An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 6 Depth Test and OpenGL Buffers

6.1 Buffers

We briefly mentioned in previous chapters two types of standard buffers, *depth buffers* and *color buffers*, which are most commonly used. Depth buffers are also referred to as z buffers. In graphics, we can generally consider a buffer as a block of memory arranged in the form of a two-dimensional array with size $m \times n$ with each location corresponding to a pixel position of the screen. Each memory location may save a k-bit datum. For example, in RGBA mode, very often an RGBA value is 32-bit as 8-bit is required to save the value for each of the red, green, blue and alpha components. Therefore, an RGBA color buffer location stores a 32-bit value (i.e. $k = 32$). Similarly, for RGB display, k is 24. For the z buffer, its k is determined by the depth precision the graphics system supports; very often, it is 32 bits which is the size of a floating point number or an integer in most C/C++ programs. We refer to any of the k $n \times m$ one-bit 'planes' in a buffer as a **bitplane**. The k bit datum at a particular buffer location describes a pixel shown on the screen.

In general, OpenGL supports a number of different types of buffers including,

1. Color buffers: front-left, front-right, back-left, back-right, and any number of auxiliary color buffers
2. Depth buffer
3. Accumulation buffer
4. Stencil buffer

The buffers are collectively known as **frame buffer** in OpenGL. When we work with the frame buffer, we usually work with one of the above buffers at a time. Therefore, when we use the term *buffer*, we refer to one particular buffer listed above, which is a constituent of the frame buffer. Exactly what kind of buffers and what value of k is used in a system is determined by the particular OpenGL implementation used. We may use the function **glGetIntegerv**() to query the OpenGL system about the storage information of each buffer location. For example, the following code queries the number of bits used to specify a depth value; the number is returned in the variable *depth*:

```
int depth;
glGetIntegerv ( GL_DEPTH_BITS, &depth );
```

The general syntax of this command is

void glGetIntegerv(GLenum *pname*, GLint **p*);

where *pname* specifies the parameter value (see Table 6-1) to be returned and *p* points to the memory location that the value or values will be returned. Other commands that perform the same task but operate on different data types include

glGetBooleanv(), **glGetDoublev**(), and **glGetFloatv**().

Table 6-1 lists the parameters that can be used with **glGetIntegerv**() to query the bits-per-pixel information of an OpenGL system. If we know the internal format of how data are stored in any of the buffers, we can often write applications that run more efficiently. Moreover, this knowledge helps us encode the image data into some standard image formats such as PNG, GIF and JPEG formats.

Table 6-1 Parameters for **glGetIntegerv**() to Query Per-Pixel Buffer Storage

Parameter	Meaning
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	Number of bits per R, G, B, or A component in the color buffers
GL_INDEX_BITS	Number of bits per index in color buffers
GL_DEPTH_BITS	Number of bits per pixel in depth buffer
GL_STENCIL_BITS	Number of bits per pixel in stencil buffer
GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, GL_ACCUM_ALPHA_BITS	Number of bits per R, G, B, or A component in the accumulation buffer

Color Buffers

The color buffers are the ones to which we store the color data for each pixel. They contain color-index, RGB or RGBA color data. If an OpenGL implementation supports double-buffering, the color buffers consist of front and back buffers. On the other hand, a single-buffered system only has the front buffer. If the system also supports stereoscopic viewing, the color buffers will have left and right buffers for the left and right stereo images. If stereo is not supported, only the left buffer is used. Every OpenGL implementation must provide a front-left color buffer.

Some implementations may also support nondisplayable auxiliary color buffers. OpenGL does not specify any particular uses for these buffers, so we can define and use them in our own way.

We can use the parameters GL_STEREO or GL_DOUBLEBUFFER for **glGetBooleanv**() to find out whether our system supports stereo or double-buffering, and use parameter GL_AUX_BUFFERS to find out how many auxiliary buffers are present.

Depth Buffer

The depth buffer is also called z buffer. It stores a depth value (z coordinate) for each pixel. It is used to remove any hidden-surface in the scene. We usually measure depth in terms of distance to the viewpoint, so pixels with larger depth-buffer values are overwritten by pixels with smaller values.

Accumulation Buffer

The accumulation buffer is simply an extra image buffer that is used to accumulate composite images. It is typically used for accumulating a series of images into a final, composite image to create some special graphics effects such as antialiasing, depth-of-field, and motion blur. We shall use this buffer to create stereoscopic images by accumulating images for the left and right eyes. The accumulation buffer stores RGBA color data just like the color buffers do in RGBA mode. Actually, we do not draw directly into the accumulation buffer; accumulation operations are always performed in rectangular blocks, usually transferring data to or from a color buffer.

Stencil Buffer

The data in a stencil buffer do not represent colors or depths but have application-specific meanings. Unlike the data of a color buffer, the stencil buffer data are not directly visible. By calling the stencil function and the stencil operations, we can change the bits in the stencil planes. We can also use the stencil function along with a stencil test to control whether we want to discard a fragment. Also we can make use of the stencil operation to determine how we update the stencil planes.

A common use for the stencil buffer is to restrict drawing to certain portions of the screen. For example, if we want to draw an image as it would appear through a window of a train, we can store an image of the window's shape in the stencil buffer, and then draw the entire scene. The stencil buffer prevents anything that are not visible through the window from being drawn. Therefore, if our application is a simulation of a running train, we can draw all the items inside the train only once, and as the train moves, we only need to update the outside scene. Another example of application is the use of stencil buffer to define a shadow volume in in the process of creating shadows; we only shadow objects within the shadow volume.

Clearing Buffers

It is always computing-expensive to clear the screen or a constituent of the frame buffer. To clear a 512×512 RGBA color buffer requires clearing a memory of more than one million bytes. It may take more time to clear a buffer than computing the drawing of the scene in a simple graphics application. Very often, a machine has special hardware to handle the buffer-clearing process.

