view port definition uses the operating system's window coordinates, but counts in the y-direction from the bottom up.
4. Use glColor to set the foreground drawing color, changing as needed. The color is one of many state variables that remain the same until the programmer directly changes them via function or procedure call.
5. Presenting a 3-D scene on a 2-D screen requires projection.
6. Orthographic projection does not change the apparent size of an object to correct for distance. It is also called parallel projection.
7. The projection is a volume in space bounded by the view port.
8. The programmer may assign the numbers defining the projection volume for clarity or convenience.
9. The projection region is a clipping volume. Any part of a scene that lies outside the volume simply does not appear.
10. Adjust the view port in the window's OnResize event handler.
11. Projection involves matrix multiplication. Repeated invocation is repeated multiplication which probably gives unintended results. The solution in this chapter was to invoke projection only once in the FormCreate method.

12. OpenGL does not hinder the use of Windows features such as menus and dialog boxes.

13. The glGetError function returns the error code number for recent operations. It returns a number equal to GL_NO_ERROR if no errors have accumulated. OpenGL stores accumulated errors, so the program must call glGetError repeatedly until GL_NO_ERROR comes back.

Chapter 3

Introducing 3-D

Chapter 2 introduced drawing a two-dimensional figure and projecting it onto the screen. However, OpenGL is rather under-utilized in 2-D. OpenGL shines in three-dimensional operations. This chapter explores the third (z) dimension and shows a way to draw a cube in three-dimensional space. The cube has six square faces with different colors for each. This chapter shows how to rotate, move, and stretch the cube, as well as cut it open to show what is inside.

The Rect program from chapter 2 makes a good starting point for the next program. Create a new directory as before and name it "Chapter.3." Under that directory create a directory called "Cube." In Delphi load the Rect.Dpr project and do a File|Save Project As in the new directory, naming it "Cube.Dpr." Do File|Save As for the "Rect1.Pas" unit. Name it "Cube1.Pas" in the new directory.

DEPTH

The introduction of a third dimension calls for an understanding of the direction in which that dimension runs. Construct a new FormPaint method as shown in Listing 3-1.

Listing 3-1. FormPaint Method with Depth.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
   if not openGLReady then
       exit;

   {green background}
   glClearColor(0.0,0.4,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

   glBegin(GL_POLYGON); {near face}
      glColor3f(0.5,0.0,0.0);
      glVertex3f(-0.4,-0.4,-0.4);
      glVertex3f(+0.4,-0.4,-0.4);
      glVertex3f(+0.4,+0.4,-0.4);
      glVertex3f(-0.4,+0.4,-0.4);
```

```
    glEnd;

    glBegin(GL_POLYGON); {back face}
       glColor3f(0.0,0.0,0.5);
       glVertex3f(-0.2,-0.2,+0.4);
       glVertex3f(+0.6,-0.2,+0.4);
       glVertex3f(+0.6,+0.6,+0.4);
       glVertex3f(-0.2,+0.6,+0.4);
    glEnd;

    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
       raise Exception.Create('Error in Paint'#13+
          gluErrorString(errorCode));
end; {FormPaint}
```

The z value of the original (red) rectangle is – 0.4. The second rectangle is 0.8 units farther along the positive z direction with a z value of +0.4. The second (blue) rectangle also lies 0.2 units farther in the positive x direction and 0.2 units farther in the positive y direction, so that one rectangle does not completely obscure the other.

Without a *depth test* an object drawn later may appear on top of an object drawn earlier, no matter how near or far the respective objects actually were. The explanation for the depth test appears just a few paragraphs later, but for now, add some lines to the bottom of the FormCreate method:

```
    glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity;
end; {FormCreate}
```

Save, compile, and run to see the result. As Figure 3-1 shows, the red rectangle is in front and the blue rectangle is in back. The conclusion is that with an orthographic projection, the farther an object lies in the positive z direction, (or at least less negative), the farther the object is from the viewer.

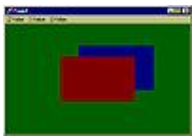

Figure 3-1.Depth

CUBE

Now you are ready for another illustration. For the sake of this illustration, comment out the glEnable(GL_DEPTH_TEST) statement you just put in.

Square View Port

The new project name certainly suggests the next shape under discussion. A cube provides a fairly simple figure for introducing 3-D. In geometry a cube has identical height, width, and depth. Unfortunately the projection of such a figure onto the computer screen may not look like a cube unless the view port is square. Since the current view port matches the form's client height and client width, those properties should be set equal to each other. Select the form and go to the object inspector. Set ClientHeight to 340, and set ClientWidth to 340.

Distinguish Faces

A cube has six square faces. In FormPaint draw six colored squares, using different colors for each face for easy distinction. If all the faces of the cube were the same color, they would blend into each other and obscure the shape. The application of lighting effects solves such problems, but that subject comes later. Modify the FormPaint method as in listing 3-2.

Listing 3-2. Six Colored Faces.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
   if not openGLReady then
      exit;

   {green background}
   glClearColor(0.0,0.4,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

   glBegin(GL_POLYGON); {near face}
```

```
   glColor3f(0.5,0.0,0.0);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(+0.4,+0.4,-0.4);
   glVertex3f(-0.4,+0.4,-0.4);
glEnd;

glBegin(GL_POLYGON); {right face}
   glColor3f(0.3,0.3,0.8);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(+0.4,-0.4,+0.4);
   glVertex3f(+0.4,+0.4,+0.4);
   glVertex3f(+0.4,+0.4,-0.4);
glEnd;

glBegin(GL_POLYGON); {left face}
   glColor3f(0.5,0.0,0.5);
   glVertex3f(-0.4,-0.4,+0.4);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(-0.4,+0.4,-0.4);
   glVertex3f(-0.4,+0.4,+0.4);
glEnd;

glBegin(GL_POLYGON); {back face}
   glColor3f(0.0,0.0,0.5);
   glVertex3f(+0.4,-0.4,+0.4);
   glVertex3f(-0.4,-0.4,+0.4);
   glVertex3f(-0.4,+0.4,+0.4);
   glVertex3f(+0.4,+0.4,+0.4);
 glEnd;

 glBegin(GL_POLYGON); {bottom face}
   glColor3f(0.5,0.5,0.0);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(-0.4,-0.4,+0.4);
   glVertex3f(+0.4,-0.4,+0.4);
   glVertex3f(+0.4,-0.4,-0.4);
 glEnd;
```

```
    glBegin(GL_POLYGON); {top face}
      glColor3f(0.3,0.0,0.1);
      glVertex3f(-0.4,+0.4,-0.4);
      glVertex3f(+0.4,+0.4,-0.4);
      glVertex3f(+0.4,+0.4,+0.4);
      glVertex3f(-0.4,+0.4,+0.4);
    glEnd;

    glFlush;

    errorCode := glGetError;
    if errorCode<>GL_NO_ERROR then
      raise Exception.Create('Error in Paint'#13+
        gluErrorString(errorCode));
end; {FormPaint}
```

Depth Test

Save, compile, and run and admire the beautiful cube! Well, at least tolerate the colored rectangle. Why is there no evidence of a third dimension? Actually the straight-on view guarantees that only the front face shows. Or does it? Notice the color. The front face is red. The back face is blue. The blue back face shows on the screen! Why? OpenGL drew all six faces. The four edge-on faces have no effect, but the back face was drawn after the front face, so it is the one that shows. Fortunately there is a way to tell OpenGL to test depth and not draw pixels that should be hidden. This ability, as well as others, becomes effective by means of the glEnable command:

```
procedure glEnable(cap:GLenum); stdcall;
```

Turn on each capability by passing the appropriate constant. Here is the declaration for the depth-testing constant:

```
   GL_DEPTH_TEST = $0B71;
```

Enabling depth testing sets up a depth buffer that contains depth information on each pixel within the clipping volume. Of the pixels that map to the same spot on the screen, OpenGL draws only the one nearest the viewer. This process is OpenGL's method of hidden surface removal. The bottom of the FormCreate method with depth testing added follows:

```
   openGLReady := true;
   oldw := ClientWidth;
   oldh := ClientHeight;
   glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
   glEnable(GL_DEPTH_TEST);
end; {FormCreate}
```

Matrices Revisited

Save, compile, and run. Observe that no change is evident. This means that more must be done to make the depth test work. Therefore it is time to give more consideration to matrices. OpenGL is full of them. Matrices describe the very scene itself, and there are matrices for texture, for projection, etc. There are commands to select among those matrices. There are even matrix stacks. Some commands, such as glLoadIdentity, directly manipulate the active matrix, and other commands use their parameters to construct a matrix, then multiply the appropriate matrix by the constructed matrix.

The ModelView Matrix

Consider a 4x4 matrix that represents the coordinate system of the scene. Call it the Model matrix. Apply this matrix to every vertex in the scene as part of the process of illuminating the appropriate dots on the screen. Now consider a 4x4 matrix that represents the coordinates of the viewer. Call it the View matrix. Finally consider Einstein's Theory of Relativity. What?! Well, anyway, think of two automobiles side by side. The observer is in the first car and sees the second car sliding backwards. Or is the first car with the observer actually moving forward? Or does it matter? In a universe that contains only those two cars, either description is equally valid. Similarly the Model matrix and the View matrix are equally useful. In fact they are the same matrix. Its true name is the ModelView matrix.

Here is the OpenGL command that identifies which matrix other commands will affect:

```
procedure glMatrixMode(mode:GLenum); stdcall;
```

These are the constants to pass for the mode parameter:

```
  GL_MODELVIEW  = $1700;
  GL_PROJECTION = $1701;
  GL_TEXTURE    = $1702;
```

Here is the declaration of the command that sets the current matrix to be the *identity matrix*, which is a matrix that, when multiplied with another matrix, leaves a product the same as the other matrix.

```
procedure glLoadIdentity; stdcall;
```

For the three-dimensional rendering program to work, make the bottom of FormCreate look like this:

```
  glEnable(GL_DEPTH_TEST);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
end;  {FormCreate}
```

So save, compile, and run. Ah! The correct face shows. But it still does not look like a cube.

ROTATION

Three Faces

Looking at a cube straight on just does not reveal its third dimension. The cube must have an orientation that shows more than one face. Indeed, a cube showing two faces is not much better, because it still just looks like two rectangles side by side. The obvious answer is to draw the cube skewed at some angle. Not only that, but the rotation must not be about a single coordinate axis, or about any line parallel to a single axis, otherwise no more than two faces can show.

glRotate

Drawing the cube rotated calls for some ugly math to apply to each vertex. Actually the math involved looks fairly simple when expressed as matrices, but it is ugly because the programmer has to carry out the details. There is a better way! Draw the cube simply, as before, then let OpenGL rotate it. Here is the needed command in two versions:

```
procedure glRotated(angle,x,y,z:GLdouble); stdcall;
procedure glRotatef(angle,x,y,z:GLfloat); stdcall;
```

PURPOSE The purpose of this command is to define an axis of rotation and an amount of rotation. The command then causes the scene to rotate about the axis. Or it causes the coordinate system of the scene, and therefor the viewing angle of the observer, to rotate about the axis. Take your choice; it is relative! What really happens is that OpenGL creates a rotation matrix from the parameters and multiplies it with the active matrix, in this case, the ModelView matrix.

PARAMETERS This simple program certainly does not need double precision to accomplish its purpose, so the float version is the proper choice for the command.

Either version uses four parameters. The angle parameter defines the amount of rotation. Interestingly it takes its value in degrees rather than radians, which is fortunate for mere mortals who do not spend their days immersed in trigonometry. The x, y, and z parameters define a point in three-dimensional space with respect to the coordinates of the clipping (projection) volume.

The given point helps define the *vector* that serves as the axis of rotation. Two points can define a line; the other point is (0.0,0.0,0.0), the origin for the projection coordinate system. Think of a vector as a line with direction, from the origin toward the other point, and then continuing on forever.

DIRECTION To understand the direction of rotation, imagine an observer located at the origin looking toward the point defined by the x, y, and z parameters. The object turns counter-clockwise with respect to that vantage point.

Action

In the Menu Designer add a fourth main menu item. Give it a caption of &Angle and exit from the Menu Designer. Click the menu item to produce the event handler and fill it in as follows:

```
procedure TForm1.Angle1Click(Sender: TObject);
var
   astring: string[7];
begin
   astring := InputBox('Angle of Rotation','float:','0.0');
   angle := StrToFloat(astring);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity;

   glRotatef(angle,xvalue,yvalue,zvalue);
end;
```

The other three menu choices, left over from the Rect program, set the coordinates for the point (other than the origin) that defines the axis of rotation. The new menu choice defines the angle of rotation and performs the rotation. Notice that glMatrixMode makes the ModelView matrix active every time, since that is the matrix on which to perform the action. This action is one of several *transformations*. OpenGL enables the programmer to perform three basic ModelView transformations or combinations thereof on the ModelView matrix, rotation, *translation* (movement in a straight line through space), and *scale* (changing the size of an object in each of the three dimensions independently). The call to glLoadIdentity is present to insure that the rotation transformation acts from a known and understood condition. Some situations may naturally call for allowing the transformations to accumulate, but not this simple interactive angle entry.

