Chapter 7

Imaging and Blending Extensions

This chapter discusses imaging and blending extensions. After some introductory information, it looks at each extension in some detail. You learn about:

Introduction to Imaging and Blending Extensions

This section discusses where extensions are in the imaging pipeline established by OpenGL 1.0; it also lists the commands to which imaging extensions apply.

Where Extensions Are in the Imaging Pipeline

The OpenGL 1.0 imaging pipeline is shown in the *OpenGL Programming Guide* in the illustration "Drawing Pixels with glDrawPixels*()" in Chapter 8, "Drawing Pixels, Bitmaps, Fonts, and Images."

You can see how, after the scale and bias operations and after the shift and offset operations, color conversion (LUT in Figure 7-1 below) takes place with a lookup table. After that, the extension modules may be applied. Unless the histogram or minmax extensions were called to collect information only, pixel processing continues, as shown in the *OpenGL Programming Guide.*

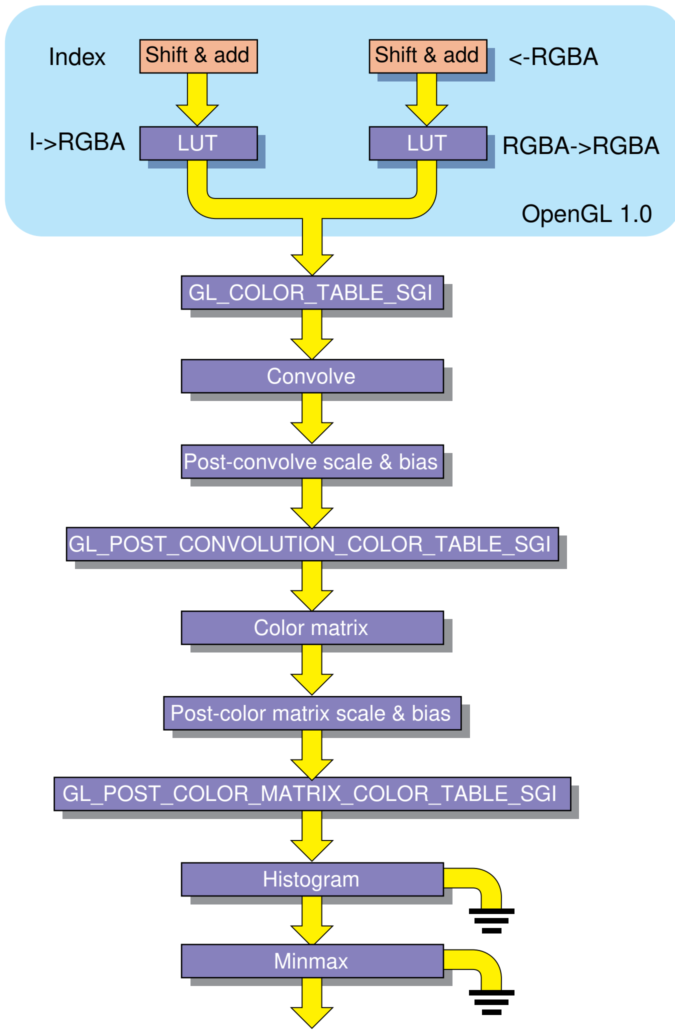

Figure 7-1 How Extensions Fit in the Imaging Pipeline

Functions Affected by Imaging Extensions

Imaging extensions affect all functions that are affected by the pixel transfer modes (see "Storing, Transforming, and Mapping Pixels" in Chapter 8, "Drawing Pixels, Bitmaps, Fonts, and Images," of the *OpenGL Programming Guide).* In general, these are all the commands that draw and copy pixels or define texture images and all commands that read pixels or textures back to host memory.

The ABGR Extension

The ABGR extension, EXT_abgr, extends the list of host-memory color formats by an alternative to the RGBA format that uses reverse component order. The ABGR component order matches the cpack IRIS GL format on big-endian machines. This is the most convenient way to use an ABGR source image with OpenGL.

Note that the ABGR extension provides the best performance on the following graphics systems: Starter, XZ, Elan, XS24, Extreme.

To use this extension, call **glDrawPixels()**, **glGetTexImage()**, **glReadPixels()**, and **glTexImage*()** with GL_ABGR_EXT as the value of the *format* parameter.