OpenGL provides a number of functions to specify values to clear the buffers:

void glClearColor(float red, float $green$, float $blue$, float $alpha$);
void glClearIndex(float $index$);
void glClearDepth(double $depth$);
void glClearStencil(int s);
void glClearAccum(float red, float $green$, float $blue$, float $alpha$);

The above functions are for clearing the color buffer in RGBA mode, the color buffer in color-index mode, the depth buffer, the stencil buffer, and the accumulation buffer respectively with the values specified by the parameters passed to their arguments. The color values are clamped to be between 0.0 and 1.0. The default clearing values for the functions are 0 except the depth-clearing value which is defaulted to 1.0. The actual clearing occurs when the command **glClear**() is executed with the specified buffer. For example, the commands,

```
glClearColor( 1.0, 1.0, 1.0, 0 );
glClear(GL_COLOR_BUFFER_BIT);
```

will fill the color buffer with white color values.

6.2 Depth Test

When rendering a graphics scene, we do not want to display a fragment which is from an object behind another opaque object. The elimination of parts of solid objects that are

obscured by others is called **hidden-surface removal**. Back face culling is also a form of hidden-surface removal. From another point of view, hidden-surface removal is to discover what part of a surface is visible to the viewer and thus the process is also referred to as **visible-surface determination**. There are two approaches to do this. One approach is to use visible-surface algorithms to find the surfaces that are visible. The other approach is to use hidden-surface-removal algorithms to remove those surfaces that should not be visible to the viewer, and there are two types of such algorithms:

1. **Object-space algorithms** attempt to order the surfaces of the objects in the scene according to their distances from the viewer so that drawing surfaces in a particular order provides the correct image. These algorithms may not work well with pipeline architectures. The **painter's algorithm** is an example of such algorithms. It works as follows:
   (a) First draw the objects that are farthest from the viewpoint.
   (b) Continue to draw objects from far to near and draw the closest objects last.
   (c) Objects drawn later overwrite those drawn earlier at the same location of the projection plane just like the way we paint. Consequently, the nearer objects will obscure the farther objects as we draw from far to near.

   Painter's algorithm is very simple and there are variants of it. The problem of this algorithm is that objects must be drawn in a particular order based upon their distances from the viewpoint. If the viewing position is changed, the drawing order must be changed.
2. **Image-space algorithms** work as part of the projection process. Such an algorithm finds the relationship among object points in each projection step. This works well with pipeline architectures.
   The **z-buffer algorithm** is the most commonly used algorithm of this type. OpenGL also uses the z-buffer algorithm to remove hidden surfaces. The feature is enabled by the command:

   ```
   glEnable ( GL_DEPTH_TEST );
   ```

6.2.1 The z-Buffer Algorithm

Because of its ease of implementation and fitting naturally with pipeline architectures, the **z-buffer algorithm** is the most widely used hidden-surface removal algorithm. The algorithm uses a depth or z buffer to keep track of the distance from the projection plane to each point of the object. For each pixel position, the frame buffer only retains the point of the surface that has the smallest z coordinate. Very often, the depth or z values are normalized to values in the range $[0, 1]$. During the rendering process, the depth-buffer value (normalized z coordinate) $d[i][j]$ contains the depth of the closest object rendered so far at that pixel. As the rendering process proceeds pixel by pixel across a scanline and fills the face of the current polygon, the algorithm tests whether the depth of current point of the face is less than the depth $d[i][j]$ stored in the depth-buffer at the corresponding location. If so, the color of the point of the closer surface replaces the color $c[i][j]$ of the color buffer, and this smaller depth replaces the old value in $d[i][j]$, otherwise no replacement takes place. Polygons and other graphics primitives such as lines and points can be drawn in any order. The following is the algorithm in detail:

1. Clear the color buffer to the background color.

2. Initialize the values at all locations of the depth buffer to 1.
3. For each fragment of each surface, compare depth values to those already stored in the depth buffer:
   (a) Calculate the distance from the projection plane for each xy position on the surface.
   (b) Normalize the distance to $[0, 1]$, which is the depth value of the fragment.
   (c) If the distance is less than the value currently stored in the depth buffer,
      i. set the corresponding position in the color buffer to the color of the fragment,
      ii. set the value in the depth buffer to the distance to that object,

   otherwise

      leave the color and depth buffers unchanged.

Note that specialized hardware depth buffers are much faster than software buffers, and the number of bitplanes associated with the depth buffer determines the depth precision or resolution.

6.2.2 OpenGL z-Buffering

OpenGL uses z-buffering for hidden-surface removal. As we have mentioned above, the z buffer is also referred to as depth buffer. Since using a depth buffer means we need a lot of extra memory, we have to inform OpenGL our request of the extra memory. This can be done in the initialization steps using a command like the following:

glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

Before using the depth buffer, we must enable it and clear it. Very often, we clear the depth buffer whenever we clear the color buffer. This is done by the command,

glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

Depth test is enabled by

glEnable (GL_DEPTH_TEST);

and disabled by

glDisable (GL_DEPTH_TEST);

We use the function **glDepthFunc**() to set the comparison function for the depth test. This function specifies the function used to compare each incoming pixel depth value with the depth value present in the depth buffer. The comparison is performed only if depth testing is enabled. Its syntax is as follows:

void glDepthFunc (GLenum $func$);

where $func$ specifies the comparison function, which may be

GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL

For example, when $func$ is GL_GEQUAL, the depth test passes if the incoming pixel depth value is greater than or equal to the stored depth value. The default of $func$ is GL_LESS. Initially, depth testing is turned off (disabled). If depth testing is disabled or if no depth buffer exists, the program works as if the depth test always passes. Table 6-2 lists all the valid values of $func$ and their meanings.

Table 6-2 Valid Parameters for **glDepthFunc**()

func	Depth Test
GL_NEVER	Never passes.
GL_LESS	Passes if the incoming depth value is less than the stored depth value.
GL_EQUAL	Passes if the incoming depth value is equal to the stored depth value.
GL_LEQUAL	Passes if the incoming depth value is less than or equal to the stored depth value.
GL_GREATER	Passes if the incoming depth value is greater than the stored depth value.
GL_NOTEQUAL	Passes if the incoming depth value is not equal to the stored depth value.
GL_GEQUAL	Passes if the incoming depth value is greater than or equal to the stored depth value.
GL_ALWAYS	Always passes.

We can turn on depth buffering by

glEnable(GL_DEPTH_TEST);

and turn it off by

glDisable(GL_DEPTH_TEST);

Each time prior to drawing the scene, we clear the depth buffer when we clear the color buffer in the display callback:

glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | ...);

We usually use a single **glClear** command to clear all buffers.