Save, compile and run. Enter coordinates to define the axis (vector) of rotation. Enter various angles and see the results. Notice that any multiple of 360.0 returns the cube to its original orientation and that negative angles rotate the cube clockwise. All the different colors for the faces help identify which faces come into view.

TWO-FACED POLYGONS

Open the Cube

There is no way to avoid it. Polygons are two-faced. That is not a character assassination but a geometric statement. When a polygon facing the viewer rotates 180 degrees, it does not disappear; it has a back face that the viewer can see. To illustrate, expose the inside of the cube by commenting out the right side as in Listing 3-3.

Listing 3-3. Right Side Missing.

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  if not openGLReady then
      exit;

   {green background}
   glClearColor(0.0,0.4,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

   glBegin(GL_POLYGON); {near face}
      glColor3f(0.5,0.0,0.0);
      glVertex3f(-0.4,-0.4,-0.4);
      glVertex3f(+0.4,-0.4,-0.4);
      glVertex3f(+0.4,+0.4,-0.4);
      glVertex3f(-0.4,+0.4,-0.4);
   glEnd;
(*
   glBegin(GL_POLYGON); {right face}
      glColor3f(0.3,0.3,0.8);
      glVertex3f(+0.4,-0.4,-0.4);
      glVertex3f(+0.4,-0.4,+0.4);
      glVertex3f(+0.4,+0.4,+0.4);
      glVertex3f(+0.4,+0.4,-0.4);
   glEnd;
 *)
   glBegin(GL_POLYGON); {left face}
      glColor3f(0.5,0.0,0.5);
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(-0.4,-0.4,-0.4);
      glVertex3f(-0.4,+0.4,-0.4);
      glVertex3f(-0.4,+0.4,+0.4);
   glEnd;
```

```
   glBegin(GL_POLYGON); {back face}
      glColor3f(0.0,0.0,0.5);
      glVertex3f(+0.4,-0.4,+0.4);
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(-0.4,+0.4,+0.4);
      glVertex3f(+0.4,+0.4,+0.4);
   glEnd;

   glBegin(GL_POLYGON); {bottom face}
      glColor3f(0.5,0.5,0.0);
      glVertex3f(-0.4,-0.4,-0.4);
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(+0.4,-0.4,+0.4);
      glVertex3f(+0.4,-0.4,-0.4);
   glEnd;

   glBegin(GL_POLYGON); {top face}
      glColor3f(0.3,0.0,0.1);
      glVertex3f(-0.4,+0.4,-0.4);
      glVertex3f(+0.4,+0.4,-0.4);
      glVertex3f(+0.4,+0.4,+0.4);
      glVertex3f(-0.4,+0.4,+0.4);
   glEnd;

   glFlush;

   errorCode := glGetError;
   if errorCode<>GL_NO_ERROR then
      raise Exception.Create('Error in Paint'#13+
         gluErrorString(errorCode));
end; {FormPaint}
```

Save, compile, and run. With the menus make xvalue and yvalue each equal to 1.0, but leave zvalue alone. For the angle enter 30.0 (degrees). Note the inside face for the back square and the top square. Experiment with various combinations of positive and negative values for all four parameters until all interior faces have presented themselves.

Culling

There it is! Those polygons have a back side! What if the cube was not open? The shocking truth is that even if there is no possible way to see the back side of the polygons, OpenGL draws them! Fortunately *depth testing* (hidden surface removal) keeps them from bleeding though. Better yet, an OpenGL command can tell it not even to draw them.

```
procedure glCullFace(mode:GLenum); stdcall;
```

The possible values for mode (with obvious meanings) are:

```
   GL_FRONT         = $0404;
   GL_BACK          = $0405;
   GL_FRONT_AND_BACK= $0408;
```

To *cull* faces means to remove them from the list of things to draw. Like many other OpenGL options, the programmer must first enable this option before using it. The value to give glEnable is:

```
   GL_CULL_FACE= $0B44;
```

Add some code so the bottom of FormCreate looks like this:

```
   glEnable(GL_DEPTH_TEST);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity;

   glEnable(GL_CULL_FACE);
   glCullFace(GL_BACK);
end; {FormCreate}
```

Save, compile, and run yet again. As before, experiment with various combinations of positive and negative values for all four parameters and see what the inside of the cube looks like when the inner faces simply are not drawn.

VERTEX DIRECTION

How does OpenGL know which face of a polygon the programmer intended to be front and which to be back? The answer lies in the order of the vertices. Consider the front square of this chapter's cube.

```
glBegin(GL_POLYGON); {near face}
   glColor3f(0.5,0.0,0.0);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(+0.4,+0.4,-0.4);
   glVertex3f(-0.4,+0.4,-0.4);
glEnd;
```

Mentally number the vertices from one to four. Vertex one is the lower left corner. Vertex two is the lower right corner. Vertex three is the upper right corner. Vertex four is the upper left corner. Imagine movement from one to two to three to four and back to one. The direction or *winding* is counter-clockwise, which is the default direction for OpenGL to identify the front face of a polygon. Those same vertices, if viewed from the other side, would present a clockwise winding. Figure 3-2 shows the counter-clockwise winding of the front view.

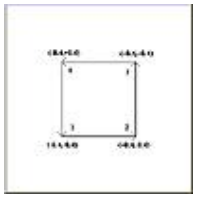

Figure 3-2.Counter-Clockwise Winding.

Consider the back square of the cube. Its vertex order was designed to present a clockwise winding from the viewer's position, making the face inside the cube the back face and the face outside the cube the front face. That is the usual arrangement. If the cube were rotated 180 degrees around the z-axis, the front face of that square would be toward the viewer. The next experiment reverses the back square's winding. Here is the original code for the back square:

```
glBegin(GL_POLYGON); {back face}
   glColor3f(0.0,0.0,0.5);
   glVertex3f(+0.4,-0.4,+0.4);
   glVertex3f(-0.4,-0.4,+0.4);
```

```
      glVertex3f(-0.4,+0.4,+0.4);
      glVertex3f(+0.4,+0.4,+0.4);
   glEnd;
```

Rearrange the vertices to reverse the winding:

```
   glBegin(GL_POLYGON); {back face}
      glColor3f(0.0,0.0,0.5);
      glVertex3f(+0.4,+0.4,+0.4);
      glVertex3f(-0.4,+0.4,+0.4);
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(+0.4,-0.4,+0.4);
   glEnd;
```

Save, compile, and run. Give the xvalue 0.3, the yvalue 1.0, and the angle 50.0. While the other interior faces have been culled, the interior face of the back square has been drawn. With a large enough angle, the back square is toward the viewer, but since its exterior face is a "back" face, it does not appear. Indeed, only the bottom of the cube shows at all.

TRANSLATION

Restore the original winding of the back square and uncomment the right square. While rotation may be the most interesting transformation, it is certainly not the only one of importance. Translation may be the most common transformation.

Menu

Use the Menu Designer to add &Translate as another main menu item. Click on it and fill in the event handler thusly:

```
procedure TForm1.Translate1Click(Sender: TObject);
begin
   glMatrixMode(GL_MODELVIEW);
   glTranslatef(xvalue,yvalue,zvalue);
   Invalidate;
end;
```

The glTranslate command takes three parameters that specify the relative amount of movement in each direction. The translation command has two forms:

```
procedure glTranslated(x,y,z:GLdouble); stdcall;
```

```
procedure glTranslatef x,y,z:GLfloat); stdcall;
```

Accumulation

Notice the absence of glLoadIdentity. Transformations will accumulate and demonstrate the effects of combining different kinds. Comment out the glLoadIdentity in the Angle1Click method as well. Notice also the presence of the Invalidate Windows API call. This causes Windows to send a WM_PAINT message, which triggers the OnPaint event. This is certainly superior to the (temporary) reliance on covering and uncovering the window by another window.

More Clipping

As usual, save, compile, run, and play with the various menu choices and values. The transformations accumulate, and may allow the cube to disappear altogether. Certainly the translation transformation provides more opportunity to experiment with clipping. Enough translation in the positive z direction provides some interesting effects. The far side of the cube disappears as if cut off by an invisible wall.

Did you see that flicker each time you selected Translate? Ugh! Do not despair; the chapter on animation shows how to redraw images without flicker.

Even more interesting is the effect of translation in the negative z direction. With just the right z values, a face or a corner nearest the viewer disappears, allowing a peek inside the cube without having to neglect drawing a square. What does the viewer see inside? He sees a mysterious green nothing because of the culling, as in Figure 3-3, where clipping in both the negative y direction and the negative z direction is evident.

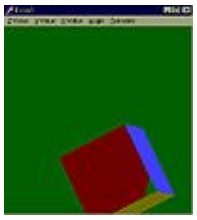

Figure 3-3.Clipping and Culling.

SCALE

Menu

With the Menu Designer add still another main menu item, &Scale. Produce the following event handler in the same manner as before:

```
procedure TForm1.Scale1Click(Sender: TObject);
begin
   glMatrixMode(GL_MODELVIEW);
   glScalef(xvalue,yvalue,zvalue);
   Invalidate;
end;
```

The only new command has the following declarations:

```
procedure glScaled(x,y,z:GLdouble); stdcall;
procedure glScalef(x,y,z:GLfloat); stdcall;
```

Description

The scale command multiplies the size of the objects in the scene. The value given for the x parameter multiplies the x-dimension, the y value passed to the command multiplies the y-dimension, and the z parameter's value multiplies the z-dimension. A call to glScale(2.0,2.0,2.0) makes the cube twice as wide, twice as tall, and twice as deep as the original specifications. So why are there three parameters? Each dimension's multiplier has a separate specification so each dimension can be scaled differently. A call to glScale(0.5,3.0,0.8) makes the cube (or whatever drawing follows) only half as wide, three times as tall, and 80 percent of the original depth. The programmer can draw something and then distort it freely. Of course a cube so mutilated is no longer a cube. Save, compile, run, and experiment. Figure 3-4 shows a distorted (former) cube.

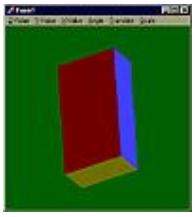

Figure 3-4.The Cube After glScale.

ORDER OF TRANSFORMATIONS

A New Interface

The admittedly clumsy interface used thus far has exhausted its usefulness. It was kept simple in order to minimize distraction from the OpenGL coding, but further development calls for a better user interface. The new interface is nothing elegant, but is a great improvement.

Create a new directory under "Chapter.3" named "Multi." Do a File|Save Project As of Cube.Dpr into the new directory, calling it Mult.Dpr. Save Cube1.Pas as Multi.Pas. Empty all the menu item event handlers by deleting the lines of code within and save again so Delphi can remove the newly empty event handlers. In the Menu Designer delete the existing menu items and put in &Parameters, &Transformation, &Identity, and &Cull Disable. Create and fill in their event handlers as in Listing 3-4.

Listing 3-4. Menu Item Event Handlers.

```
procedure TForm1.Parameters1Click(Sender: TObject);
begin
```

```
   FormTransform.ShowModal;
end;

procedure TForm1.Transformation1Click(Sender: TObject);
var
   list: TStringList;
   lno: integer;
begin
   list := TStringList.Create;
   list.sorted := true;
   with list,transData do
   begin
     if translate then
        Add(OrT+'T');
     if scale then
        Add(OrS+'S');
     if rotate then
        Add(OrR+'R');
     if count=0 then
        exit;

     glMatrixMode(GL_MODELVIEW);
     for lno := 0 to count-1 do
     begin
        case strings[lno][2] of
           'R': begin
              glRotatef(ra,rx,ry,rz);
           end;
           'S': begin
              glScalef(sx,sy,sz);
           end;
           'T': begin
              glTranslatef(tx,ty,tz);
           end;
        end; {case}
      end; {for}
    end; {with}
    List.Free;
```

```
    Invalidate;
end;
procedure TForm1.Identity1Click(Sender: TObject);
begin
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity;
   Invalidate;
end;

procedure TForm1.Cull1Click(Sender: TObject);
begin
   glDisable(GL_CULL_FACE);
   Invalidate;
end;
```

Create a new form and name it FormTransform, then save the unit as Trans1. To the form add three check boxes, four labels, ten edit controls, three radio groups, and two bit buttons to look like Figure 3-5. An efficient way to describe the component names and properties is to show the form as text as in Listing 3-5.

Listing 3-5. FormTransform as Text.