The following code fragment illustrates the use of the extension:

```
/*
 *  draw a 32x32 pixel image at location 10, 10 using an ABGR source
 *  image. "image" *should* point to a 32x32 ABGR UNSIGNED BYTE image
 */

{
    unsigned char *image;

    glRasterPos2f(10, 10);
    glDrawPixels(32, 32, GL_ABGR_EXT, GL_UNSIGNED_BYTE, image);
}
```

The Packed Pixels Extension

The packed pixels extension, EXT_packed_pixels, provides support for packed pixels in host memory. A packed pixel is represented entirely by one unsigned byte, unsigned short, or unsigned integer. The fields within the packed pixel are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes, such as GL_PACK_SKIP_PIXELS, GL_PACK_ROW_LENGTH, and so on, and their unpacking counterparts, all work correctly with packed pixels.

Why Use the Packed Pixels Extension?

The packed pixels extension lets you store images more efficiently by providing additional pixel types you can use when reading and drawing pixels or loading textures. Packed pixels may also be faster than unpacked pixels in some cases because less bandwidth (or less processing) is required to transfer them to and from the graphics hardware.

In addition, some of the types defined by this extension match the internal texture formats (see "The Texture Extension" on page 85) so less processing is required to transfer texture images to texture memory.

Using Packed Pixels

To use packed pixels, provide one of the types listed in Table 7-1 as the *type* parameter to **glDrawPixels()**, **glReadPixels()**, and so on.

Table 7-1 Types That Use Packed Pixels

Parameter Token Value	GL Data Type
GL_UNSIGNED_BYTE_3_3_2_EXT	GLubyte
GL_UNSIGNED_SHORT_4_4_4_4_EXT	GLushort
GL_UNSIGNED_SHORT_5_5_5_1_EXT	GLushort
GL_UNSIGNED_INT_8_8_8_8_EXT	GLuint
GL_UNSIGNED_INT_10_10_10_2_EXT	GLuint

The already available types for **glReadPixels()**, **glDrawPixels()**, and so on are listed in Table 8-2 "Data Types for glReadPixels or glDrawPixels," in the *OpenGL Programming Guide*.

What the Different Pixel Types Mean

Each packed pixel type includes a base type, for example GL_UNSIGNED_BYTE, and a field width (for example, 3_3_2):

- The base type, GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT, determines the type of "container" into which each pixel's color components are packed.
- The field widths, 3_3_2, 4_4_4_4, 5_5_5_1, 8_8_8_8, or 10_10_10_2, determine the sizes (in bits) of the fields that contain a pixel's color components. The field widths are matched to the components in the pixel format, in left-to-right order.

  For example, if a pixel has the type GL_UNSIGNED_BYTE_3_3_2_EXT and the format GL_RGB, the pixel is contained in an unsigned byte, the red component occupies three bits, the green component occupies three bits, and the blue component occupies two bits.

  The fields are packed tightly into their container, with the leftmost field occupying the most-significant bits and the rightmost field occupying the least-significant bits.

Because of this ordering scheme, integer constants (particularly hexadecimal constants) can be used to specify pixel values in a readable and system-independent way. For example, a packed pixel with type GL_UNSIGNED_SHORT_4_4_4_4_EXT, format GL_RGBA, and color components red == 1, green == 2, blue == 3, alpha == 4 has the value 0x1234.

The ordering scheme also allows packed pixel values to be computed with system-independent code. For example, if there are four variables (red, green, blue, alpha) containing the pixel's color component values, a packed pixel of type GL_UNSIGNED_INT_10_10_10_2_EXT and format GL_RGBA can be computed with the following C code:

```
GLuint pixel, red, green, blue, alpha;
pixel = (red << 22) | (green << 12) | (blue << 2) | alpha;
```

While the source code that manipulates packed pixels is identical on both big-endian and little-endian systems, you still need to enable byte swapping when drawing packed pixels that have been written in binary form by a system with different endianness.

The Color Matrix Extension

The color matrix extension, SGI_color_matrix, lets you transform the colors in the imaging pipeline with a 4 x 4 matrix. You can use the color matrix to reassign and duplicate color components and to implement simple color-space conversions.

This extension adds a 4 x 4 matrix stack to the pixel transfer path. The matrix operates on RGBA pixel groups, multiplying the 4 x 4 color matrix on top of the stack with the components of each pixel. The stack is manipulated using the OpenGL 1.0 matrix manipulation functions: **glPushMatrix()**, **glPopMatrix()**, **glLoadIdentity()**, **glLoadMatrix()**, and so on. All standard transformations, for example **glRotate()** or **glTranslate()**, also apply to the color matrix.