If necessary, we may change the depth range so that z values may lie in a range other than $[0, 1]$. This is done using

void **glDepthRange**(GLclampd $near$, GLclampd far);

where $near$ and far specify the minimum and maximum values that can be stored in the depth buffer. By default, $near$ and far are 0.0 and 1.0 respectively. The data type **Glclampd** specifies a floating-point (double) value clamped to $[0, 1]$.

We can also enable or disable writing to the depth buffer using the following command:

void **glDepthMask**(GLboolean $flag$);

where $flag$ specifies whether the depth buffer is enabled for writing:

- GL_TRUE enables writing
- GL_FALSE disables writing

Writing is enabled by default. When writing is disabled, the depth buffer is read-only; its value will not be changed after a depth test.

We can similarly use the function **glColorMask**() to enable or disable the writing of a color component to the color buffer:

void glColorMask(GLboolean $rflag$, GLboolean $gflag$, GLboolean $bflag$,

GLboolean $aflag$);

The parameters $rflag, gflag, bflag$, and $aflag$ are flags specifing whether red, green, blue, and alpha color components can or cannot be written into the color buffer. For example, if $bflag$ is GL_FALSE, no change is made to the blue component of any pixel in any of the color buffers, regardless of the drawing operation we have used. Note that we cannot control changes to individual bits of components. Rather, changes are either enabled or disabled for entire color components. The initial values of all the flags are GL_TRUE, indicating that we can write the color components to the buffer. Making use of this function, we can easily create a red-blue stereoscopic pair of images; when creating the red image, we enable only the writing of the red component and when creating the blue image, we enable only the writing of the blue component. We shall have more detailed discussions and examples on this in later chapters.

6.3 Writing to Buffers

We have learned the principles and usage of the buffers in OpenGL. In an application, we may want to read and write into the buffers directly and we would like to do these efficiently. OpenGL provides a few functions to transfer data to and from these buffers in large chunks.

In general, we call a frame buffer with one bit per pixel a **bitmap**, and a frame buffer with multiple bits per pixel a **pixmap**. We also use these terms to describe other rectangular arrays; we may refer to any pattern of binary values as a bitmap, and multicolor pattern as a pixmap. Specifications for a pixel array may consist of a pointer to the color matrix, the size of the matrix, and the position and size of the screen area involved in the data reading or writing. Figure 6-1 shows an example of data transfer of an $m \times n$ block of pixels from a **source buffer** to another buffer, the **destination buffer**. Sometimes the source buffer and the destination buffer can be the same constituent of the frame buffer.

In Figure 6-1, the operation writes an $m \times n$ source block whose lower-left corner is at (x_0, y_0) to the destination buffer starting at a location (u_0, v_0). The transfer of the entire block is usually performed by a single function call in the following form:

writeBlock ($source$, x_0, y_0, $destination$, u_0, v_0, m, n);

where $source$ and $destination$ points to the memory locations of the source and destination blocks respectively.

In OpenGL, the commands that perform block data transfer include **glReadPixels**(), **glDrawPixels**(), **glCopyPixels**(), and **glBitmap**(). Some commands require the functions **glRasterPos***() to help specify the frame buffer location that the data are read from or written to. We discuss these functions below.

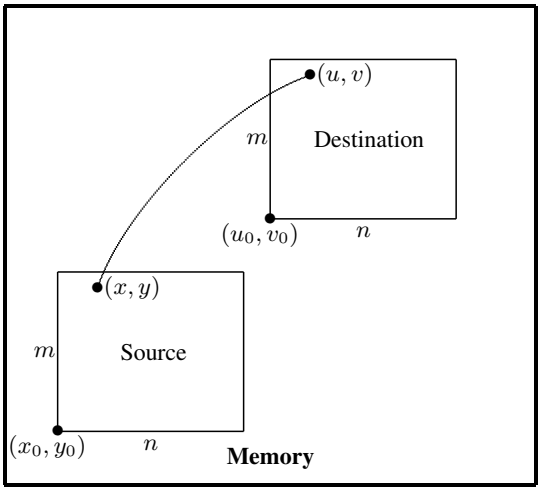

Figure 6-1 Transfer of an $m \times n$ Block of Data

6.3.1 OpenGL Pixel-Array Function

OpenGL provides a bitmap function and a pixmap function to define a shape or pattern specified by a rectangular array. Also, OpenGL provides several other functions for saving, copying, and manipulating arrays of pixel values.

Bitmap Function

The OpenGL functions **glRasterPos***() and **glBitmap**() specify the position and draw a single bitmap on the screen. The commands **glPasterPos***() specify the current raster position which is the origin where the next bitmap is to be drawn. The suffix codes and parameters are the same as those for the **glVertex** functions. Therefore, a current raster position is specified in world coordinates, and it is transformed to screen coordinates by the modelview and projective transformations. The color for the bitmap is the color that is in effect when the **glRasterPos** command is invoked. Any subsequent color changes do not affect the bitmap. Once we have set the desired raster position, we can use the **glBitmap**() function to draw the scene specified by the data passed to the function:

void glBitmap (GLsizei $width$, GLsizei $height$, GLfloat $x0$, GLfloat $y0$, GLfloat xi, GLfloat yi, const GLubyte *$bitmap$);

where $bitmap$ is a pointer to the bitmap image to be drawn. The origin of the bitmap image is at the current raster position. If the current raster position is invalid, nothing is drawn or changed. We use the arguments $width$ and $height$ to specify the width and height, in pixels, of the bitmap image. i We use the parameters $x0$ and $y0$ to define the origin of the bitmap image relative to the current raster position; a positive $x0$ shifts the origin right and a positive $y0$ shifts it upwards relative to the raster position. Negative values shift it down and left. Parameters xi and yi specify the x and y increments that are added to the raster position after the bitmap has been rasterized. This means that xi and yi are measured in pixels rather than world coordinates. After the bitmap image has been drawn, the current raster position is advanced by xi and yi in the x and y directions respectively. Figure 6-2

below shows an example illustrating these parameters, with $width = 10$, $height = 12$, $(x0, y0) = (0, 0)$, and $(xi, yi) = (11, 0)$; each square represents a pixel.