```
object FormTransform: TFormTransform
   Left = 200
   Top = 100
   Width = 360
   Height = 185
   Caption = 'Transformation Parameters'
   Font.Charset = DEFAULT_CHARSET
   Font.Color = clWindowText
   Font.Height = -11
   Font.Name = 'MS Sans Serif'
   Font.Style = []
   PixelsPerInch = 96
   TextHeight = 13
   object Label1: TLabel
      Left = 76
      Top = 4
      Width = 7
      Height = 13
      Caption = 'X'
   end
   object Label2: TLabel
      Left = 120
      Top = 4
      Width = 7
      Height = 13
      Caption = 'Y'
   end
   object Label3: TLabel
      Left = 164
      Top = 4
      Width = 7
      Height = 13
      Caption = 'Z'
   end
   object Label4: TLabel
      Left = 208
      Top = 4
```

```
   Width = 27
   Height = 13
   Caption = 'Angle'
end
object CheckBoxTranslate: TCheckBox
   Left = 4
   Top = 20
   Width = 64
   Height = 24
   Caption = 'Translate'
   TabOrder = 0
   OnClick = CheckBoxTranslateClick
end
object CheckBoxScale: TCheckBox
   Left = 4
   Top = 56
   Width = 64
   Height = 24
   Caption = 'Scale'
   TabOrder = 5
   OnClick = CheckBoxScaleClick
end
object CheckBoxRotate: TCheckBox
   Left = 4
   Top = 92
   Width = 64
   Height = 24
   Caption = 'Rotate'
   TabOrder = 10
   OnClick = CheckBoxRotateClick
end
object EditTX: TEdit
   Left = 72
   Top = 20
   Width = 40
   Height = 21
   Enabled = False
   TabOrder = 1
```

```
end
object EditTY: TEdit
   Left = 116
   Top = 20
   Width = 40
   Height = 21
   Enabled = False
   TabOrder = 2
end
object EditTZ: TEdit
   Left = 160
   Top = 20
   Width = 40
   Height = 21
   Enabled = False
   TabOrder = 3
end
object EditRA: TEdit
   Left = 204
   Top = 92
   Width = 40
   Height = 21
   Enabled = False
   TabOrder = 14
end
object EditSX: TEdit
   Left = 72
   Top = 56
   Width = 40
   Height = 21
   Enabled = False
   TabOrder = 6
end
object EditSY: TEdit
   Left = 116
   Top = 56
   Width = 40
   Height = 21
```

```
    Enabled = False
    TabOrder = 7
  end
  object EditSZ: TEdit
    Left = 160
    Top = 56
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 8
  end
  object EditRX: TEdit
    Left = 72
    Top = 92
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 11
  end
  object EditRY: TEdit
    Left = 116
    Top = 92
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 12
  end
  object EditRZ: TEdit
    Left = 160
    Top = 92
    Width = 40
    Height = 21
    Enabled = False
    TabOrder = 13
  end
  object BitBtnOk: TBitBtn
    Left = 60
    Top = 124
```

```
   Width = 90
   Height = 25
   Enabled = False
   TabOrder = 16
   OnClick = BitBtnOkClick
   Kind = bkOK
end
object BitBtnCancel: TBitBtn
   Left = 220
   Top = 124
   Width = 80
   Height = 25
   TabOrder = 17
   Kind = bkCancel
end
object RadioGroupT: TRadioGroup
   Left = 252
   Top = 14
   Width = 97
   Height = 32
   Columns = 3
   Enabled = False
   ItemIndex = 0
   Items.Strings = (
      '1'
      '2'
      '3')
   TabOrder = 4
   OnClick = RadioGroupClick
end
object RadioGroupS: TRadioGroup
   Left = 252
   Top = 50
   Width = 97
   Height = 32
   Columns = 3
   Enabled = False
   ItemIndex = 1
```

```
      Items.Strings = (
         '1'
         '2'
         '3')
      TabOrder = 9
      OnClick = RadioGroupClick
   end
   object RadioGroupR: TRadioGroup
      Left = 252
      Top = 86
      Width = 97
      Height = 32
      Columns = 3
      Enabled = False
      ItemIndex = 2
      Items.Strings = (
         '1'
         '2'
         '3')
      TabOrder = 15
      OnClick = RadioGroupClick
   end
end
```

Figure 3-5.Parameter Data Entry Form for Transformations

The text version of the form identified the names of the event handlers, and the code for unit Trans1 in Listing 3-6 reveals the contents of the event handlers:

Listing 3-6. The Trans1 Unit.

```
unit Trans1;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
   Dialogs, StdCtrls, Buttons, OpenGL, ExtCtrls;

type
   TFormTransform = class(TForm)
      Label1: TLabel;
      Label2: TLabel;
      Label3: TLabel;
      Label4: TLabel;
      CheckBoxTranslate: TCheckBox;
      CheckBoxScale: TCheckBox;
      CheckBoxRotate: TCheckBox;
      EditTX: TEdit;
      EditTY: TEdit;
      EditTZ: TEdit;
      EditRA: TEdit;
      EditSX: TEdit;
      EditSY: TEdit;
      EditSZ: TEdit;
      EditRX: TEdit;
      EditRY: TEdit;
      EditRZ: TEdit;
      BitBtnOk: TBitBtn;
      BitBtnCancel: TBitBtn;
      RadioGroupT: TRadioGroup;
      RadioGroupS: TRadioGroup;
      RadioGroupR: TRadioGroup;
      procedure BitBtnOkClick(Sender: TObject);
      procedure CheckBoxTranslateClick(Sender: TObject);
      procedure CheckBoxScaleClick(Sender: TObject);
      procedure CheckBoxRotateClick(Sender: TObject);
      procedure RadioGroupClick(Sender: TObject);
```

```
   private
      { Private declarations }
   public
      { Public declarations }
   end;

var
   FormTransform: TFormTransform;
      transData: record
      translate,
      scale,
      rotate: boolean;
      orT,
      orS,
      orR: char;
      tx,ty,tz,
      sx,sy,sz,
      rx,ry,rz,ra: GLfloat;
   end;

implementation

{$R *.DFM}

procedure TFormTransform.BitBtnOkClick(Sender: TObject);
begin
   ModalResult := mrNone;

   with transData do
   begin
      translate := checkBoxTranslate.Checked;
      scale := checkBoxScale.Checked;
      rotate := checkBoxRotate.Checked;

      if translate then
      begin
         tx := StrToFloat(EditTX.Text);
         ty := StrToFloat(EditTY.Text);
```

```
         tz := StrToFloat(EditTZ.Text);
      end;

      if scale then
      begin
         sx := StrToFloat(EditSX.Text);
         sy := StrToFloat(EditSY.Text);
         sz := StrToFloat(EditSZ.Text);
      end;

      if rotate then
      begin
         rx := StrToFloat(EditRX.Text);
         ry := StrToFloat(EditRY.Text);
         rz := StrToFloat(EditRZ.Text);
         ra := StrToFloat(EditRA.Text);
      end;

      orT := char(48+RadioGroupT.ItemIndex);
      orS := char(48+RadioGroupS.ItemIndex);
      orR := char(48+RadioGroupR.ItemIndex);
   end; {with}

   ModalResult := mrOk;
end;

procedure TFormTransform.CheckBoxTranslateClick(Sender: TObject);

var
   chck: boolean;
begin
   with Sender as TCheckBox do
       chck := checked;
    EditTX.Enabled := chck;
    EditTY.Enabled := chck;
    EditTZ.Enabled := chck;
    RadioGroupT.Enabled := chck;
    BitBtnOk.Enabled := chck
```

```
   or EditSX.Enabled
   or EditRX.Enabled;
end;

procedure TFormTransform.CheckBoxScaleClick(Sender: TObject);
var
    chck: boolean;
begin
   with Sender as TCheckBox do
      chck := checked;
   EditSX.Enabled := chck;
   EditSY.Enabled := chck;
   EditSZ.Enabled := chck;
   RadioGroupS.Enabled := chck;
   BitBtnOk.Enabled := chck
   or EditTX.Enabled
   or EditRX.Enabled;
end;

procedure TFormTransform.CheckBoxRotateClick(Sender: TObject);
var
   chck: boolean;
begin
   with Sender as TCheckBox do
      chck := checked;
   EditRX.Enabled := chck;
   EditRY.Enabled := chck;
   EditRZ.Enabled := chck;
   EditRA.Enabled := chck;
   RadioGroupR.Enabled := chck;
   BitBtnOk.Enabled := chck
   or EditSX.Enabled
   or EditTX.Enabled;
end;

procedure TFormTransform.RadioGroupClick(Sender:TObject);
type
   TRange=0..2;
```

```
   SRange=Set of TRange;
var
   ch: char;
   Xii,
   Uii,
   Tii,
   Sii,
   Rii: integer;
   Used: SRange;
   ar: array['R'..'T'] of TRadioGroup;
   notif: TNotifyEvent;
begin
   {build set of used values to find what's left}
   Tii := RadioGroupT.ItemIndex;
   Sii := RadioGroupS.ItemIndex;
   Rii := RadioGroupR.ItemIndex;
   Used := [Tii]+[Sii]+[Rii];
   Xii := TRadioGroup(Sender).ItemIndex;

   {build array of rg for convenient searching}
   ar['T'] := RadioGroupT;
   ar['S'] := RadioGroupS;
   ar['R'] := RadioGroupR;

   {find rg not the sender but with same ItemIndex}
   for ch := 'R' to 'T' do
   with ar[ch] do
   if (ar[ch]<>Sender) and (ItemIndex=Xii) then
   begin
      for Uii := 0 to 2 do
      if not (Uii in Used) then
      begin
         notif := OnClick;
         OnClick := nil; {prevent second triggering}

      {set to unused ItemIndex and leave}
      ar[ch].ItemIndex := Uii;
      OnClick := notif;
```

```
      exit;
      end; {for if}
   end; {for with if}

end;

end.
```

The scope of this book is to teach OpenGL with Delphi, with the assumption that the reader is at least somewhat experienced in programming with Delphi. Therefore the ordinary Delphi code is presented with little comment.

Simultaneous Commands

This program still draws the same old cube, but provides for specifying two or three transformations (with independent parameters) to apply to the cube at the same time. The check boxes allow the user to specify which transformations to apply, and the radio groups allow the user to specify the order in which to apply them. The edit controls allow the user to specify the floating point values for the various parameters.

The event handler for the &Transformation menu item performs the transformations specified on the secondary form. TForm1.Transformation1Click creates a sorted string list that identifies the order in which to perform the transformation. The handler sets the matrix mode to GL_MODELVIEW and performs the transformations in the appropriate order, using the arguments stored in the transData record. The call to Invalidate insures that FormPaint fires after the transformation calls.

Save, compile, run, and experiment, not only with the values to give to each transformation, but also with the order of the transformation. Admittedly the difference is subtle for just one "object" in the scene, but many combinations will show a difference according to the sequence of the transformations. Order does matter; try it and see.

DEFINITIONS

Culling	Removing faces from the list of things to draw based on front or back designation.
Depth buffer	A list of information about the distance of pixels from the viewer.
Depth test	Determination of whether to draw pixels by comparing their distance from the viewer with that of other pixels at the same screen position.
Hidden surface removal	Not drawing surface that would otherwise be obscured from the viewer anyway.
Identity matrix	A matrix whose product with a second matrix is identical to the second matrix.
ModelView matrix	The matrix representing the coordinate system.
Origin	The point located at (0.0,0.0,0.0).
Scale	Each dimension of an object is adjusted independently by specified factors.
Transformation	Adjusting the coordinate system.
Translation	Moving an object through space.
Vector	A line with direction from a starting point toward a second point, then continuing on forever. Vectors are specified as if the first point were the origin and the second point simply sets the direction. With the first point understood to be the origin, the second point alone identifies the vector.

IDENTIFIER REFERENCE

glCullFace	An OpenGL command that specifies culling of front or back faces or both.
glEnable	An OpenGL command that allows certain features.
glLoadIdentity	An OpenGL command that sets the specified matrix

	equal to the identity matrix.
glMatrixMode	An OpenGL command that makes the specified matrix the currently active matrix.
glRotate	An OpenGL command that performs the rotation transformation.
glScale	An OpenGL command that performs the scale transformation.
glTranslate	An OpenGL command that performs the translation transformation.
GL_BACK	An OpenGL constant for glCullFace.
GL_CULL_FACE	An OpenGL constant for glEnable.
GL_DEPTH_BUFFER_BIT	An OpenGL constant for glClear.
GL_DEPTH_TEST	An OpenGL constant for glEnable.
GL_FRONT	An OpenGL constant for glCullFace.
GL_FRONT_AND_BACK	An OpenGL constant for glCullFace.
GL_MODELVIEW	An OpenGL constant for glMatrixMode.
GL_PROJECTION	An OpenGL constant for glMatrixMode.
GL_TEXTURE	An OpenGL constant for glMatrixMode.
Invalidate	A Windows function that causes Windows to send WM_PAINT and returns true on success, false on failure.

SUMMARY

Chapter 3 introduced three-dimensional rendering with these concepts:

1. Left to itself, OpenGL draws all surfaces whether they should show or not.
2. Turn on Depth Testing with glEnable to allow hidden surface removal.
3. The ModelView matrix represents the coordinate system of the scene or the coordinates of the viewer, whichever is convenient.
4. The ModelView transformations are rotation, transformation, and scale.
5. Use glRotate to rotate the scene counter-clockwise by a certain angle about a given axis.
6. Polygons have two faces. Cull back or front faces with glCullFace. Pass it GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK.

7. Determine clockwise or counter-clockwise winding of a polygon by the order of specifying the vertices. By default the front face of a polygon has counter-clockwise winding.

8. Successive transformations before painting accumulate, and changing the order of the transformations can produce different results.

9. Use the Invalidate Windows API call to insure that a window redraws.

Chapter 4

Perspective

Orthographic, or parallel, projection is fine for many purposes, but when the goal is realism in the graphics, *perspective* projection may be more appropriate. In the real world the fact that objects appear smaller when more distant is universally familiar. Artists recognize that elements of a scene appear to come together to a vanishing point at some great distance. This chapter shows how to produce this realistic effect and to position vertices in such an environment.

ILLUSTRATION

With OpenGL the programmer can construct a (you guessed it!) matrix to project a scene onto the computer screen in a manner that simulates perspective. This process is called (appropriately) perspective projection. Consider two parallel lines as in the following code.