Below is an example of a color matrix that swaps BGR pixels to form RGB pixels:

```
GLfloat colorMat[16] = {0.0, 0.0, 1.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        1.0, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.0, 0.0 };
glMatrixMode(GL_COLOR);
glPushMatrix();
glLoadMatrixf(colorMat);
```

After the matrix multiplication, each resulting color component is scaled and biased by the appropriate user-defined scale and bias values. Color matrix multiplication follows convolution (and convolution scale and bias).

To set scale and bias values to be applied after the color matrix, call **glPixelTransfer*()** with the following values for *pname:*

- GL_POST_COLOR_MATRIX_{RED/BLUE/GREEN/ALPHA}_SCALE_SGI
- GL_POST_COLOR_MATRIX_{RED/BLUE/GREEN/ALPHA}_BIAS_SGI

The Convolution Extension

The convolution extension, EXT_convolution, allows you to filter images, for example to sharpen or blur them, by convolving the pixel values in a one- or two- dimensional image with a convolution kernel.

The convolution kernels are themselves treated as one- and two- dimensional images. They can be loaded from application memory or from the framebuffer.

Convolution is performed only for RGBA pixel groups, although these groups may have been specified as color indexes and converted to RGBA by index table lookup.

Figure 7-2 shows the equations for general convolution at the top and for separable convolution at the bottom.

$$Dest[i,j] = \sum_{l=0}^{Kh} \sum_{k=0}^{Kw} Kernel[k,l]\, Source[i+k, j+l]$$

$$Dest[i,j] = \sum_{l=0}^{Kh} Vert[l] \sum_{k=0}^{Kw} Horiz[k]\, Source[i+k, l+j]$$

Figure 7-2 Convolution Equations

Performing Convolution

Performing convolution consists of these steps:

1. If desired, specify filter scale, filter bias, and convolution parameters for the convolution kernel. For example:

```
glConvolutionParameteriEXT(GL_CONVOLUTION_2D_EXT,
   GL_CONVOLUTION_BORDER_MODE_EXT,
   GL_REDUCE_EXT /*nothing else supported at present */);
```

```
glConvolutionParameterfvEXT(GL_CONVOLUTION_2D_EXT,
   GL_CONVOLUTION_FILTER_SCALE_EXT,filterscale);
glConvolutionParameterfvEXT(GL_CONVOLUTION_2D_EXT,
   GL_CONVOLUTION_FILTER_BIAS_EXT, filterbias);
```

2. Define the image to be used for the convolution kernel.

Use a 2D array for 2D convolution and a 1D array for 1D convolution. Separable 2D filters consist of two 1D images for the row and the column.

To specify a convolution kernel, call **glConvolutionFilter2DEXT()**, **glConvolutionFilter1DEXT()**, or **glSeparableFilter2DEXT()**.

The following example defines a 7 x 7 convolution kernel that is in RGB format and is based on a 7 x 7 RGB pixel array previously defined as rgbBlurImage7x7:

```
glConvolutionFilter2DEXT(
   GL_CONVOLUTION_2D_EXT,    /*has to be this value*/
   GL_RGB,                   /*filter kernel internal format*/
   7, 7,                     /*width & height of image pixel array*/
   GL_RGB,                   /*image internal format*/
   GL_FLOAT,                 /*type of image pixel data*/
   (const void*)rgbBlurImage7x7      /* image itself*/
   )
```

For more information about the different parameters, see the reference page for the relevant function.

3. Enable convolution.
4. Perform pixel operations (for example pixel drawing or texture image definition).

   Convolution happens as the pixel operations are executed.
5. Use **glGetConvolutionParameter*EXT()** to retrieve the following convolution state parameters:

GL_CONVOLUTION_BORDER_MODE_EXT
: Convolution border mode. For a list of border modes, see **glConvolutionParameterEXT()**.

GL_CONVOLUTION_FORMAT_EXT
: Current internal format. For lists of allowable formats, see **glConvolutionFilter*EXT()**, and **glSeparableFilter2DEXT()**.

GL_CONVOLUTION_FILTER_{BIAS, SCALE}_EXT
: Current filter bias and filter scale factors. *params* must be a pointer to an array of four elements, which receive the red, green, blue, and alpha filter bias terms in that order.

GL_CONVOLUTION_{WIDTH, HEIGHT}_EXT
: Current filter image width.