Figure 6-3 shows an example of a bitmap which is the image pattern of an upward arrow. In this example, the width is 9 pixels and the height is 10 pixels. The image is saved in a rectangular bit array where each row of it is stored in multiples of 8 bits (16 here) and the binary data are arranged as a set of 8-bit unsigned characters. When this image pattern is applied to the pixels in the frame buffer, all bit values beyond the 9-th column are ignored. Suppose we are going to draw this bitmap and that the current raster color is green. Wherever there is a 1 in the bitmap, the corresponding pixel is replaced by a green pixel. If there is a 0 in the bitmap, the contents of the pixel are unaffected. Therefore, an upward arrow will be drawn using this bitmap. Actually, the most common use of bitmaps is for drawing characters on the screen.

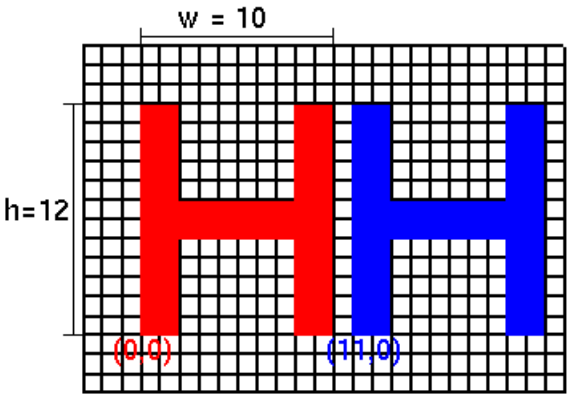

Figure 6-2 Bitmap Image

9	0	0	0	0	**1**	0	0	0	0	0	0					0x08 0x00
	0	0	0	**1**	**1**	**1**	0	0	0	0	0					0x1c 0x00
	0	0	**1**	**1**	**1**	**1**	**1**	0	0	0	0					0x3e 0x00
	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0					0x7f 0x00
	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0					0xff 0x80
	0	0	0	**1**	**1**	**1**	0	0	0	0	0					0x1c 0x00
	0	0	0	**1**	**1**	**1**	0	0	0	0	0					0x1c 0x00
	0	0	0	**1**	**1**	**1**	0	0	0	0	0					0x1c 0x00
	0	0	0	**1**	**1**	**1**	0	0	0	0	0					0x1c 0x00
0	0	0	0	**1**	**1**	**1**	0	0	0	0	0					0x1c 0x00
	0							8		11					15	

Figure 6-3 10×9 Bit Pattern Saved in 10×16 Block

We can display the arrow image of Figure 6-3 by applying the bit pattern to a frame buffer location with the following code:

```
void display(void)
{
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho (-1.0, 1.0, -1.0, 1.0, -10.0, 10.0);
```

```
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glClear (GL_COLOR_BUFFER_BIT );
    glColor4f(0, 0, 0, 0);
    GLubyte arrow[20] = {0x1c, 0x00, 0x1c, 0x00, 0x1c, 0x00,
                         0x1c, 0x00, 0x1c, 0x00, 0xff, 0x80,
                         0x7f, 0x00, 0x3e, 0x00, 0x1c, 0x00,
                         0x08, 0x00};
    //set pixel storage mode, byte-alignment
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
    glRasterPos2f( 0.1, 0.2 );
    glBitmap ( 9, 10, 0, 0, 20, 30, arrow );
    glBitmap ( 9, 10, 0, 0, 20, 30, arrow );
    glBitmap ( 9, 10, 0, 0, 20, 30, arrow );
}
```

In the above code, we specify the array values for $arrow$ row by row, starting at the bottom of the rectangular-grid pattern as shown in Figure 6-3. We set the storage mode of the bitmap using the the function **glPixelStorei**(). The parameter GL_UNPACK_ALIGNMENT specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries). We have used the value 1 in the code. Thus, the data values are to be aligned on byte-boundaries. We use the function **glRasterPos2f**() to set the current raster position to $(0.1, 0.2)$, which are in world coordinates. The parameters to the function **glBitmap**() indicate that the bit pattern is given in the array $arrow$, and that this array has 9 columns (width) and 10 rows (height). The location of this pattern in the block is $(0, 0)$, which is the lower left corner of the grid. We illustrate a raster position increment with values $(20, 30)$ which are the offset between two successive arrows drawn; after each call to the function **glBitmap**(), the raster position is incremented by this amount in **pixels**. Therefore, the code displays three arrows, each with displacement $(20, 30)$ relative to the preceding one. Figure 6-4 shows the output of the code.

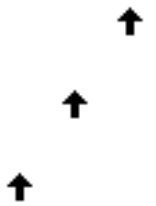

Figure 6-4 Output Of Bitmap Example Code

Pixmap Function

A **pixmap** is a general **image** which is similar to a bitmap, but instead of containing only a single bit for each pixel in a rectangular region of the screen, a pixmap can contain much more information such as a complete (R, G, B, A) color of each pixel. The image sources of a pixmap include the following,

1. a photo image that is taken by a digital camera,

2. a photograph that is digitized with a scanner,
3. an image that was first generated on the screen by a graphics program using the graphics hardware. and then read back, pixel by pixel, and
4. a software program that generates the image in memory pixel by pixel.

OpenGL provides three functions for manipulating image data:

1. **glReadPixels** () – Reads a rectangular array of pixels from the frame buffer and stores the data in processor memory.
2. **glDrawPixels**() – Writes a rectangular array of pixels from data kept in processor memory into the frame buffer at the current raster position specified by glRaster-Pos*().
3. **glCopyPixels**() – Copies a rectangular array of pixels from one part of the frame buffer to another. This function behaves similarly to a call to **glReadPixels**() followed by a call to **glDrawPixels**(), except that the data are never written into processor memory.

Figure 6-5 shows the OpenGL pixel data flow and the tasks of these functions.