```
glBegin(GL_LINES);
   glVertex3f(-0.5,0.1,-2.2);
   glVertex3f(-0.5,15.0,-32.2);
   glVertex3f(+0.5,0.1,-2.2);
   glVertex3f(+0.5,15.0,-32.2);
glEnd;
```

The two line segments run side by side off into the z distance, rising in the y direction. Figure 4-1 shows how they appear with perspective projection. Although reproducing this effect requires more code than given here, this is enough to show that the two lines are indeed parallel. The apparent convergence of the lines is due entirely to the perspective projection.

Figure 4-1.Parallel Lines in Perspective Projection

COMMAND

To set up the perspective projection matrix, use this command:

```
procedure glFrustum(left,right,bottom,top,zNear,zFar:GLdouble);stdcall;
```

Clipping Volume

What in the world is a frustum? Basically a frustum is a pyramid with the top chopped off. Since the clipping volume looks like a frustum with a rectangular base, lying horizontally, the command name makes sense. Figure 4-2 shows the shape of the clipping volume.

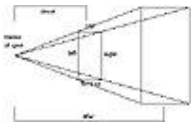

Figure 4-2.Clipping Volume from glFrustum.

Parameters

Notice that the parameters for glFrustum are the same as the parameters for glOrtho. In general these parameters have a similar usage, but there are enough differences between them to warrant caution. Just as in glOrtho, left, right, bottom, and top define a rectangle. In glOrtho, however, the front rectangle and the rear rectangle are identical except for their z value, so only one rectangle specification is sufficient. In glFrustum, the near and the far rectangles are very different sizes, but the command only takes the values for the near rectangle.

How in the world does this command define the far rectangle? The answer lies in the usage of the znear and zfar parameters. The znear value is the distance from the viewer to the near face of the frustum, which is the “cut” across the pyramid. The znear distance must be expressed as a positive number. The zfar value is the distance to the farther face of the furstum (base of the pyramid), which must be a positive number larger than znear. In effect the viewer is at the apex of the pyramid, so znear and zfar are enough to define the base of the frustum from its front face, as Figure 4-2 suggests.

DEPTH

Previous Cube

Create a directory (under the OpenGL directory, of course) and name it Chapter.4. Under that directory create a directory named Multi. Copy the files from OpenGL\Chapter.3 \Multi to OpenGL\Chapter.4\Multi. Open this project in Delphi and edit the FormCreate method in Multi.Pas. Replace

```
glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
```

with

```
glMatrixMode(GL_PROJECTION);
glFrustum(-1.0,1.0,-1.0,1.0,2.0,37.0);
```

This command change maps the same near face rectangle, but specifies the z range differently, as well as providing a different projection. The clipping volume starts at 2.0 units from the viewer and extends to 37.0 units from the viewer. In glOrtho znear and zfar just provided a range of values whose meaning was entirely up to the programmer. With parallel projection apparent sizes are not affected by distance from the viewer, so that distance is not a consideration. With perspective projection, distance from the viewer is a primary ingredient.

Compile and Run. Nothing but the green background shows because under the new coordinate system the cube was constructed entirely in front of the front clipping face. Use transformations to bring the cube into view. Fill out the parameters as in Figure 4-3, click Ok, and select Transformation from the menu.

Figure 4-3.Parameters to Bring the Cube into View

Now the cube shows, but it looks a little strange. It is like the back is larger than the front. Test this theory with a rotation transformation. Set it up as in Figure 4-4.

Figure 4-4.Parameters to Rotate the Cube

Sure enough, the back of the cube appears larger than the front. Since this is the opposite of what perspective projection should produce, something must be wrong. This is a good time to test the coordinates.

Test

Like the depth test in Chapter 3 draw two rectangles in the FormPaint method, replacing the current contents. The reason for the choice of z coordinates will soon become evident.

Save the project as Depth.Dpr in OpenGL\Chapter.4\Depth and save the main unit as Depth1.Pas in the Depth directory also. Remove the Trans1 unit from the project, remove the uses Trans1 statement from the main unit, remove the Main Menu control and empty the contents of the menu item event handlers. Adjust the form's ClientHeight back to 340. Here is what FormPaint should look like:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
   if not openGLReady then
      exit;

   {green background}
   glClearColor(0.0,0.4,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

   glBegin(GL_POLYGON); {near face}
      glColor3f(0.5,0.0,0.0);
      glVertex3f(-0.4,-0.4,-2.4);
      glVertex3f(+0.4,-0.4,-2.4);
      glVertex3f(+0.4,+0.4,-2.4);
      glVertex3f(-0.4,+0.4,-2.4);
   glEnd;

   glBegin(GL_POLYGON); {back face}
      glColor3f(0.0,0.0,0.5);
      glVertex3f(-0.2,-0.2,-2.8);
      glVertex3f(+0.6,-0.2,-2.8);
      glVertex3f(+0.6,+0.6,-2.8);
      glVertex3f(-0.2,+0.6,-2.8);
   glEnd;

   glFlush;

   errorCode := glGetError;
   if errorCode<>GL_NO_ERROR then
      raise Exception.Create('Error in Paint'#13+
         gluErrorString(errorCode));
end; {FormPaint}
```

Save, Compile, and Run. Again the red rectangle is in front of the blue rectangle. But look again at the z coordinates of the two rectangles. The farther (blue) rectangle is more negative in the z direction than the near rectangle. In the orthographic projection clipping volume a more distant point is more positive (or less negative) than a nearer point, and the specifications of the cube vertices took that fact into account. So when the same vertices appeared in a perspective projection clipping volume whose z coordinates run in the opposite direction, naturally the cube was distorted.

Better Cube

In Delphi load the previous project under Chapter.4, Multi.Dpr. Adapt the coordinates of the cube vertices to the new z direction reality. Remember that moving away from the viewer is to travel in the negative z direction, even though the znear and zfar parameters are positive and more positive respectively. Make the FormPaint method look like this:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
   if not openGLReady then
      exit;

   {green background}
   glClearColor(0.0,0.4,0.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

   glBegin(GL_POLYGON); {near face}
      glColor3f(0.5,0.0,0.0);
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(+0.4,-0.4,+0.4);
      glVertex3f(+0.4,+0.4,+0.4);
      glVertex3f(-0.4,+0.4,+0.4);
   glEnd;

   glBegin(GL_POLYGON); {right face}
      glColor3f(0.3,0.3,0.8);
      glVertex3f(+0.4,-0.4,+0.4);
      glVertex3f(+0.4,-0.4,-0.4);
      glVertex3f(+0.4,+0.4,-0.4);
      glVertex3f(+0.4,+0.4,+0.4);
   glEnd;
```

```
glBegin(GL_POLYGON); {left face}
   glColor3f(0.5,0.0,0.5);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(-0.4,-0.4,+0.4);
   glVertex3f(-0.4,+0.4,+0.4);
   glVertex3f(-0.4,+0.4,-0.4);
glEnd;

glBegin(GL_POLYGON); {back face}
   glColor3f(0.0,0.0,0.5);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(-0.4,+0.4,-0.4);
   glVertex3f(+0.4,+0.4,-0.4);
glEnd;

glBegin(GL_POLYGON); {bottom face}
   glColor3f(0.5,0.5,0.0);
   glVertex3f(-0.4,-0.4,+0.4);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(+0.4,-0.4,+0.4);
glEnd;

glBegin(GL_POLYGON); {top face}
   glColor3f(0.3,0.0,0.1);
   glVertex3f(-0.4,+0.4,+0.4);
   glVertex3f(+0.4,+0.4,+0.4);
   glVertex3f(+0.4,+0.4,-0.4);
   glVertex3f(-0.4,+0.4,-0.4);
glEnd;

glFlush;

errorCode := glGetError;
if errorCode<>GL_NO_ERROR then
   raise Exception.Create('Error in    Paint'#13+
```

```
        gluErrorString(errorCode));
end; {FormPaint}
```

For convenience, move the cube inside the clipping volume with a call to glTranslatef near the bottom of the FormCreate method. Place it there so it will only happen once.

```
   glLoadIdentity;
   glTranslatef(0.0,0.0,-2.4);

   glEnable(GL_CULL_FACE);
   glCullFace(GL_BACK);
end; {FormCreate}
```

MOVEMENT

Rotation

Set up a transformation using the parameters of Figure 4-4, then perform the rotation. Notice that since the cube is so close to the front clipping face, the rotation sends a corner right outside the clipping volume. Now it is apparent that clipping works just like in the orthographic projection clipping volume. Notice too that back face culling also works as expected. The exposed interior of the cube reveals a green nothing. Select Cull Disable from the menu and behold the interior faces of the cube.

Translation

In the parameter screen uncheck rotation and check translation. Set z to －0.1 and the rest to 0.0. Click Ok, and select Transformation from the menu repeatedly until the entire cube is inside the clipping volume. Now the image, shown in Figure 4-5, demonstrates clearly that the far side of the cube appears smaller than the near side.

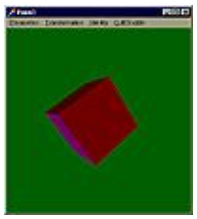

Figure 4-5.Near and Far Sides of a Cube in Perspective

Next hold down Alt+T so that the keyboard starts repeating. Watch the cube retreat into the distance. Now that's perspective!

MULTIPLE OBJECTS

Matrix Stack

Most scenes built with OpenGL consist of more than just a single object on a plain background. Building various objects in the right places is an obvious way to build more complex scenes, but it can get tedious. Suppose the scene has a number of objects that are similar to each other and the programmer has to describe them over and over. Isn't computer software supposed to relieve people of certain tiresome repetitive tasks? OpenGL provides a way!

Put a commonly used object in its own procedure or method. Build it around the origin (0.0,0.0,0.0) for ease of coding. Rotate it, translate it, even scale it, to put it in the right place, with the right orientation, and in the right shape. Then (this is important!) restore the ModelView matrix to its former condition and build and place another object.

What programming structure useful for saving and restoring something comes to mind? A stack would be nice. Indeed, as mentioned previously, OpenGL even provides a matrix stack. The commands for storing the current matrix on the stack, and retrieving it back, are:

```
procedure glPopMatrix; stdcall;
procedure glPushMatrix; stdcall;
```

Reuse an Object

Save the Mult project as Obj.Dpr in a new directory named Objects under Chapter.4. Remove Trans1.Pas from the project and save Multi.Pas as Objct.Pas in the new directory. Delete the menu object from the form, and empty all the menu item event handlers, and adjust the ClientHeight of the form back to 340.

The object of choice these days is a cube. To make it available for reuse, collect the code to draw the cube and put it in a separate method.