GL_MAX_CONVOLUTION_{WIDTH, HEIGHT}_EXT
: Maximum acceptable filter image width and filter image height.

Separable and General Convolution Filters

A convolution that uses separable filters operates faster than one that uses general filters.

Special facilities are provided for the definition of two-dimensional separable filters. For separable filters, the image is represented as the product of two one-dimensional images, not as a full two-dimensional image.

To specify a two-dimensional separable filter, call **glSeparableFilter2DEXT()**.

- *target* must be GL_SEPARABLE_2D_EXT.
- *internalformat* specifies the formats of two one-dimensional images that are retained; it must be one of GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_INTENSITY_EXT, GL_RGB, or GL_RGBA.
- *row* and *column* point to two one-dimensional images in memory.
  - The *row* image, defined by *format* and *type*, is *width* pixels wide.
  - The *column* image, defined by *format* and *type*, is *height* pixels wide.

The two images are extracted from memory and processed just as if **glConvolutionFilter1DEXT()** were called separately for each, with the resulting retained images replacing the current 2D separable filter images, except that each scale and bias are applied to each image using the 2D separable scale and bias vectors.

If you're using convolution on a texture image, keep in mind that the result of the convolution must obey the constraint that the dimensions have to be a power of 2. If you use the reduce border convolution mode, the image shrinks by the filter width minus 1, so you may have to take that into account ahead of time.

New Functions

glConvolutionFilter1DEXT(), glConvolutionFilter2DEXT(), glCopyConvolutionFilter1DEXT(), glCopyConvolutionFilter2DEXT(), glGetConvolutionFilterEXT(), glSeparableFilter2DEXT(), glGetSeparableFilterEXT(), glConvolutionParameterivEXT(), glConvolutionParameterfEXT(), glConvolutionParameterfvEXT(), glGetConvolutionParameterivEXT, glGetConvolutionParameterfvEXT().

The Histogram and Minmax Extensions

The histogram extension, EXT_histogram, defines operations that count occurrences of specific color component values and that track the minimum and maximum color component values in images that pass through the image pipeline. You can use the results of these operations to create a more balanced, better-quality image.

Figure 7-3 illustrates how the histogram extension collects information for one of the color components: The histogram has the number of bins specified at creation, and information is then collected about the number of occurrences of that color component. Assuming the example below is for the red component of an image, you can see that R values between 95 and 127 occurred least often and those between 127 and 159 most often.

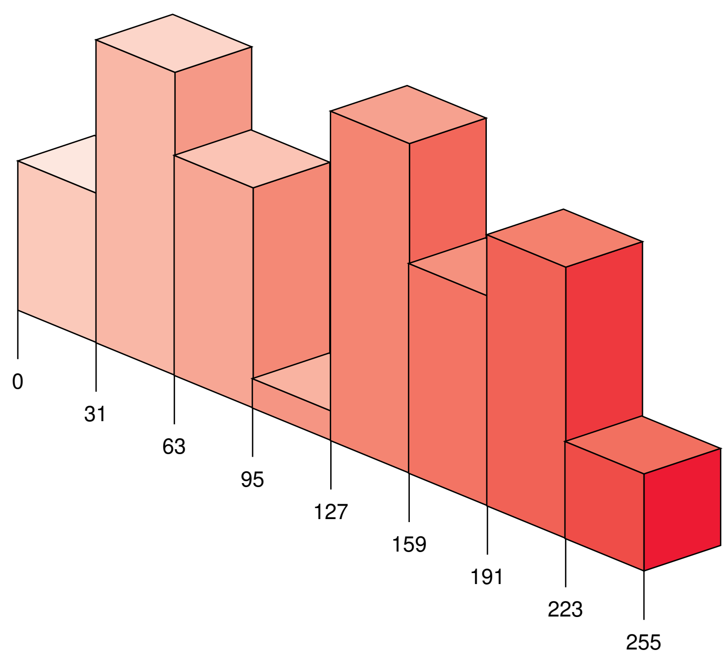

Figure 7-3 How the Histogram Extension Collects Information

Histogram and minmax operations are performed only for RGBA pixel groups, though these groups may have been specified as color indexes and converted to RGBA by color index table lookup.