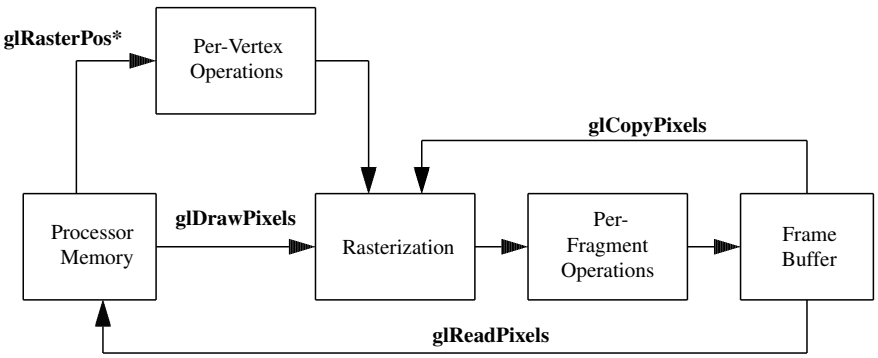

Figure 6-5 Pixel Data Flow

6.3.2 Pixel Data Block Transfer

The three pixmap functions discussed above provide an efficient way to transfer data between the processor memory and the frame buffer in large blocks. The following shows more details of these functions.

void **glReadPixels**(GLint $x0$, GLint $y0$, GLsizei $width$, GLsizei $height$,
GLenum $format$, GLenum $type$, GLvoid *$pixels$);
Reads pixel data from the framebuffer rectangle whose lower-left corner is at $(x0, y0)$ in screen coordinates (pixels) and whose dimensions are $width$ and $height$ and stores it in the array pointed to by $pixels$. Parameter $format$ indicates the kind of pixel data elements that are read (an index value or an R, G, B, or A component value, as listed in Table 6-1 below), and $type$ indicates the data type of each element shown below.

void **glDrawPixels**(GLsizei $width$, GLsizei $height$, GLenum $format$,
GLenum $type$, const GLvoid *$pixels$);

Draws a rectangle of pixel data with dimensions $width$ and $height$. The pixel rectangle is drawn with its lower-left corner at the current raster position. Parameters $format$ and $type$ have the same meaning as with **glReadPixels**(). The array pointed to by $pixels$ contains the pixel data to be drawn.

void **glCopyPixels**(GLint $x0$, GLint $y0$, GLsizei $width$, GLsizei $height$, GLenum *buffer*);
Copies pixel data from the frame buffer rectangular region whose lower-left corner is at $(x0, y0)$ and whose dimensions are $width$ and $height$. The data are copied to a new position whose lower-left corner is given by the current raster position. Parameter *buffer* is either GL_COLOR, GL_STENCIL, or GL_DEPTH, specifying the frame buffer that is used. **glCopyPixels**() behaves similarly to a **glReadPixels**() followed by a **glDrawPixels**(), with the following translation for the *buffer* to $format$ parameter:

- If *buffer* is GL_DEPTH or GL_STENCIL, then GL_DEPTH_COMPONENT or GL_STENCIL_INDEX is used respectively.
- If GL_COLOR is specified, then GL_RGBA or GL_COLOR_INDEX is used, depending on whether the system is in RGBA or color-index mode.

Table 6-1 Pixel Formats for **glReadPixels**() or **glDrawPixels**()

Format Constant	Pixel Format
GL_COLOR_INDEX	A single color index
GL_RGB	Red, green, blue color components
GL_RGBA	Red, green, blue, alpha color components
GL_RED	A single red color component
GL_GREEN	A single green color component
GL_BLUE	A single blue color component
GL_ALPHA	A single alpha color component
GL_LUMINANCE	A single luminance component
GL_LUMINANCE_ALPHA	A luminance component followed by an alpha
GL_STENCIL_INDEX	A single stencil index
GL_DEPTH_COMPONENT	A single depth component

Example 6-1

This example presents a code section shown below that displays a 16×16 checker board of alternate black and white squares on the screen. Each black or white square is 16×16 pixels. It uses **glDrawPixels**() to draw a pixel rectangle in the lower-left corner of a window. The function **getBoard()** creates a 256×256 RGBA array representing the black-and-white checker board image. The function call **glRasterPos2i**(0,0) positions the lower-left corner of the image at the origin of the world coordinate system. The output of this code is shown in Figure 6-6.

```
const int width = 256, height = 256;
unsigned char board[width][height][4];

//create checker board image
void getBoard(void)
{
  int  c;

  for ( int i = 0; i < width; i++ ) {
```

```
      for ( int j = 0; j < height; j++ ) {//start from lower-left corner
        c = ((((i&16)==0)^((j&16))==0))*255; //0 or 255-->black or white
        board[i][j][0] =  c; //red component
        board[i][j][1] =  c; //green component
        board[i][j][2] =  c; //blue  component
      }
    }
}

//initialization
void init(void)
{
   glClearColor (0.8, 0.8, 1.0, 1.0);
   glShadeModel(GL_FLAT);
   getBoard();    //create checker image array
   //Sets pixel-storage mode: byte-aligned
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glRasterPos2i(0, 0);
  glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, board);
  glFlush();
}
```

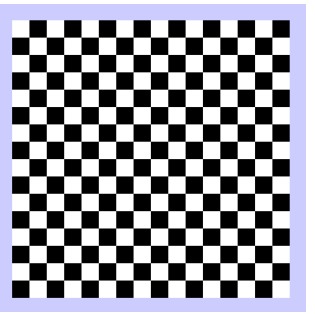

Figure 6-6 Output of Code of Example 6-1

6.4 Displaying and Saving Images

OpenGL does not provide functions to handle images in standard formats such as PNG, JPEG, TIFF, GIF, and PPM. This is because there exists too many proprietary image and video file formats, each with clear strengths and weaknesses. The file formats are generally not a user-defined option and many of the features are specified by the vendors. Because of the lack of OpenGL support, we have to write our own routines to read image data from files to the frame buffer or to take images formed in the frame buffer and save them in a standard format. We are certainly interested in exploring all the file formats. Often, we may use some open-source libraries to process the images. As an illustration of the basic ideas of OpenGL pixel functions, here we develop our own simple routines that read and write data in the PPM format, which is the simplest of all image formats. We shall also discuss a couple of simple standard formats and some related tools that can convert from one format to the other.