```
procedure TForm1.Cube;
begin
   glBegin(GL_POLYGON); {near face}
      glColor3f(0.5,0.0,0.0);
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(+0.4,-0.4,+0.4);
      glVertex3f(+0.4,+0.4,+0.4);
      glVertex3f(-0.4,+0.4,+0.4);
   glEnd;

   glBegin(GL_POLYGON); {right face}
      glColor3f(0.3,0.3,0.8);
      glVertex3f(+0.4,-0.4,+0.4);
      glVertex3f(+0.4,-0.4,-0.4);
      glVertex3f(+0.4,+0.4,-0.4);
      glVertex3f(+0.4,+0.4,+0.4);
   glEnd;

   glBegin(GL_POLYGON); {left face}
      glColor3f(0.5,0.0,0.5);
      glVertex3f(-0.4,-0.4,-0.4);
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(-0.4,+0.4,+0.4);
      glVertex3f(-0.4,+0.4,-0.4);
   glEnd;

   glBegin(GL_POLYGON); {back face}
      glColor3f(0.0,0.0,0.5);
      glVertex3f(+0.4,-0.4,-0.4);
      glVertex3f(-0.4,-0.4,-0.4);
      glVertex3f(-0.4,+0.4,-0.4);
      glVertex3f(+0.4,+0.4,-0.4);
   glEnd;

   glBegin(GL_POLYGON); {bottom face}
      glColor3f(0.5,0.5,0.0);
```

```
      glVertex3f(-0.4,-0.4,+0.4);
      glVertex3f(-0.4,-0.4,-0.4);
      glVertex3f(+0.4,-0.4,-0.4);
      glVertex3f(+0.4,-0.4,+0.4);
   glEnd;

   glBegin(GL_POLYGON); {top face}
      glColor3f(0.3,0.0,0.1);
      glVertex3f(-0.4,+0.4,+0.4);
      glVertex3f(+0.4,+0.4,+0.4);
      glVertex3f(+0.4,+0.4,-0.4);
      glVertex3f(-0.4,+0.4,-0.4);
   glEnd;
end;
```

Declare the method at the end of the private section of the form:

```
   private
      GLContext: HGLRC;
      glDC: HDC;
      errorCode: GLenum;
      openGLReady: boolean;
      oldw,
      oldh: integer;
     procedure ExceptionGL(Sender:TObject;E:Exception);
     procedure Cube;
   public
```

Make a Scene

This is enough information to build a scene with multiple copies of the cube. Make the FormPaint method look like this:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
   if not openGLReady then
      exit;

   {background}
   glClearColor(0.6,0.8,0.8,0.0);
```

```
   glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

   glPushMatrix;
   glTranslatef(-0.7,0.7,-2.8);
   Cube;
   glPopMatrix;

   glPushMatrix;
   glTranslatef(1.9,-1.6,-5.4);
   Cube;
   glPopMatrix;

   glFlush;

   errorCode := glGetError;
   if errorCode<>GL_NO_ERROR then
      raise Exception.Create('Error in Paint'#13+
         gluErrorString(errorCode));
end; {FormPaint}
```

Notice the sequence: push the ModelView matrix, translate to the desired location, call the routine that builds the object about the origin, and pop the matrix. Do this for each object in the scene; even if a given object is not repeated, it is usually simpler to construct at the origin, especially if it has symmetry. In this simple example only a translation appears before each cube, but any combination the three transformations could take its place.

Save, Compile, and Run. Behold a 3-D, two-object scene! Notice the marvelous new background color. Perhaps its name should be "Clear Blue Sky with Light Smog." Figure 4-6 shows how it looks. The new color, though far from pretty, provides better contrast for the cubes.

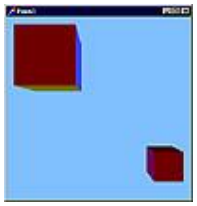

Figure 4-6.Two Cubes in Perspective

INDEPENDENT MOVEMENT

The techniques for creating multiple copies of an object and placing them in different places also lend themselves to moving the objects separately. To demonstrate, add some fields to the private section of the form type declaration:

```
x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,
xr2,yr2,zr2,angle1,angle2,sign: glFloat;
```

Initialize sign at the very bottom of the FormCreate method:

```
sign := 0.1;
```

Translate

In the FormPaint method replace

```
glTranslatef(-0.7,0.7,-2.8);
```

with

```
glTranslatef(x1,y1,z1);
```

Also replace

```
glTranslatef(1.9,-1.6,-5.4);
```

with

```
glTranslatef(x2,y2,z2);
```

Now that the arguments in the glTranslatef calls are variables, code to manipulate the variables is appropriate. Using the Object Inspector set the form's KeyPreview property to true. On the Events page double click the OnKeyPress event and fill in the following event handler:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
var
   ok: boolean;
begin
   ok := true;
   case Key of
      '+': sign := 0.1;
      '-': sign := -0.1;
```

```
      'X': x1 := x1+sign;
      'x': x2 := x2+sign;
      'Y': y1 := y1+sign;
      'y': y2 := y2+sign;
      'Z': z1 := z1+sign;
      'z': z2 := z2+sign;
      else
         ok := false;
   end; {case}
   if ok then
       Invalidate;
end;
```

Whenever the user executes a keystroke the FormKeyPress method tests the character in the case statement and adjusts the variables accordingly. Pressing "+" or "-" sets the sign variable accordingly. The letters "x", "y", and "z" set the corresponding variables, with the capital letters setting x1, etc., and the lowercase letters setting x2, etc. If a desired letter came through, then a call to Invalidate causes the FormPaint method to fire again.

The sign variable starts with a positive value of one tenth. Pressing "-" changes it to negative one tenth and pressing "+" changes it back. This value adds to the other variables in response to the appropriate letter. Save, Compile, and Run. Start by pressing the minus key, then pressing and holding a capital "Z" until a cube comes into view. Since the variables are fields of the form, they start as 0.0 unless explicitly changed. That means both cubes remain out of site at the origin until the user gives them a lot of negative z. Capital letters affect one cube and lower case letters affect the other cube. Experiment with pressing the desired keys and enjoy the power of moving the two cubes independently.

Rotate

Being able to move two cubes independently represents a definite advance in the available OpenGL power, but rotating the cubes separately is an even greater advance. Better yet is the power to combine rotation and translation for each cube (independently). The variables have already been declared. After each glTranslatef goes a glRotatef:

```
   glTranslatef(x1,y1,z1);
   glRotatef(angle1,xr1,yr1,zr1);
```

```
glTranslatef(x2,y2,z2);
glRotatef(angle2,xr2,yr2,zr2);
```

The case statement in the FormKeyPress method needs an addition so that it can manipulate the rotation variables. The letters to check are upper and lower case "r," "s," and "t."

```
'a': angle2 := angle2+10.0*sign;
'r': xr2 := xr2+sign;
's': yr2 := yr2+sign;
't': zr2 := zr2+sign;
'A': angle1 := angle1+10.0*sign;
'R': xr1 := xr1+sign;
'S': yr1 := yr1+sign;
'T': zr1 := zr1+sign;
```

The letters "r," "s," and "t" set up the axis of rotation for each cube. Each axis is a *vector* (line segment with direction) from (0.0,0.0,0.0) toward the vertex constructed from (xr1,yr1,yr1) or (xr2,yr2,zr2). The letter "a" adjusts the amount of rotation (angle1 or angle2) about the appropriate vector. It would be nice to have a little help visualizing the axes of rotation. Add the following method to draw a representation of the coordinate axes along with the two rotation vectors:

```
procedure TForm1.Axes;
begin
   glLineWidth(3.0);
      glBegin(GL_LINES);
      glColor3f(0.0,0.0,0.0);

      glVertex3f(+0.0,+0.0,+0.4);
      glVertex3f(+0.0,+0.0,-0.4);

      glVertex3f(+0.4,+0.0,+0.0);
      glVertex3f(-0.4,+0.0,+0.0);

      glVertex3f(+0.0,+0.4,+0.0);
      glVertex3f(+0.0,-0.4,+0.0);
   glEnd;

   glLineWidth(1.0);
   glBegin(GL_LINES);
      glColor3f(1.0,1.0,0.0);
```

```
      glVertex3f(0.0,0.0,0.0);
      glVertex3f(xr1,yr1,zr1);

      glColor3f(0.0,1.0,1.0);
      glVertex3f(0.0,0.0,0.0);
      glVertex3f(xr2,yr2,zr2);
   glEnd;
end;
```

Declare the new method at the bottom of the form's private section:

```
      procedure ExceptionGL(Sender:TObject;E:Exception);
      procedure Cube;
      procedure Axes;
   public
      { Public declarations }
   end;
```

Call the Axes method inside the FormPaint method, just before the call to glFush. Translate it along the z axis to bring it inside the clipping volume. Translate it a little along the x and y axes so that the viewer will see it slightly from above and from the side, to make its three-dimensional nature more evident.

```
   glPushMatrix;
   glTranslatef(-0.6,-0.6,-2.8);
   Axes;
   glPopMatrix;

   glFlush;
```

Save, Compile, and Run. In addition to all the previous abilities of the program, it can now use the three new letters to rotate the cubes. Whew!

Transformation Order Revisited

About the only thing left for this structure is to add scaling as well, but that would not add much to the OpenGL knowledge because the results are easy to predict. There is something else that can contribute significantly to the level of OpenGL understanding. The ability to switch back and forth between rotating before translation and rotating after translation reveals a dramatic difference between the two. Add the following to the case statement:

```
'B': before := true;
'b': before := false;
```

The form declaration needs to include the before field:

```
errorCode: GLenum;
before,
openGLReady: boolean;
oldw,
oldh: integer;
x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,
xr2,yr2,zr2,angle1,angle2,sign: glFloat;
procedure ExceptionGL(Sender:TObject;E:Exception);
procedure Cube;
procedure Axes;
```

FormPaint needs an enhancement using the value of the before field to control whether the call to glRotatef happens before or after the call to glTranslatef:

```
glPushMatrix;
if before then
   glRotatef(angle1,xr1,yr1,zr1);
glTranslatef(x1,y1,z1);
if not before then
   glRotatef(angle1,xr1,yr1,zr1);
Cube;
glPopMatrix;

glPushMatrix;
if before then
   glRotatef(angle2,xr2,yr2,zr2);
glTranslatef(x2,y2,z2);
if not before then
   glRotatef(angle2,xr2,yr2,zr2);
Cube;
glPopMatrix;
```

Again Save, Compile, and Run. Bring the cubes into view and position and orient them as you please, but be sure to include some rotation for each of them. Now press “B” and

"b" to show the difference between rotation before translation and rotation after translation. The experiment with order of transformations at the end of Chapter 3 gave small distinctions, but this time the difference is exceedingly obvious. For example, a small rotation about the y axis changes the direction of the cube's own personal z axis, so a subsequent translation along the z axis sends it off in a different direction than if the translation happens first. Try it and see.

Another View

Frankly, transformations in OpenGL can be very confusing, so any aid to comprehension is welcome. For this purpose think of the translations as applying to the objects rather than the coordinate system. Then think of the transformations as happening in reverse order. When you push and pop the ModelView matrix several times, treat this reverse order concept separately between each push and pop.

Different people may need to consider OpenGL transformations in different ways to understand them. Possibly this alternate view will be the best one for many readers of this book.

DEFINITIONS

Frustum	A geometric figure formed when the top of a cone or pyramid is cut off by a plane parallel to the base.
Perspective projection	A projection in which the apparent size of each object is adjusted for distance from the viewer.

IDENTIFIER REFERENCE

glFrustum	An OpenGL command that sets up perspective projection and the clipping volume.
glPopMatrix	An OpenGL command that saves the current OpenGL matrix on the matrix stack.
glPushMatrix	An OpenGL command that retrieves a matrix from the matrix stack and makes it the current OpenGL matrix.

SUMMARY

Here are the high points of Chapter 4:

1. The shape of the clipping region for perspective projection is a frustum, which is a pyramid with the top chopped off.
2. The command is glFrustum; the parameters are the same as for glOrtho, but the znear and zfar parameters have different effects and must be positive.
3. To move away from the viewer, deeper into the screen, is to move in the negative z direction.
4. The effects of clipping and back face culling are the same as with glOrtho.
5. When viewed in perspective, parallel lines appear to converge in the distance, and objects appear smaller as they retreat into the distance.
6. Use glPushMatrix to save the ModelView matrix and glPopMatrix to restore it. Placing transformations between those calls allows for independent positioning and movement of objects in a scene.
7. Rotation changes the orientation of the axes so that a subsequent translation may go off in a different direction.When planning a positioning or movement within a scene, be careful about the order of the transformations.
8. OpenGL transformations seem to take effect in the reverse order of their specification in program code.

Chapter 5

Lighting

In the real world things are visible because of light. All light comes from various sources, usually identifiable, but even when the source is unknown, the light was actually emitted from somewhere. Not only does the visibility of everything (except those objects that emit their own light) depend on light emitted from somewhere and reflected to the viewer, but the detailed appearance of each object depends on properties of the material composing the illuminated surface. A realistic graphics system should incorporate these facts. This chapter explores lighting and material properties.

Building on the work of Chapter 4, Save Obj.Dpr as ALight.Dpr in OpenGL\Chapter.5\Ambient. Save the main unit as ALight1.Pas.

ENABLE LIGHTING

One of the great features of OpenGL is the ability to produce realistic effects through simulation of lighting. In the code in the first four chapters, OpenGL provided the equivalent of lighting so that the objects were visible. Now is the time to provide lighting specifications directly. Here is a necessary OpenGL constant:

```
GL_LIGHTING = $0B50;
```

At the bottom of the FormCreate method enable lighting:

```
glEnable(GL_LIGHTING);
```

Save, Compile and Run. Maneuver one of the cubes into the viewing volume (clipping region). Ugh! The red square is black! Rotate the cube so that two more faces show at least a little. They are black, too! It is difficult to distinguish one face from another because the have exactly the same color. Now that the program is taking responsibility for making the objects in the scene visible, OpenGL is taking no responsibility at all. Obviously the program needs to do more than just enable lighting.

Basically three kinds of lighting are available through OpenGL, *Ambient*, *Diffuse*, and *Specular*. This chapter will examine each of them in turn.

AMBIENT LIGHT

Definition

Ambient light has no apparent source, but seems to come from all directions. In the real world, all light has a source, but some light may be reflected and scattered from several surfaces before finally reaching the viewer. The source of such light may be difficult or impossible to trace. This kind of light is called ambient light. OpenGL simulates this effect by providing illumination equally from all directions. It can be associated with a particular light or it can have no specific source, just magically appearing (at the programmer's command).

Commands and Constants

The command for defining a light is a variation of glLight. Use the "f" suffix to use glFloat parameters. Many commands, including ones already discussed, have a vector version, indicated by a "v" suffix. The vector version substitutes a pointer to an array in place of several scalar parameters. The glLight command only takes three parameters in any version, so only the vector version meets the current need. The glLightModel command takes three parameters and also has a vector version suitable for the occasion. Then there is the glColorMaterial command, which introduces the idea of material properties. The relevant declarations follow. First a necessary type:

```
PGLfloat =  ^GLFloat;
```

Here are some useful constants:

```
GL_LIGHT_MODEL_AMBIENT = $0B53;
GL_COLOR_MATERIAL      = $0B57;
GL_AMBIENT             = $1200;
GL_LIGHT0              = $4000;
GL_LIGHT1              = $4001;
GL_LIGHT2              = $4002;
GL_LIGHT3              = $4003;
GL_LIGHT4              = $4004;
GL_LIGHT5              = $4005;
GL_LIGHT6              = $4006;
GL_LIGHT7              = $4007;
```

Finally, the commands:

```
procedure glColorMaterial(face,mode:GLenum); stdcall;
```

```
procedure glLightModelfv(pname:GLenum;params:PGLfloat); stdcall;
procedure glLightfv(light,pname:GLenum;params:PGLfloat); stdcall;
```

Consider the parameters in the commands. The face parameter can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK. Right now GL_FRONT is sufficient. The mode parameter specifies the kind of lighting to which the material property relates. The example under development uses GL_AMBIENT. The params pointer could be pointing to a single glFloat, or the first element of an array of them. While this is very C-like, Delphi can handle it nicely. The pname parameter identifies the purpose for two of the commands. The command for the simplest method of implementing ambient light is glLightModel. Pass it GL_LIGHT_MODEL_AMBIENT for the first parameter.

THE EASY WAY

Lighting introduces a new level of complexity to the specification of an object's appearance. A simple color specification does not tell the whole story about how a surface responds to light. *Material* properties allow a more complete way to describe responses to light. The present example avoids some of the complexity by having the material color track the current color by means of the glColorMaterial command. This is one of the options that requires enabling before use.

At the bottom of the FormCreate method, after the enabling of lighting , initialize the array and place some of the new commands with the appropriate arguments.

```
AmbientDef[0] := 1.0; {red}
AmbientDef[1] := 1.0; {green}
AmbientDef[2] := 1.0; {blue}
AmbientDef[4] := 1.0; {alpha}
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT,GL_AMBIENT);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,@AmbientDef);
```

In this version of the glLightModel command, using GL_LIGHT_MODEL_AMBIENT, means the array contains the color specification of the ambient light. Here, giving maximum intensity (1.0) to the red, green, and blue components specifies a white ambient light. A later chapter covers alpha; just set it to 1.0 for now. Declare the array in the private section of the form.

```
AmbientDef: array[0..3] of glFloat;
```

Save, Compile, and Run. Now the cubes look like they did before. So what did the new commands accomplish? One benefit is control. Experiment with decreasing the red, green, and blue components of AmbientDef (positions 0, 1, and 2 in the array). Reducing the overall intensity of the ambient light or giving it a color bias produces effects that may be desirable in some applications.

THE HARD WAY

A more flexible approach uses the glLight command. Make a new directory named OpenGL\Chapter.5\Ambient0. In that directory save the program as Amb0.Dpr and the main unit as Ambient0.Pas. Change the bottom of the FormCreate method to this:

```
   glEnable(GL_LIGHTING);
   AmbientDef[0] := 1.0; {red}
   AmbientDef[1] := 1.0; {green}
   AmbientDef[2] := 1.0; {blue}
   AmbientDef[3] := 1.0; {alpha}

   glLightfv(GL_LIGHT0,GL_AMBIENT,@AmbientDef);
   glEnable(GL_LIGHT0);
```

This approach defines a particular light (LIGHT0) instead of just setting up a light model. At least eight lights are available; some implementations of OpenGL may allow more. Notice that you must turn on any light you want to use. This example switches on LIGHT0 with glEnable(GL_LIGHT0). To use a light, the simple color tracking of glColorMaterial is not adequate. So instead of glColor commands inside the Cube method, use glMaterial commands. A vector version is most convenient, so define an array at the top of Cube. For this example there is no attempt to produce the same colors as before.

```
procedure TForm1.Cube;
var
  ambient: array[0..3] of glFloat;
begin
  fillchar(ambient,sizeOf(ambient),0);
  ambient[3] := 1.0; {alpha}
  glBegin(GL_POLYGON); {near face}
     ambient[0] := 0.5; {red}
     glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
     glVertex3f(-0.4,-0.4,+0.4);
     glVertex3f(+0.4,-0.4,+0.4);
     glVertex3f(+0.4,+0.4,+0.4);
     glVertex3f(-0.4,+0.4,+0.4);
  glEnd;

  glBegin(GL_POLYGON); {right face}
```

```
   ambient[1] := 0.5; {green} {total=yellow}
   glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
   glVertex3f(+0.4,-0.4,+0.4);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(+0.4,+0.4,-0.4);
   glVertex3f(+0.4,+0.4,+0.4);
glEnd;

glBegin(GL_POLYGON); {left face}
   ambient[0] := 0.0; {red} {total=green}
   glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(-0.4,-0.4,+0.4);
   glVertex3f(-0.4,+0.4,+0.4);
   glVertex3f(-0.4,+0.4,-0.4);
glEnd;

glBegin(GL_POLYGON); {back face}
   ambient[2] := 0.5; {blue} {total=turquoise}
   glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(-0.4,+0.4,-0.4);
   glVertex3f(+0.4,+0.4,-0.4);
glEnd;

glBegin(GL_POLYGON); {bottom face}
   ambient[1] := 0.0; {green} {total=blue}
   glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
   glVertex3f(-0.4,-0.4,+0.4);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(+0.4,-0.4,+0.4);
glEnd;

glBegin(GL_POLYGON); {top face}
   ambient[0] := 0.5; {red} {total=magenta/magenta}
   glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
```

```
    glVertex3f(-0.4,+0.4,+0.4);
    glVertex3f(+0.4,+0.4,+0.4);
    glVertex3f(+0.4,+0.4,-0.4);
    glVertex3f(-0.4,+0.4,-0.4);
  glEnd;
end;
```

Save, Compile, and Run. Bring the cubes into view and move them around. Notice something rather strange. A side of the cube facing directly toward the viewer is brighter than the same side with a significantly different angle to the viewer.

That is not the behavior of ambient light. Perhaps it has been set too bright. Ambient light is often a minor part of the total light in a scene. In the FormCreate method cut it down as follows:

```
AmbientDef[0] := 0.3; {red}
AmbientDef[1] := 0.3; {green}
AmbientDef[2] := 0.3; {blue}
AmbientDef[3] := 1.0; {alpha}
```

Save, Compile, and Run. Manipulate the scene as before. Alas, there is little noticeable difference between ambient light at 30% and ambient light at 100%. Something is very wrong.

Actually, LIGHT0 has a special characteristic different from LIGHT1, LIGHT2 or any of the others. LIGHT0 by default has a white light diffuse component and a white light specular component. The default ambient component is black, but the present code replaced the default value. To see the effect of ambient light alone requires replacing the diffuse and specular parts (with black). Make these changes to the FormCreate method:

```
glEnable(GL_LIGHTING);
fillchar(AmbientDef,SizeOf(AmbientDef),0);
glLightfv(GL_LIGHT0,GL_DIFFUSE,@AmbientDef);
glLightfv(GL_LIGHT0,GL_SPECULAR,@AmbientDef);

AmbientDef[0] := 0.3; {red}
AmbientDef[1] := 0.3; {green}
AmbientDef[2] := 0.3; {blue}
AmbientDef[3] := 1.0; {alpha}
```

```
glLightfv(GL_LIGHT0,GL_AMBIENT,@AmbientDef);
glEnable(GL_LIGHT0);
```

The new constants are:

```
GL_DIFFUSE  = $1201;
GL_SPECULAR = $1202;
```

Now Save, Compile, and Run. Bring the cubes into view. What a difference! The cubes are rather dim, which is natural for only 30% ambient illumination, but most importantly, they are uniformly dim. This is how ambient light should appear!

Can OpenGL handle bright white ambient light, or did the earlier supposition that the ambient light was set out of natural range have merit? Find out by replacing all the 0.3 values in AmbientDef with 1.0. Save, Compile, Run, and move cubes around. Now uniform illumination abounds. Though it is quite bright, the ambient light appears evenly distributed regardless of angle, which is the right behavior for ambient light.

DIFFUSE LIGHT

Definition

Imagine a lamp lighting a large room. Much of the light bounces off many surfaces before reaching a viewer. While the light actually came from the lamp, it seems simply to be everywhere. This is the ambient portion of the light. Obviously, however, some of the light proceeds directly from the source to the surface of some object in the room and reflects directly to the viewer. However, unless the surface is a mirror, the light tends to scatter rather than bounce off at a predictable angle. In the real world nothing is perfectly smooth, but some things have been polished enough that the surface irregularities are too small to matter. Such surfaces are "shiny." Most surfaces have irregularities large enough to scatter most light as in Figure 5-1. These surfaces are "flat." Unless the surface is white, it will absorb some colors from the light and reflect the rest, thus defining its color in terms of the portion of the light it reflects. This scattering effect determines the diffuse portion of the light. Not only does OpenGL allow the definition of at least eight light sources, but it also allows independent specification of the ambient portion and diffuse portion of each light.

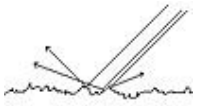

Figure 5-1.Diffuse Reflection from a Flat Surface

Positional Lamps

Save the project as Difu.Dpr in a new directory called OpenGL\Chapter.5\Dif1 and save the main unit as Diffus.Pas. For the lamp under development, most light shines directly on an object from the lamp (the diffuse portion), and a lesser contribution comes from multiple reflections (the ambient portion). Prepare to change the AmbientDef assignments in the FormCreate method as follows, but do not type it in just yet:

```
glEnable(GL_LIGHTING);
AmbientDef[0] := 0.3; {red}
AmbientDef[1] := 0.3; {green}
AmbientDef[2] := 0.3; {blue}
AmbientDef[3] := 1.0; {alpha}
```

While bland, homogenized arrays like this provide a simple, fairly language-independent way of passing data to the OpenGL DLL's, Delphi allows more meaningful structures

that use memory exactly the same way. The following type declaration of a record structure is more intuitive and does not require the programmer to memorize the order of the colors:

```
   TLightDesc=record
      red,
      green,
      blue,
      alpha: glFloat;
   end;
```

Wait! Don't type that in either! Arrays use contiguous memory, but record structures may have "holes" in the data due to memory alignment by the compiler. This particular record structure is safe (for the moment) because each of the elements is four bytes in size, but to guarantee that the record memory layout is exactly the same as the array memory layout, use the packed reserved word and place the declaration just above the form's class declaration:

```
type
   TLightDesc=packed record
      red,
      green,
      blue,
      alpha: glFloat;
   end;
TForm1 = class(TForm)
```

Of course the type for the AmbientDef field within the form declaration should become:

```
      AmbientDef: TLightDesc;
```

Now the new assignments to AmbientDef in the FormCreate method can proceed:

```
   glEnable(GL_LIGHTING);
   with AmbientDef do
   begin
      red := 0.3;
      green := 0.3;
      blue := 0.3;
      alpha := 1.0;
```

```
   end;
```

The key distinction between diffuse light and ambient light is that the source of diffuse light has a position. Therefore a demonstration of diffuse light requires the ability to set the position of the lamp. Accordingly add these lines to the case statement in the FormKeyPress method:

```
      'l': xl := xl+sign;
      'm': yl := yl+sign;
      'n': zl := zl+sign;
```

In the form’s class declaration add these “L” variable declarations to describe the lamp position:

```
      xl,yl,zl,
      x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,
      xr2,yr2,zr2,angle1,angle2,sign: glFloat;
      AmbientDef: TLightDesc;
      LightPlace: TLightPlace;
```

LightPlace is also new and needs a type declaration.