Using the Histogram Extension

To collect histogram information, follow these steps:

1. Call **glHistogramExt()** to define the histogram, for example:

   ```
   glHistogramEXT(GL_HISTOGRAM_EXT,
                  256              /* width (number of bins) */,
                  GL_LUMINANCE     /* internalformat */,
                  GL_TRUE          /* sink */);
   ```

   - *width*, which must be a power of 2, specifies the number of entries in the histogram.
   - *internalformat* specifies the format of each table entry.
   - *sink* specifies whether pixel groups are consumed by the histogram operation (GL_TRUE) or passed further down the image pipeline (GL_FALSE).

2. Enable histogramming by calling

   ```
   glEnable(GL_HISTOGRAM_EXT)
   ```

3. Perform the pixel operations for which you want to collect information (drawing, reading, or copying pixels).

   For each component represented in the histogram internal format, let the corresponding component of the incoming pixel (luminance corresponds to red) be of value c (after clamping to [0, 1). The corresponding component of bin number `floor((width-1)*c)` is incremented by 1.

4. Call **glGetHistogramEXT()** to query the current contents of the histogram:

   ```
   void glGetHistogramEXT( GLenum target, GLboolean reset, GLenum format,
             GLenum type, GLvoid *values )
   ```

   - *target* must be GL_HISTOGRAM_EXT.
   - *reset* is either GL_TRUE or GL_FALSE. If GL_TRUE, each component counter that is actually returned is reset to zero. Counters that are not returned are not modified.
   - *format* must be one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGBA, GL_RGB, GL_ABGR_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA.
   - *type* must be GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.

- *values* is used to return a 1D image with the same width as the histogram. No pixel transfer operations are performed on this image, but pixel storage modes that apply for **glReadPixels()** are performed. Color components that are requested in the specified *format*, but are not included in the internal format of the histogram, are returned as zero. The assignments of internal color components to the components requested by *format* are as follows:

internal component	resulting component
red	red
green	green
blue	blue
alpha	alpha
luminance	red/luminance

Using the Minmax Part of the Histogram Extension

The minmax part of the histogram extension lets you find out about minimum and maximum color component values present in an image. Using the minmax part of the histogram extension is similar to using the histogram part.

To determine minimum and maximum color values used in an image, follow these steps:

1. Specify a minmax table by calling **glMinmaxEXT()**.

   ```
   void glMinmaxEXT( GLenum target, GLenum internalformat, GLboolean sink)
   ```

   - *target* is the table in which the information about the image is to be stored. *target* must be GL_MINMAX_EXT.
   - *internalformat* specifies the format of the table entries. It must be an allowed internal format (see the reference page for **glMinmaxEXT**).
   - *sink* is set to GL_TRUE or GL_FALSE. If GL_TRUE, no further processing happens and pixels or texels are discarded.

   The resulting minmax table always has two entries. Entry 0 is the minimum and entry 1 is the maximum.

2. Enable minmax by calling

   ```
   glEnable(GL_MINMAX_EXT)
   ```

3. Perform the pixel operations, for example, **glCopyPixels()**.

Each component of the internal format of the minmax table is compared to the corresponding component of the incoming RGBA pixel (luminance components are compared to red).

- If a component is greater than the corresponding component in the maximum element, then the maximum element is updated with the pixel component value.
- If a component is smaller than the corresponding component in the minimum element, then the minimum element is updated with the pixel component value.

4. Query the current context of the minmax table by calling **glGetMinMaxExt()**:

```
void glGetMinMaxEXT ( GLenum target, GLboolean reset, GLenum format,
                      GLenum type, glvoid *values)
```

You can also call **glGetMinmaxParameterEXT()** to retrieve minmax state information; setting *target* to GL_MINMAX_EXT and *pname* to one of the following values:

GL_MINMAX_FORMAT_EXT
: internal format of minmax table

GL_MINMAX_SINK_EXT
: value of *sink* parameter

Using Proxy Histograms

Histograms can easily get quite large and require more memory than is available to the graphics subsystem. You can call **glHistogramEXT()** with *target* set to GL_PROXY_HISTOGRAM_EXT to find out whether a histogram fits into memory. The process is similar to that discussed in “The Texture Extension Proxy Mechanism” on page 89 and illustrated in Example 6-1.

To query histogram state values, call **glGetHistogramParameter*EXT()**. Histogram calls with the proxy target (like texture and color table calls with the proxy target) have no effect on the histogram itself.

New Functions

glHistogramEXT(), glResetHistogramEXT(), glGetHistogramEXT(), glGetHistogramParameterivEXT(), glGetHistogramParameterfvEXT() glMinmaxEXT(), glGetMinmaxEXT(), glGetMinmaxParameterivEXT(), glGetMinmaxParametervEXT().