6.4.1 Portable Pixel Map (PPM)

The Portable Pixel Map (PPM) file format is a lowest and simplest common denominator color image format. A PPM file contains very little information about the image besides basic colors and the data are not compressed. Therefore, it is easy to write programs to process the file, which is the purpose of this format. A PPM file consists of a sequence of one or more PPM images. There are no data, delimiters, or padding before, after, or between images. The PPM format closely relates to two other bitmap formats, the PBM format, which stands for Portable Bitmap (a monochrome bitmap), and PGM format, which stands for Portable Gray Map (a gray scale bitmap). All these formats are not compressed and consequently the files stored in these formats are usually quite large. In addition, the PNM format means any of the three bitmap formats. You may use the unix manual command **man** to learn the details of the PPM format:

$**man ppm**

The three bitmap formats can be stored in two possible representations:

1. an ASCII text representation (which is extremely verbose), and
2. a binary representation (which is comparatively smaller).

Each PPM image consists of the following (taken from unix ppm manual):

1. A "magic number" for identifying the file type. A ppm image's magic number is the two characters "P6".
2. Whitespace (blanks, TABs, CRs, LFs).
3. A width, formatted as ASCII characters in decimal.
4. Whitespace.
5. A height, again in ASCII decimal.
6. Whitespace.
7. The maximum color value (*Maxval*), again in ASCII decimal. Must be less than 65536 and more than zero.
8. Newline or other single whitespace character.
9. A raster of *Height* rows, in order from top to bottom. Each row consists of *Width* pixels, in order from left to right. Each pixel is a triplet of red, green, and blue samples, in that order. Each sample is represented in pure binary by either 1 or 2 bytes. If the *Maxval* is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first. A row of an image is horizontal. A column is vertical. The pixels in the image are square and contiguous.
10. In the raster, the sample values are "nonlinear". They are proportional to the intensity of the ITU-R Recommendation BT.709 red, green, and blue.

In summary, a PPM file has a header and a body, which may be created using a text editor. The header is very small with the following properties:

1. The first line contains the magic identifier "P3" or "P6".
2. The second line contains the *width* and *height* of the image in ascii code.
3. The last part of the header is the maximum color intensity integer value.
4. Comments are preceded by the symbol #.

Here are some header examples:

Header example 1

```
P6 1024 788 255
```

Header example 2

```
P6
1024 788
# A comment
255
```

Header example 3

```
P3
1024  # the image width
788   # the image height
      # A comment
1023
```

The following is an example of a PPM file in P3 format.

```
P3
# feep.ppm
4 4
15
0  0  0    0  0  0    0  0  0   15  0 15
0  0  0    0 15  7    0  0  0    0  0  0
0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```

You can simply use a text editor to create it; for example, copy-and-paste the content into a file named "feep.ppm", which then becomes a PPM file and can be viewed by a browser or the unix utility **xview**. When you execute the unix command,

$ **xview feep.ppm**

you should see a tiny image appear on the upper left corner of your screen along with the following messages displayed in the console,

```
feep.ppm is a 4x4 PPM image with 16 levels
  Building XImage...done
```

6.4.2 The **Convert** Utility

Once we obtain an image in PPM format, we can easily convert it to other popular formats such as PNG, JPG, or GIF using the **convert** utility, which is a member of the ImageMagick suite of tools. Conversely, if you obtain an image from other sources in another format, you may also use **convert** to convert it to the PPM format. Besides making conversion between image formats, the utility can also resize an image, blur, crop, despeckle, dither, draw on, flip, join, re-sample, and do much more. It can even create an image from text. We use the unix manual command to see the details of its usage:

$ **man convert**

We can also run 'convert -help' to get a summary of its command options. The following are some simple examples of its usage.

```
$convert feep.ppm  feep.png
$convert house.jpg house.ppm
$convert house.jpg -resize 60% house.png
$convert -size 128x32 xc:transparent -font \
    Bookman-DemiItalic -pointsize 28 -channel RGBA \
    -gaussian 0x4 -fill lightgreen -stroke green \
    -draw "text 0,20 'Freedom'" freedom.png
```

The last command creates a PNG (Portable Network Graphics) file named "freedom.png" from the text "Freedom". Figure 6-7 shows the image thus created.

Figure 6-7 Image Created by **convert**

If you want to convert a PDF file to PPM, you may use the utility **pdftoppm**. You may run **"pdftoppm –help"** to find out the details of its usage.

6.4.3 Read and Write PPM Files

To process any PPM and related graphics file, you may use the the **netpbm** library (*http://netpbm.sourceforge.net*), which can be downloaded from the Internet. However, for the purpose of this book, we just need something very simple to read or write a PPM file. In this section, we present a simple C program that shows how to read or write a PPM file.

The C/C++ program shown in **Listing 6-1** briefly demonstrates the reading and writing of PPM files; the file names and some parameters are hard-coded; the class **CImage** with public members *red, green*, and *blue* is used to save the color data of one pixel. In a C++ program, a public **class** is the same as a C **struct**.

Program Listing 6-1 Read and Write PPM Files

```
/* ppmdemo.cpp
 * Demostrate read and write of PPM files.
 * Compile: g++ -o ppmdemo ppmdemo.cpp
 * Execute: ./ppmdemo
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//A public class is the same as a 'struct'
class CImage {
public:
  unsigned char red;
  unsigned char green;
  unsigned char blue;
};

// Create PPM header from image width and height. "P6" format used.
// PPM header returned in integer array ppmh[].
void make_ppm_header ( int ppmh[], int width, int height )
{
```

```
  //standard header data, 'P' = 0x50, '6' = 0x36, '\n' = 0x0A
  int ca[] = {0x50, 0x36, 0x0A,                    //"P6"
      //image width=260, height = 288
      0x33, 0x36, 0x30, 0x20, 0x32, 0x38,
              //color levels / pixel = 256
              0x38, 0x0A, 0x32, 0x35, 0x35, 0x0A };