```
type
   TLightDesc=packed record
      red,
      green,
      blue,
      alpha: glFloat;
   end;
   TLightPlace=packed record
      x,
      y,
      z,
      w: glFloat;
   end;
TForm1 = class(TForm)
```

In the FormPaint method, place some code to manipulate the light position:

```
   glPushMatrix;
```

```
   if before then
     glRotatef(angle2,xr2,yr2,zr2);
   glTranslatef(x2,y2,z2);
   if not before then
     glRotatef(angle2,xr2,yr2,zr2);
   Cube;
   glPopMatrix;
   glPushMatrix;
   glTranslatef(xl,yl,zl);
   glLightfv(GL_LIGHT0,GL_POSITION,@LightPlace);
   glPopMatrix;
```

Save, Compile, Run, and bring a cube barely into view, then go two keystrokes more down the negative z-axis. Move the cube near the upper left corner of the form. The cube appears very bright, but that will change as the lamp moves.

Press the minus key, then press the lowercase “n” repeatedly. After about five such presses, a slight difference is noticeable. As you continue to advance the lamp down the negative z-axis, watch the shading develop on the cube!

Now it should be possible to have a uniformly colored cube that still shows its three-dimensional nature by its shading in a diffuse light source. In the Cube method change the type of the ambient local variable:

```
procedure TForm1.Cube;
var
   ambient: TLightDesc;
begin
```

Remove all assignements to ambient inside the Cube method and replace them with one set of assignments as follows:

```
procedure TForm1.Cube;
var
   ambient: TLightDesc;
begin
   ambient.red := 0.2;
   ambient.green := 0.5;
   ambient.blue := 0.8;
   ambient.alpha := 1.0;
```

```
glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
glBegin(GL_POLYGON); {near face}
```

Save, Compile, Run, and perform movements as before. Around the 22nd "n" press the shading of the cube is more than sufficient to distinguish the faces. Enjoy! This is tremendous progress from the first step of making a colored background.

Different Material Colors

Various material properties can be set independently, including colors for ambient reflectivity, specular reflectivity, and diffuse reflectivity. Allowing the surface to reflect ambient light differently from diffuse light provides for some interesting effects. Adjust the top of the Cube method as follows:

```
procedure TForm1.Cube;
var
  diffuse,
  ambient: TLightDesc;
begin
  ambient.red := 0.2;
  ambient.green := 0.5;
  ambient.blue := 0.8;
  ambient.alpha := 1.0;
  diffuse.red := 0.8;
  diffuse.green := 0.5;
  diffuse.blue := 0.2;
  diffuse.alpha := 1.0;
  glMaterialfv(GL_FRONT,GL_DIFFUSE,@diffuse);
  glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
  glBegin(GL_POLYGON); {near face}
```

Save, etc. as before. This version more clearly demonstrates the ambient contribution and the diffuse contribution from the light source. At first the diffuse light reflection color clearly dominates. As the light advances down the negative z-axis until it is almost past the near face of the cube, ambient reflection clearly shows one color, and diffuse reflection another color. When the light moves a little more, the ambient reflection color will completely dominate. Why does the diffuse contribution wane before the light is past the cube? Another behavior of real light reflecting off real surfaces is the fact that as the angle from the light source to the surface flattens, less of the diffuse light reflects to the viewer. Figure 5-2 illustrates light shining on a surface from an acute angle (left) and from a more obtuse angle (right). The right side of the illustration shows less of the scattered (diffuse) light reaching the viewer. OpenGL reproduces this behavior nicely.

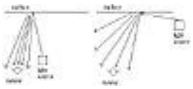

Figure 5-2.Diffuse Light at Different Angles.

So far the exercises have depended on the fact that LIGHT0 has a white diffuse component by default. The next exercise illustrates colored light on surfaces of different color. Accordingly, at the bottom of the FormCreate method, make the light pure red and explicitly define the diffuse component.

```
  glEnable(GL_LIGHTING);
  with AmbientDef do
  begin
    red := 0.6;
    green := 0.0;
    blue := 0.0;
    alpha := 1.0;
  end;
  LightPlace.w := 1.0;

  glLightfv(GL_LIGHT0,GL_AMBIENT,@AmbientDef);
  glLightfv(GL_LIGHT0,GL_DIFFUSE,@AmbientDef);
  glEnable(GL_LIGHT0);
end; {FormCreate}
```

Save ⋯ well, you know the drill. Now, since the only light is red, that is the only light reflected, so the cube looks pure red. What will happen if the material property has no red component? Find out by removing the red at the top of the Cube method. At the same time return the cube to having only one color response to the light.

```
procedure TForm1.Cube;
var
  ambient: TLightDesc;
begin
  ambient.red := 0.0;
  ambient.green := 0.5;
  ambient.blue := 0.8;
  ambient.alpha := 1.0;
  glMaterialfv(GL_FRONT,GL_DIFFUSE,@ambient);
  glMaterialfv(GL_FRONT,GL_AMBIENT,@ambient);
  glBegin(GL_POLYGON); {near face}
```

Repeat the steps and see the result. The cube is black. Only red light is available. The cube reflects no red light. Therefore the cube reflects no virtually light.

Spotlights

The light source discussed so far can be a spotlight. First, this naked light needs some clothes. For this stage of the learning adventure, dress it in a ⋯ cube. Save the project as Spot.Dpr in OpenGL\Chapter.5\Spot, and save the main unit as Spot1.pas in that directory. Give the Cube method the ability to set its material color from a passed parameter. Adjust its declaration inside the form class declaration and add a few fields.

```
  CubeColor,
  SpotColor,
  DiffDef,
  AmbientDef: TLightDesc;
  LightPlace: TLightPlace;
  procedure ExceptionGL(Sender:TObject;E:Exception);
  procedure Cube(matColor:TLightDesc);
  procedure Axes;
```

Now change its implementation code to handle the parameter.

```
procedure TForm1.Cube(matColor:TLightDesc);
begin
  glMaterialfv(GL_FRONT,GL_AMBIENT,@matColor);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,@matColor);
  glBegin(GL_POLYGON); {near face}
```

Set the color records in the FormCreate method.

```
  glEnable(GL_LIGHTING);
  with AmbientDef do
  begin
    red := 0.3;
    green := 0.3;
    blue := 0.3;
    alpha := 1.0;
  end;
  with DiffDef do
  begin
    red := 1.0;
    green := 1.0;
    blue := 1.0;
```

```
    alpha := 1.0;
  end;
  with CubeColor do
  begin
    red := 0.5;
    green := 0.0;
    blue := 0.8;
    alpha := 1.0;
  end;
  with SpotColor do
  begin
    red := 0.0;
    green := 0.5;
    blue := 0.5;
    alpha := 1.0;
  end;

  LightPlace.w := 1.0;

  glLightfv(GL_LIGHT0,GL_AMBIENT,@AmbientDef);
  glLightfv(GL_LIGHT0,GL_DIFFUSE,@DiffDef);
```

Make the two calls to Cube in the FormPaint method use the parameter.

```
  Cube(CubeColor);
```

Also adjust the code that positions the light Be sure to make this cube smaller with glScalef.

```
  glPushMatrix;
  glScalef(0.33,0.33,0.33);
  glTranslatef(xl,yl,zl);
  glLightfv(GL_LIGHT0,GL_POSITION,@LightPlace);
  Cube(SpotColor);
  glPopMatrix;
```

Now, go back to the bottom of the FormCreate method and initialize some positions so you will not have to do so much work to position the objects. Note that the "l" in the variable names is an ell not a one.

```
  z2 := -2.7;
  x2 := -0.7;
  y2 := 0.7;
  zl := -6.7;
  xl := -1.8;
  yl := -2.1;
end; {FormCreate}
```

Since the glScale command affects translation (and rotation), as well as the object size, adjust the case statement in the FormKeyPress method.

```
      'l': xl := xl+3.0*sign;
      'm': yl := yl+3.0*sign;
      'n': zl := zl+3.0*sign;
```

Save, Compile, and Run. (You are remembering not to run from the IDE, aren't you?) The magenta (or purple) cube and the cyan (blue-green, turquoise, whatever) "spotlight" are visible from the start. Use the "l," "m," and "n" keys to move the spotlight around. The center of the small cyan cube tracks the position of the light source so that the cube appears to BE the light source. Notice the effect different light positions have on illumination of the magenta cube.

A really interesting effect is the illumination of the small cube from the inside, so that the sides of the cube appear to be translucent. This brings up an important fact. OpenGL knows nothing about shadows. When you "shine" an OpenGL light source on an OpenGL object, you see a very good representation of real illumination. However, if you interpose another seemingly solid OpenGL object between the light source and the first object, the more distant object receives illumination just as if there were nothing in the way. The consequence of this fact is that if you want shadows, you will have to program them yourself. Do not despair; it can be done. More about that comes later.

Notice that only half of the cyan cube lights up. By default an OpenGL diffuse light source has a 180 degree spread as shown in figure 5-3, which includes the cyan cube that represented the spotlight. A real spotlight is more focused, and OpenGL permits this effect with an appropriate constant in a familiar command.

```
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,cutoff);
```

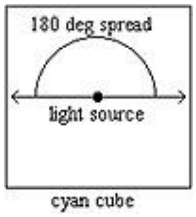

Figure 5-3.Default Spread of a Diffuse Light Source.

The new constant has this value:

```
GL_SPOT_CUTOFF = $1206;
```

Express the value of the cutoff variable in degrees. Think of the light spreading from a spotlight as a cone of light. The angle from the center of the cone to the edge of the cone is the cutoff angle, as shown in figure 5-4.

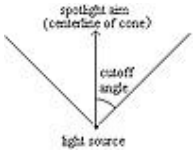

Figure 5-4.The Spotlight Cutoff Angle.

Add the cutoff field to the "ell" line in the form's class declaration.

```
    xl,yl,zl,xrl,yrl,zrl,cutoff,
    x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,
    xr2,yr2,zr2,angle1,angle2,sign: glFloat;
```

Use the cutoff field in the FormCreate method.

```
glLightfv(GL_LIGHT0,GL_AMBIENT,@AmbientDef);
glLightfv(GL_LIGHT0,GL_DIFFUSE,@DiffDef);
cutoff := 55.0;
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,cutoff);
glEnable(GL_LIGHT0);
```

Save, Compile, Run. Move the spotlight around with "l," "m," and "n." Notice that you must move the spotlight closer to the magenta cube in the x and y directions or farther in the positive z direction in order to illuminate the magenta cube.

Directional Spotlights

A feature still missing from the spotlight illustrated thus far in the chapter is the ability to aim it. What kind of spotlight is it that you can't ⋯ spot? One thing needed for that purpose is to add some more "ell" fields in the form class declaration.

```
xl,yl,zl,xrl,yrl,zrl,anglel,cutoff,
x1,y1,z1,x2,y2,z2,xr1,yr1,zr1,
xr2,yr2,zr2,angle1,angle2,sign: glFloat;
```

The xrl, yrl, and zrl fields set up the axis of rotation for the cyan cube and the aiming direction of the spotlight. Similarly the anglel field is the number of degrees of rotation about the axis for both the cube and the aiming of the spotlight. The default aim of a spotlight is a vector from the origin (0.0,0.0,0.0) to (0.0,0.0,-1.0) down the negative z-axis, which points toward the far face of the cube at its starting position. Therefore aiming the far face of the cube effectively aims the spotlight.

To manipulate those fields, add to the case statement in the FormKeyPress method.

```
'k': anglel := anglel+30.0*sign;
'o': xrl := xrl+3.0*sign;
'p': yrl := yrl+3.0*sign;
'q': zrl := zrl+3.0*sign;
```

Add a new type:

```
TLightDesc=packed record
  red,
  green,
  blue,
  alpha: glFloat;
end;
TLightPlace=packed record
  x,
  y,
  z,
  w: glFloat;
end;
TPosition=packed record
  x,
```

```
    y,
    z: glFloat;
  end;
```

Add a new field of that type in the form’ s class declaration:

```
    AmbientDef: TLightDesc;
    LightPlace: TLightPlace;
    SpotDirect: TPosition;
```

Initialialize SpotDirect in the FormCreate method to match the default value of a spotlight direction. Except for the z field, the zeros that come automatically from creating an instance of the form are sufficient.