The Color Table Extension

The color table extension, SGI_color_table, defines a new RGBA-format color lookup mechanism. It does not replace the color lookup tables provided by the color maps discussed in the *OpenGL Programming Guide* but provides additional lookup capabilities.

If the copy texture extension is implemented, this extension also defines methods to initialize the color lookup tables from the framebuffer.

Why Use the Color Table Extension?

The color tables provided by the color table extension allow you to adjust image contrast and brightness after each stage of the pixel processing pipeline.

Because you can use several color lookup tables at different stages of the pipeline (see Figure 7-1), you have greater control over the changes you want to make. In addition the extension color lookup tables are more efficient than those of OpenGL 1.0 because you may apply them to a subset of components (for example, Alpha only).

Specifying a Color Table

To specify a color lookup table, call **glColorTableSGI()**:

```
void glColorTableSGI( GLenum target, GLenum internalformat, GLsizei width,
                      GLenum format, GLenum type,const GLvoid *table)
```

- *target* must be GL_COLOR_TABLE_SGI, GL_POST_CONVOLUTION_COLOR_TABLE_SGI, or GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI.
- *internalformat* is the internal format of the color table.

- *width* specifies the number of entries in the color lookup table. It must be zero or a non-negative power of two.
- *format* specifies the format of the pixel data in the table.
- *type* specifies the type of the pixel data in the table.
- *table* is a pointer to a 1D array of pixel data that is processed to build the table.

If no error results from the execution of **glColorTableSGI()**, the following events occur:

1. The specified color lookup table is defined to have *width* entries, each with the specified internal format. The entries are indexed as zero through N-1, where N is the width of the table. The values in the previous color lookup table, if any, are lost. The new values are specified by the contents of the one-dimensional image that *table* points to, with *format* as the memory format and *type* as the data type.
2. The specified image is extracted from memory and processed as if **glDrawPixels()** were called, stopping just before the application of pixel transfer modes (see the illustration "Drawing Pixels with glDrawPixels*()" in Chapter 8, "Bitmaps, Fonts, and Images," of the *OpenGL Programming Guide*).
3. The R, G, B, and A components of each pixel are first scaled by the four GL_COLOR_TABLE_SCALE_SGI parameters, then biased by the four GL_COLOR_TABLE_BIAS_SGI parameters and clamped to [0,1].

   The scale and bias parameters are themselves specified by calling **glColorTableParameterivSGI()** or **glColorTableParameterfvSGI()**:

   - *target* specifies one of the three color tables: GL_COLOR_TABLE_SGI, GL_POST_CONVOLUTION_COLOR_TABLE_SGI, or GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI.
   - *pname* has to be GL_COLOR_TABLE_SCALE_SGI or GL_COLOR_TABLE_BIAS_SGI.
   - *params* points to a vector of four values: red, green, blue, and alpha, in that order.
4. Each pixel is then converted to have the specified internal format. This conversion maps the component values of the pixel (R, G, B, and A) to the values included in the internal format (red, green, blue, alpha, luminance, and intensity).

The new lookup tables are treated as 1-dimensional images with internal formats, like texture images and convolution filter images. As a result, the new tables can operate on a subset of the components of passing pixel groups. For example, a table with internal format GL_ALPHA modifies only the A component of each pixel group, leaving the R, G, and B components unmodified.

Using Framebuffer Image Data for Color Tables

If the copy texture extension is supported, you can define a color table using image data in the framebuffer. Call **glCopyColorTableSGI()**, which accepts image data from a color buffer region (*width* pixel wide by one pixel high) whose left pixel has window coordinates *x,y*. If any pixels within this region are outside the window that is associated with the OpenGL context, the values obtained for those pixels are undefined.

The pixel values are processed exactly as if **glCopyPixels()** had been called, until just before the application of pixel transfer modes (see the illustration "Drawing Pixels with glDrawPixels*()" in Chapter 8, "Bitmaps, Fonts, and Images," of the *OpenGL Programming Guide*).

At this point all pixel component values are treated exactly as if **glColorTableSGI()** had been called, beginning with the scaling of the color components by GL_COLOR_TABLE_SCALE_SGI. The semantics and accepted values of the *target* and *internalformat* parameters are exactly equivalent to their **glColorTableSGI()** counterparts.

Where Are the Lookup Tables in the Image Pipeline?