  //only have to change width and height
  char temp[10], k;

  sprintf(temp, "%3d", width );             //width in ascii code
  k = 0;
  for ( int i = 3; i <= 5; ++i )            //replace width
    ca[i] = temp[k++];

  sprintf(temp, "%3d", height );            //height in ascii code
  k = 0;
  for ( int i = 7; i <=9; ++i )             //replace height
    ca[i] = temp[k++];

  for ( int i = 0; i < 15; ++i )            //form header
    ppmh[i] = ca[i];
}

void save_ppmdata (FILE *fp, CImage *ip, int width, int height)
{
  int size = width * height;
  for ( int i = 0; i < size; ++i ){
    putc ( ip[i].red, fp );
    putc ( ip[i].green, fp );
    putc ( ip[i].blue, fp );
  }
}

void ppm_read_comments ( FILE *fp )
{
  int c;
  while ( (c = getc ( fp ) )  == '#' ) {
    while ( getc( fp ) != '\n' )
        ;
  }
  ungetc ( c, fp );
}

class ppm_error
{
  public:
    ppm_error() {
      printf("\nIncorrect PPM format!\n");
      exit ( 1 );
    }
};

int main()
{
  int ppmh[20];                             //PPM header
  int width = 32, height = 32;              //image width and height
  make_ppm_header ( ppmh, width, height );
  //PPM file for testing read
  FILE *input = fopen("testread.ppm", "rb");
```

```
  //PPM file for testing write
  FILE *output = fopen ("testwrite.ppm", "wb");

  //write demo
  for ( int i = 0; i < 15; ++i )          //save PPM header
    putc ( ppmh[i], output );

  CImage image[width][height];
  for ( int i = 0; i < height; ++i ) {    //create a red rectangle
    for ( int j = 0; j < width; ++j ) {
        image[i][j].red = 255;            //red component
        image[i][j].green = 0;            //green component
        image[i][j].blue = 0;             //blue component
    }
  }
  save_ppmdata ( output, (CImage*) image, width, height );
  printf("\nPPM file testwrite.ppm created!\n");
  fclose ( output );

  //read demo
  ppm_read_comments ( input );            //read comments
  char temp[100];
  fscanf ( input, "%2s", temp );
  temp[3] = 0;
  if ( strncmp ( temp, "P6", 2 ) )
    throw ppm_error();
  ppm_read_comments ( input );
  fscanf ( input, "%d", &width );
  ppm_read_comments ( input );
  fscanf ( input, "%d", &height );
  ppm_read_comments ( input );
  int colorlevels;
  fscanf ( input, "%d", &colorlevels );
  printf("\n%s PPM file: ", temp );
  printf(" \n\twidth=%d\theight=%d\tcolorlevles=%d\n",
              width,height,colorlevels+1 );
  ppm_read_comments ( input );
  while (( c = getc ( input )) == '\n'); //get rid of extra returns
  ungetc ( c ,input );

  //save the data in another file
  CImage ibuf[width][height];
  fread ( ibuf,  3, width * height, input );
  output = fopen("test.ppm", "wb"); //to save PPM data in "test.ppm"
  make_ppm_header ( ppmh, width, height );
  for ( int i = 0; i < 15; ++i )    //save PPM header
    putc ( ppmh[i], output );
  save_ppmdata (output, (CImage*) ibuf, width, height);//save data
  printf("\nPPM file test.ppm created!\n");

  fclose ( input );  fclose ( output );
  return 0;
}
```

When you execute the program **ppmdemo**, you should see messages similar to the following displayed.

```
PPM file testwrite.ppm created!
```

```
P6 PPM file:
        width=200       height=300      colorlevles=256

PPM file test.ppm created!
```

The program first creates a PPM file named "testwrite.ppm" whose data form a red square. If you view the file with the command **xview testwrite.ppm**, you should see a small red square image. The program then reads in the data from the PPM file "testread.ppm" and prints out its width, height and color levels. Finally, it writes the information to another file named "test.ppm". Again, you can view the image using the command **xview test.ppm**.

6.4.4 Saving Graphics with glReadPixels

With the PPM functions developed in the above section, we can save an OpenGL graphics scene to a PPM file. We first use the function **glReadPixels**() to read the specified scene on the screen into a buffer and then save the data in the buffer in a PPM file. Then we can use the **convert** utility to convert the PPM file to a file in any standard image format such as PNG, JPEG, TIFF, or GIF. The following code section shows an example that saves the graphics scene in the file "teapot.ppm"; the code for error-checking and lighting is omitted here. The scene saved is shown in Figure 6-8.

```
void save_ppmRGB ( FILE *fp, unsigned char *p, int width, int height )
{
  //save data up-side-down
  int k;
  for ( int i = 0; i < height; i++ ) {
    k = (height - 1 - i) * width * 3;
    for ( int j = 0; j < width; j++ ) {
      putc( p[k++], fp );
      putc( p[k++], fp );
      putc( p[k++], fp );
    }
  }
}

void ppm_save_image ( char *n1, int x0, int y0, int width, int height )
{
  unsigned char *imgBuffer;
  imgBuffer = ( unsigned char *) malloc( 3 * width * height );
  //read the screen data to buffer pointed by
  glPixelStorei(GL_PACK_ALIGNMENT, 1);
  glReadPixels(x0,y0, width,height, GL_RGB, GL_UNSIGNED_BYTE, imgBuffer);

  int ppmh[20], c;                  //PPM header
  make_ppm_header ( ppmh, width, height );
  FILE *fpo = fopen ( n1, "wb" );
  for ( int i = 0; i < 15; ++i )    //save PPM header
    putc ( ppmh[i], fpo );
  save_ppmRGB ( fpo, imgBuffer, width, height );
  fclose ( fpo );
  delete imgBuffer;
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();              // clear the matrix
  glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
```

```
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt ( 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  glColor3f (1.0, 0.0, 0.0);     //red color
  glutSolidTeapot( 0.6 );        //draw a teapot
  glColor3f (0.0, 1.0, 0.0);     //green color
  glTranslatef(-0.5, -0.5, -1.0);
  glutSolidSphere( 0.6, 40, 40 );//draw a sphere
  glFlush ();
  int x0 = 140, y0 = 160, width = 200, height = 150;
  ppm_save_image ( "teapot.ppm", x0, y0, width, height );
}
```

Note that in the function **save_ppmRGB**(), we save the image data in the buffer *imgBuffer* in an "up-side-down" manner. This is because **glReadPixels** considers the lower-left corner of the screen window as $(0, 0)$ but the PPM image format considers the upper-left corner as $(0, 0)$.

We can view the image from the program with the UNIX command "**xview teapot.ppm**" which displays the image as shown in Figure 6-8 along with the following message:

```
$ xview teapot.ppm
teapot.ppm is a 200x150 Raw PPM image with 256 levels
  Building XImage...done
```

Figure 6-8 PPM Image Created by Above Code

The command "convert teapot.ppm teapot.png" converts the PPM data to PNG format.

6.4.5 Displaying PPM Images with glDrawPixels

If we want to display a PPM image in an OpenGL application, we can use the function **glReadPixels**() to do so. The following code section demontrates how this is done; it calls the function **read_ppm_image**() to read in the data of the ppm file "../data/soolee.ppm" into a buffer. The function then uses **glDrawPixels**() to display the image at the specified location $(x0, y0)$ which is $(0, 0)$ in the example. The code also uses the function **glutSolidSphere**() to create a sphere that simulates an exaggerated table tennis ball and superimpose it on the PPM image. The error-checking statements are omitted in the program. (Readers can download complete programs which may contain error-checking statements from this book's web site.) The output of the code is shown in Figure 6-9.