```
with SpotColor do
begin
  red := 0.0;
  green := 0.5;
  blue := 0.5;
  alpha := 1.0;
end;
SpotDirect.z := -1.0;
LightPlace.w := 1.0;
```

Finally, to the FormPaint method add the use of SpotDirect and the rotational “ell” fields.

```
glPushMatrix;
glScalef(0.33,0.33,0.33);
if before then
  glRotatef(anglel,xrl,yrl,zrl);
glTranslatef(xl,yl,zl);
if not before then
  glRotatef(anglel,xrl,yrl,zrl);
glLightfv(GL_LIGHT0,GL_POSITION,@LightPlace);
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,@SpotDirect);
Cube(SourceColor);
```

Save, Compile, Run, and press the plus key. Press the lower-case “o” to set the axis of rotation for the spotlight in the positive x direction. Press and hold the lower-case “k” to

produce rotation of the spotlight about the x-axis. Watch the change in lighting on the magenta cube as the spotlight direction changes. That is not a bad effect for just a little code. There is certainly a lot of power in simple OpenGL commands! Notice the brilliance the face of the cyan cube from which the light seems to emanate.

EMISSION

Definition

Here is a profound revelation. Light sources emit light. They glow. A surface that glows looks different from a surface that merely reflects light. OpenGL provides a way to produce this effect. Notice that the subject is the appearance of surfaces. Therefore the appropriate command is glMaterialfv rather than glLightfv. The new constant is:

```
GL_EMISSION = $1600;
```

The purpose of the use of GL_EMISSION is to intensify the surface color so that it seems to glow. It does not add to the light shed on other surfaces.

Implementation

Save the spot project as Emission.Dpr in OpenGL\Chapter.5\Emit. Save the main unit as Emit1.Pas in the same directory. Since the use of GL_EMISSION applies to the material properties of a face in the spotlight cube, some special purpose code is necessary for this illustration. In the Cube method surround the {back face} part with material commands. The if statement tests for a red component so the magenta cubes will not be affected, but the cyan cube will be affected. Remember this is just special code for a brief illustration.

```
if matColor.red=0.0 then
   glMaterialfv(GL_FRONT,GL_EMISSION,@matColor);
glBegin(GL_POLYGON); {back face}
   glVertex3f(+0.4,-0.4,-0.4);
   glVertex3f(-0.4,-0.4,-0.4);
   glVertex3f(-0.4,+0.4,-0.4);
   glVertex3f(+0.4,+0.4,-0.4);
glEnd;
glMaterialfv(GL_FRONT,GL_EMISSION,@black);
```

The black emission statement confines the effect of the previous emission statement to the one face, otherwise everything would eventually end up with GL_EMISSION. Of course the black field requires declaration within the form’ s class declaration. Happily it will automatically be initialized to all zeros.

```
    black: TLightDesc;
    LightPlace: TLightPlace;
```

```
      SpotDirect: TPosition;
```

Frankly, since the back face of the cyan cube already almost seems to glow, using matColor with GL_EMISSION will have no noticeable effect. Replace it with the emit field.

```
if matColor.red=0.0 then
   glMaterialfv(GL_FRONT,GL_EMISSION,@emit);
```

Declare emit in the form's class declaration:

```
      emit: TLightDesc;
      black: TLightDesc;
      LightPlace: TLightPlace;
      SpotDirect: TPosition;
```

Initialize emit in the FormCreate method:

```
  with emit do
  begin
    red := 0.6;
    green := 1.0;
    blue := 1.0;
    alpha := 1.0;
  end;
end; {FormCreate}
```

Save, Compile, Run, press plus, and press little "o." Hold down little "k" until the back side partially faces the viewer. Notice the intensity of the back face. This illustrates a feature that is seldom used, but you may find handy on special occasions.

To see why you needed glMaterialfv(GL_FRONT,GL_EMISSION,@black), comment it out, re-compile, and re-run. GL_EMISSION affected everything. Ugh! The effect of a glMaterial command remains until explicitly changed.

Remember that emission is just a surface effect in OpenGL; it is not one of the three kinds of light that come from a light source. The next topic deals with such a type of light.

SPECULAR LIGHT

Definition

The third component of light from a light source is *specular* light. Ambient light has been scattered from many surfaces and seems to come from all directions. Diffuse light comes from a specific direction (the source, of course), but scatters from surfaces, giving a soft, "flat" appearance to surfaces. Ambient and Diffuse components of a light source commonly share the same color. Specular light is also directional, but much "tighter" in reflection than diffuse light, often with a different color. Specular light more closely follows the classical rule of reflection wherein the angle of reflection equals the angle of incidence, as in Figure 5-5.

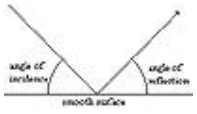

Figure 5-5.Specular Reflection Angles.

As you might expect, the result of reflection of specular light depends greatly on the material properties of the surface. You may have noticed that in the real world surfaces that are smooth and "shiny" often show a small bright spot in bright, direct light. The bright spot is known as the specular highlight. With OpenGL you can give a surface the appearance of being smooth and shiny by producing such a bright spot. While the color of the surface with respect to ambient and diffuse light may vary, the color of the specular highlight tends toward white.

Implementation

To demonstrate the specular component of a light source, load the Spot.Dpr project from OpenGL\Chapter.5\Spot. Create a new directory called OpenGL\Chapter.5\Spec and save the project as Specular.Dpr into the new directory. Save its main unit as Spec1.Pas in the same directory. At the bottom of the FormCreate method add a new glLightfv command.

```
  glLightfv(GL_LIGHT0,GL_SPECULAR,@SpecDef);
end; {FormCreate}
```

Actually, LIGHT0 by default has a pure white specular component, but this illustration needs something a little less intense. Assign a medium gray to it.

```
  with SpecDef do
  begin
    red := 0.5;
    green := 0.5;
    blue := 0.5;
```

```
    alpha := 1.0;
  end;
  glLightfv(GL_LIGHT0,GL_SPECULAR,@SpecDef);
end; {FormCreate}
```

Oh yes! Be sure to declare SpecDef in the form's class declaration.

```
      CubeColor,
      SpotColor,
      DiffDef,
      SpecDef,
      AmbientDef: TLightDesc;
      LightPlace: TLightPlace;
      SpotDirect: TPosition;
```

For convenience, change the initial value of z2 from –2.7 to –3.7, moving the larger cube farther back, so that the spotlight cube will not bury itself in the larger cube in the coming illustration.

```
   z2 := -3.7;
   x2 := -0.7;
   y2 := 0.7;
   zl := -6.7;
   xl := -1.8;
   yl := -2.1;
```

To the specular light component add the surface color response to that light.

```
  glLightfv(GL_LIGHT0,GL_SPECULAR,@SpecDef);
  glMaterialfv(GL_FRONT,GL_SPECULAR,@SpecDef);
end; {FormCreate}
```

Notice that the new glMaterialfv command is in the FormCreate method. Remember that such a command remains in effect until another command changes it. By placing it here, with no contradictory command elsewhere, you give all surfaces the same response to specular light for all time. Well, it lasts for as long as the program runs, anyway.

The specular effect needs one more specification. Remember that specular highlights appear on shiny surfaces. Naturally OpenGL defines a constant for setting shininess in a command.

```
   GL_SHININESS = $1601;
```

The shininess property is an integer that can range in value from 0 to 128. Therefore the property needs an integer form of the glMaterial command. Give it maximum shininess for this illustration.

```
  glLightfv(GL_LIGHT0,GL_SPECULAR,@SpecDef);
  glMaterialfv(GL_FRONT,GL_SPECULAR,@SpecDef);
  glMateriali(GL_FRONT,GL_SHININESS,128);
end; {FormCreate}
```

Save, Compile, Run, and press plus. Press little "m" four times to raise the spotlight toward the magenta cube. Do you see the faint light streak running from the lower left corner to the upper right corner of the near face? It is not much, but that is as good as it gets with a face of a cube. To see a better specular highlight, you need a surface with some curvature. A later chapter introduces curved surfaces.

HOW MUCH LIGHT

So far you have seen three light source components: Ambient, Diffuse, and Specular. Each of these has three color components: red, green, and blue. There is also an alpha component, which still must be put off until later. Never fear! You will see alpha when the time is right. You have also seen material properties which specify the response to light. Material properties also have Ambient, Diffuse, and Specular parts. Each of these also has the three color components.

So what determines how much light of each color reaches the viewer? Would you believe a simple multiplication? Figure 5-6 illustrates.

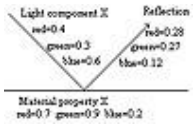

Figure 5-6.Multiplying a Light Component.

DEFINITIONS

Term	Meaning
Ambient light	Light that has reflected from various surfaces and seems to come from all directions with no readily identifiable source. The component of an OpenGL light source used to simulate that effect.
Diffuse light	Light that comes from an identifiable source and scatters when reflected. The component of an OpenGL light source used to simulate that effect.
Flat surface	A surface with enough roughness to scatter most light it reflects, producing a dull, non-glossy appearance. A surface that is not shiny.
Material properties	Characteristics of a surface that affect its appearance.
Shiny surface	A mirror-like surface. A surface that is smooth enough that much of the light it reflects is not scattered, giving a glossy appearance. Such a surface may be characterized by a specular highlight under a suitable light source.
Specular highlight	A bright spot that appears on a smooth, shiny surface under a strong light source.
Specular light	The component of an OpenGL light source used to determine the color and intensity of a specular highlight.
Spotlight	A light source with both position and direction. The space illuminated by the source is shaped like a cone. The cone is defined by an angle from the “aim” direction of the light.

IDENTIFIER REFERENCE

Identifier	Description
glColorMaterial	An OpenGL command that causes material properties to track the current color set by glColor. It takes two parameters. The first tells which face or faces will be affected. The second tells which properties will track the current color. You must enable color tracking for this to work.
glLight	An OpenGL command that defines characteristics of an OpenGL light source. You must enable lighting and enable the specific light.
glLightModel	An OpenGL command that provides a simplified method of setting up a lighting environment instead of defining specific lights.
glMaterial	An OpenGL command for defining material properties.
GL_AMBIENT	An OpenGL constant specifying the ambient portion of light or material properties in various OpenGL commands.
GL_BACK	An OpenGL constant specifying the back face of polygons to be affected in various OpenGL commands.
GL_COLOR_MATERIAL	The OpenGL constant to pass to enable or disable color tracking.
GL_DIFFUSE	An OpenGL constant specifying the diffuse portion of light or material properties in various OpenGL commands.
GL_EMISSION	An OpenGL constant to pass to the glMaterial command, so that the affected surfaces appear to be glowing.
GL_FRONT	An OpenGL constant specifying the front face of polygons to be affected in various OpenGL commands.
GL_FRONT_AND_BACK	An OpenGL constant specifying both the front

	and back faces of polygons to be affected in various OpenGL commands.
GL_LIGHT0 .. GL_LIGHT7	OpenGL constants for specifying specific OpenGL light sources.
GL_LIGHTING	An OpenGL constant to pass to glEnable or glDisable to turn on or turn off lighting calculations.
GL_LIGHT_MODEL_AMBIENT	An OpenGL constant to pass to glLightModel to specify global ambient light.
GL_SPECULAR	An OpenGL constant specifying the specular portion of light or material properties in various OpenGL commands.
GL_SPOT_CUTOFF	An OpenGL constant to pass to glLight to specify the angle that defines the cone of illumination from a spotlight.
Packed	An Object Pascal reserved word used to make a data structure take the minimum amount of memory. The elements of the structure fit together with no unused memory between them because the compiler does not attempt to align the data elements on word or double-word boundaries.
PGLfloat	A data type which is a pointer to a Glfloat.

SUMMARY

Chapter 5 presented new levels of realism with lighting. Here are the high points:

1. Use glEnable(GL_LIGHTING) to inform OpenGL that you will take responsibility for lighting specification.

2. OpenGL allows at least eight separate light sources.

3. All light has a source. Each light source has three components whose color compositions can be separately identified:

a. Ambient light seems to come from all directions.

b. Diffuse light comes from a specific direction and scatters upon reflection, giving a soft, flat appearance to surfaces.

c. Specular light reflects sharply and appears as a specular highlight on smooth surfaces.

4. Specify the color response of a surface to each light component with the glMaterial commands.

5. Emission is also a material property for OpenGL.

6. In addition to the color response to specular light, you must also specify the smoothness or shininess of the surface with the GL_SHININESS constant in the glMateriali command.