The three lookup tables exist at different points in the image pipeline (see Figure 7-1):

- GL_COLOR_TABLE_SGI is located immediately after index lookup or RGBA to RGBA mapping, and immediately prior to the convolution operation.
- GL_POST_CONVOLUTION_COLOR_TABLE_SGI is located immediately after the convolution operation (including its scale and bias operations) and immediately prior to the color matrix operation.
- GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI is located immediately after the color matrix operation (including its scale and bias operations) and immediately prior to the histogram operation.

To enable and disable color tables, call **glEnable()** and **glDisable()** with the color table name passed as the *cap* parameter. Color table lookup is performed only for RGBA groups, though these groups may have been specified as color indexes and converted to RGBA by an index-to-RGBA pixel map table.

When enabled, a color lookup table is applied to all RGBA pixel groups, regardless of the command that they are associated with.

New Functions

glColorTableSGI(), glColorTableParameterivSGI(), glGetColorTableSGI(), glGetColorTableParameterivSGI(), glGetColorTableParameterfvSGI().

The Interlace Extension

The interlace extension, SGIX_interlace, provides a way to interlace rows of pixels when rasterizing pixel rectangles, or loading texture images. In this context, interlacing means skipping over rows of pixels or texels in the destination. This is useful for dealing with interlace video data since single frames of video are typically composed of two fields: one field specifies the data for even rows of the frame, the other specifies the data for odd rows of the frame.

Caution: This extension is an SGIX (experimental) extension. The interface may change, or it may be dropped altogether.

Using the Interlace Extension

To turn interlacing on or off, call **glEnable()** or **glDisable()** with the *cap* parameter GL_INTERLACE_SGIX. When enabled, it modifies the behavior of **glDrawPixels()**, **glCopyPixels()**, **glTexImage2D()**, **glTexSubImage2DEXT()**, **glCopyTexImage2DEXT()** and **glCopyTexSubImage2DEXT()** as well as the 3D versions of texture calls.

If GL_INTERLACE_SGIX is enabled, all the groups that belong to a row m are treated as if they belonged to the row $2\times m$. If the source image has a height of h rows, this effectively expands the height of the image to $2\times h$ rows.

After interlacing, only every other row of the image is defined:

- If the interlaced pixel rectangle is rasterized to the framebuffer, then only those rows are converted to fragments.

- If the interlaced pixel rectangle is a texture image, then only those rows are written to texture memory.

In cases where errors can result from the specification of invalid image dimensions, the resulting dimensions—not the dimensions of the source image—are tested.

For example, when you use **glTexImage2D()** with GL_INTERLACE_SGIX enabled, the source image you provide must be of height `(texture_height + texture_border)/2`.

How to Use Interlace and Copy Texture Together

One application of the interlace extension is to use it together with the copy texture extension: You can use **glCopyTexSubImage2DEXT()** to copy the contents of the video field to texture memory and end up with de-interlaced video.

You can interlace pixels from two images as follows:

```
glEnable(GL_INTERLACE_SGIX);
< set current raster position to (xr,yr) >
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I0);
< set raster position to (xr,yr+zoomy) >
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I1);
```

This process is equivalent to taking pixel rows (0,2,4,...) of I2 are from image I0, and rows (1,3,5,...) from image I1, as follows:

```
glDisable( GL_INTERLACE_SGIX);
< set current raster position to (xr,yr)>
glDrawPixels(width, 2*height, GL_RGBA, GL_UNSIGNED_BYTE, I2);
```

Blending Extensions

Blending refers to the process of combining color values from a source (an incoming pixel fragment) with current values of the stored pixel in the framebuffer (the destination). The final effect is that parts of a scene appear translucent. In OpenGL 1.0, you specify the blending operation by calling **glBlendFunc()**, then enable or disable blending using **glEnable()** or **glDisable()** with GL_BLEND.

Blending is discussed in the first section of Chapter 7, "Blending, Antialiasing, and Fog," of the *OpenGL Programming Guide*. The section also lists a number of sample uses of blending.

This section explains how to use extensions that support color blending for images and rendered geometry in a variety of ways:

- "The Constant Color Blending Extension"
- "The Minmax Blending Extension"
- "The Logical Operation Blending Extension"
- "The Blend Subtract Extension"

The Constant Color Blending Extension

The standard blending feature allows you to blend source and destination pixels. The constant color blending extension, EXT_blend_color, enhances this capability by defining a constant color that you can include in blending equations.