```
void read_ppm_image( char *n1, float x0, float y0 )
{
  FILE *fpi = fopen( n1, "rb");
  int width, height, c, colorlevels;;
  unsigned char *imgBuffer;
  char temp[100];
```

```
  ppm_read_comments ( fpi );
  fscanf ( fpi, "%2s", temp );
  temp[3] = 0;
  if ( strncmp ( temp, "P6", 2 ) )
    throw ppm_error();
  ppm_read_comments ( fpi );
  fscanf ( fpi, "%d", &width );
  ppm_read_comments ( fpi );
  fscanf ( fpi, "%d", &height );
  ppm_read_comments ( fpi );
  fscanf ( fpi, "%d", &colorlevels );
  ppm_read_comments ( fpi );
  while ( ( c = getc ( fpi )) == '\n' );//get rid of extra line returns
  ungetc ( c , fpi);

  imgBuffer = ( unsigned char *) malloc( 3 * width * height );
  //read image up-side down
  int k;
  for ( int i = 0; i < height; i++ ) {
    k = (height - 1 - i) * width * 3;
    for ( int j = 0; j < width; j++ ) {
      imgBuffer[k++] = (unsigned char) getc( fpi );
      imgBuffer[k++] = (unsigned char) getc( fpi );
      imgBuffer[k++] = (unsigned char) getc( fpi );
    }
  }

  //read data pointed by imgBuffer to frame buffer for display
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glRasterPos2f ( x0, y0 );
  glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, imgBuffer);

  fclose ( fpi );
  delete imgBuffer;
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();             // clear the matrix
   glOrtho (-1.0, 1.0, -1.0, 1.0, -10.0, 10.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity();
   float x0 = -1.0, y0 = -1.0;
   read_ppm_image ("../data/soolee.ppm", x0, y0);
   glColor3f (1.0, 0.8, 0.0);
   glutSolidSphere( 0.2, 40, 40 );//draw a ball
   glFlush ();
}
```

Figure 6-9 Display PPM Image Using glDrawPixels

6.4.6 Portable Network Graphics (PNG)

We have seen that PPM image format is very simple and easy to process. It is an excellent format for illustration and testing certain graphics programming concepts. However, the data are stored without any compression and the header contains only very little information about the image data. If we want to read and write image data from and to a file more efficiently, it is better to use one of the more sophisticated open standards. We feel that the best open format that we should use is the **portable network graphics** (PNG), which is an open, extensible image format with lossless compression and the following features:

1. It is designed in 1995 specifically in response to the patent problems with the LZW algorithm used in GIF.
2. It is 'completely' patent-free.
3. It supports alpha channels and allows up to 254 levels of partial transparency.
4. It supports gamma correction.
5. One can find more detailed information of PNG at the site *http://www.libpng.org/*.

Instead of writing our own routines to process PNG files, we shall use a simple open-source library to accomplish the task. The library that we shall use is the **imageio library utility** developed by Adrien Treuille of CMU. It is particularly simple to compile and use. One can download the library source code from the site

http://www.cs.cmu.edu/~treuille/resc/imageio/

or from this book's web site. This simple library allows users read and write png and tiff files. The package consists of the program "imageio.cpp" and the header file "imageio.h". Assuming that the OpenGL directories are accessible, we can compile the program with the command,

g++ -c imageio.cpp

which generates the object file "imageio.o". To use the functions of the programs, we include the header file "imageio.h" in our application program:

#include "imageio.h"

Then we link the object file with our application program using a command like the following:

g++ -o *program program.o* imageio.o -ltiff -lpng

Display PNG Image

We can use **loadImageRGBA**() of **imageio** to read a PNG image into memory:

unsigned char *loadImageRGBA(unsigned char **fileName*, int **width*, int **height*);

This function returns a pointer pointing to the uncompressed image data loaded from the file specified by *filename*. Parameters *width* and *height* will contain the image width and height respectively. The byte order of images created with this library is compatible with that used by OpenGL. We can use **glDrawPixels** to write the data to screen. The following piece of code is an example of using this function to render an image on the screen:

```
int width, height;
char *filename = "soolee.png";
unsigned char *ibuff = loadImageRGBA( filename, &width, &height);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glRasterPos2f ( 1.0, 2.0 ); //position to display image
glDrawPixels ( width, height, GL_RGBA, GL_UNSIGNED_BYTE, ibuff );
```

Save Graphics Scene in PNG

We can save memory data into a PNG file using the function **saveImageRGBA**() of **imageio**:

bool saveImageRGBA(char **fileName*, unsigned char **buffer*, int *width*, int *height*);

This function returns true if the image data stored in *buffer* is successfully saved in the file specified by *filename*, otherwise it returns false. The parameters *width* and *height* specify the width and height of the image. The following code section is an example that saves a graphics scene into a PNG file:

```
void png_save_image( char *n1, int x0, int y0, int width, int height)
{
  unsigned char *imgBuffer;
  imgBuffer = ( unsigned char *) malloc( 4 * width * height );
  if ( imgBuffer == NULL ) {
    printf("\nError in allocating memory!\n");
    return;
  }
  glPixelStorei(GL_PACK_ALIGNMENT, 1);
  glReadPixels(x0,y0, width,height, GL_RGBA,GL_UNSIGNED_BYTE,imgBuffer);
  saveImageRGBA(n1, imgBuffer, width, height);
  delete imgBuffer;
}
```

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273