Constant color blending allows you to specify input source with constant alpha that is not 1 without actually specifying the alpha for each pixel. Alternatively, when working with visuals that have no alpha, you can use the blend color for constant alpha. This also allows you to modify a whole incoming source by blending with a constant color (which is faster than clearing to that color). In effect, the image looks as if it is viewed through colored glasses.

Using Constant Colors for Blending

To use a constant color for blending, follow these steps:

1. Call **glBlendColorEXT()** to specify the blending color:

   ```
   void glBlendColorEXT( GLclampf red, GLclampf green, GLclampf blue,
                         GLclampf alpha )
   ```

   The four parameters are clamped to the range [0,1] before being stored. The default value for the constant blending color is (0,0,0,0).

2. Call **glBlendFunc()** to specify the blending function, using one of the tokens listed in Table 7-2 as source or destination factor, or both.

Table 7-2 Blending Factors Defined by the Blend Color Extension

Constant	Computed Blend Factor
GL_CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)
GL_ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)
GL_CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)
GL_ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)

Rc, Gc, Bc, and Ac are the four components of the constant blending color. These blend factors are already in the range [0,1].

You can, for example, fade between two images by drawing both images with Alpha and 1-Alpha as Alpha goes from 1 to 0, as in the following code fragment:

```
glBlendFunc(GL_ONE_MINUS_CONSTANT_COLOR_EXT, GL_CONSTANT_COLOR_EXT);
for (alpha = 0.0; alpha <= 1.0; alpha += 1.0/16.0) {
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, image0);
    glEnable(GL_BLEND);
    glBlendColorEXT(alpha, alpha, alpha, alpha);
    glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, image1);
    glDisable(GL_BLEND);
    glXSwapBuffers(display, window);
    }
```

New Functions

glBlendColorEXT().

The Minmax Blending Extension

The minmax blending extension, EXT_blend_minmax, extends blending capability by introducing two new equations that produce the minimum or maximum color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

This extension also introduces a mechanism for defining alternate blend equations. Note that even if the minmax blending extension is not supported on a given system, that system may still support the logical operation blending extension or the subtract blending extension. When these extensions are supported, the **glBlendEquationEXT()** function is also supported.

Using a Blend Equation

To specify a blend equation, call **glBlendEquationEXT()**:

```
void glBlendEquationEXT(GLenum mode)
```

The *mode* parameter specifies how source and destination colors are combined. The blend equations GL_MIN_EXT, GL_MAX_EXT, and GL_LOGIC_OP_EXT use source or destination factors.

If *mode* is set to GL_FUNC_ADD_EXT, then the blend equation is set to GL_ADD, the equation used currently in OpenGL 1.0. The **glBlendEquationEXT()** reference page lists other modes. These modes are also discussed in "The Blend Subtract Extension" on page 144 and "The Logical Operation Blending Extension" on page 143. While OpenGL 1.0 defines logic operation only on color indices, this extension extends the logic operation to RGBA pixel groups. The operation is applied to each component separately.

New Functions

glBlendEquationEXT()

The Logical Operation Blending Extension

The logical operation blending extension, EXT_blend_logic_op, specifies a single additional blending equation, using the **glBlendEquationEXT()** call discussed in "The Minmax Blending Extension" on page 142. The blending equation is a logical combination of the source and destination colors. The specific logical operation is specified by **glLogicOp()**, which is discussed in the section "Logical Operations" in Chapter 10, "The Framebuffer" of the *OpenGL Programming Guide.*

A possible application of this extension is for annotations, for example, assume you want to annotate a map without redrawing the map itself. To make the annotations visible, XOR them with the map to make them disappear again, XOR them once more. You should XOR the most significant bit of the color.

If you call **glBlendEquationEXT()** with *mode* set to GL_LOGIC_OP, the blending equation becomes

```
C = Cs OP Cd
```

Cs and Cd are the source and destination colors, and OP is the logic operation as specified by **glLogicOp()**. The value of the boolean parameter GL_LOGIC_OP, specified by **glEnable()** and **glDisable()**, has no effect on blending.

The Blend Subtract Extension

The blend subtract extension, EXT_blend_subtract, provides two additional blending equations that can be used by **glBlendEquationEXT()**. These equations are similar to the default blending equation, but produce the difference of its left- and right-hand sides, rather than the sum. See the reference page for **glBlendEquationEXT()** for a detailed description.

Image differences are useful in many image-processing applications; for example, comparing two pictures that may have changed over time.