

Using OpenGL[®] for Imaging

Randi J. Rost

Hewlett-Packard Company

ABSTRACT

Although OpenGL is not usually thought of as a library for imaging, it was designed to expose the capabilities of modern frame buffer hardware. The emphasis in OpenGL is on 3D graphics (i.e., geometry), but OpenGL also includes a fairly rich set of capabilities for 2D imaging. This paper describes the capabilities of OpenGL for imaging applications, including pixel transfer operations (draw, read, copy); color lookup tables; linear transformation of color values; pixel conversion capabilities; and pixel operations such as blending, masking, and clipping. Several recently proposed extensions to OpenGL add significant capabilities to the core imaging model, including convolution, window level mapping, and image transformation and resampling. These capabilities will be discussed in the context of the pixel processing pipeline defined by OpenGL.

KEYWORDS

Image display system, imaging API, OpenGL, display algorithms, interactive image processing

INTRODUCTION

OpenGL is rapidly gaining popularity with both application programmers and hardware vendors as a strategic application programming interface (API) for graphics and imaging. Application developers are embracing OpenGL because of its rich and flexible feature set, and because it is supported on a wide range of workstations and PC platforms. Hardware vendors support OpenGL because it provides a way to expose the features and the full performance of their graphics hardware products.

Although its design center was 3D graphics, OpenGL includes support for a variety of visualization capabilities, including the display and manipulation of 2D image data. However, the

imaging capabilities in OpenGL have not received as much attention as they have deserved for a number of reasons. Because of the emphasis on 3D graphics, many people assume that OpenGL only supports 3D. The terminology used to describe the imaging capabilities in OpenGL is somewhat different than the terminology commonly used in imaging markets, and some of OpenGL's imaging capabilities are disguised as 3D graphics capabilities. The pixel processing pipeline in OpenGL has not been discussed as widely or as frequently as the 3D geometry pipeline. This paper attempts to document the imaging capabilities of OpenGL so that they can be clearly understood by people with a background in imaging rather than 3D graphics.

Using OpenGL for imaging has several benefits. First, imaging and 3D graphics are integrated seamlessly by definition within a single API. Vendors are also working to provide volume rendering capabilities in OpenGL, so OpenGL holds promise as the API most likely to provide an integrated environment for dealing with geometry, images, and volumetric data. Second, using OpenGL for imaging precludes the need for a (perhaps mostly redundant) low-level imaging interface based on an alternative API or rendering model. Third, since OpenGL will typically be the lowest level API available on many platforms, it is reasonable to assume that it will be optimized for maximum utilization of hardware acceleration. Also, since many vendors will implement direct hardware access as part of OpenGL, by definition all OpenGL imaging features will have direct hardware access as well. Finally, OpenGL is supported on a broader range of platforms than any other graphics API: from PC's to graphics superworkstations.

The current OpenGL standard [1] does not contain all of the capabilities of interest to people writing imaging applications. A number of key imaging capabilities are currently defined as extensions to OpenGL [2-9,15-18]. In this paper, we will first discuss the imaging capabilities that are available in the current OpenGL standard, and in later sections we will discuss the capabilities available through extensions to OpenGL that are supported by some vendors. OpenGL entry points that have an "EXT" suffix belong to an OpenGL extension that has public support from at least two OpenGL vendors. Entry

Author's address: Hewlett-Packard Company, MS-74, 3404
E. Harmony Road, Fort Collins, CO 80525
rost@tortola.fc.hp.com

points with a suffix such as “HP” or “SGI” are currently supported by a single vendor. OpenGL entry points with no such suffix are part of the current OpenGL standard.

To simplify the presentation somewhat, the “gl” and “GL” prefixes are omitted from procedure names and constants in the following discussion. This discussion also omits some features of the OpenGL pixel processing pipeline that are of marginal usefulness to mainstream imaging applications.

In order that it might be supported in a variety of windowing environments, OpenGL was designed to be window-system neutral. The set of functions that effectively binds OpenGL to the operating environment of the X Window System is called GLX [9]. Similar sets of functions are used to bind OpenGL in the Microsoft Windows[®] environment and the OS/2[®] environment. For simplicity, we will limit our discussion to OpenGL and GLX. This paper describes the capabilities of OpenGL V1.1 [1] and GLX V1.1 [10].

THE OPENGL PIXEL PROCESSING PIPELINE

The imaging capabilities in OpenGL are aimed at transferring pixels to, from, and within the frame buffer. The focus is on the interactive display and manipulation of image data. Exotic or compute-intensive image processing algorithms are left to higher levels of software.

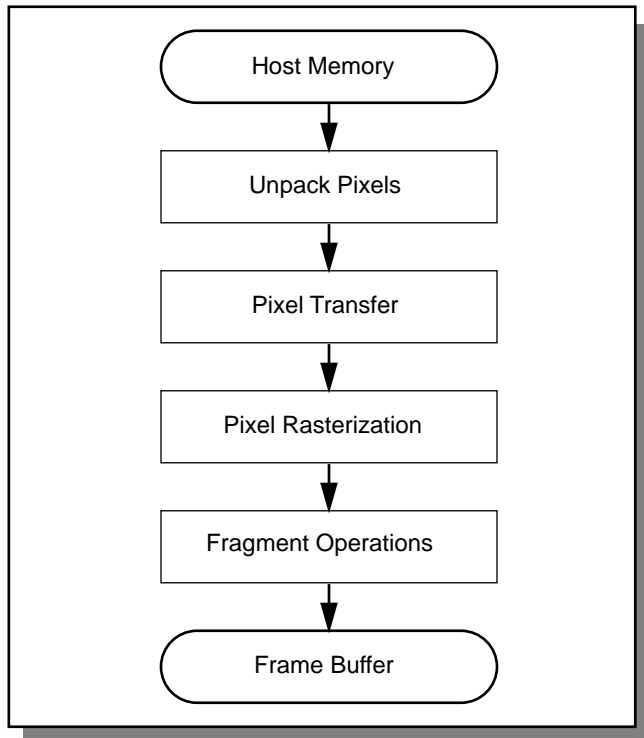


Figure 1: Block diagram of the operations in *DrawPixels*

Figure 1 shows the sequence of operations that occur when the OpenGL *DrawPixels* routine is called. This routine is the primary interface for transferring image data from host memory to the frame buffer. We will discuss in detail each of the processing steps that occur as pixels are transferred from host memory to frame buffer memory.

PIXEL UNPACKING

The *DrawPixels* routine provides applications with a fair amount of flexibility in describing how image data is stored in host memory. A *component* is the fundamental building block for a pixel value. For instance, an RGBA pixel is made up of red, green, blue, and alpha components. A pixel value may consist of one, two, three, or four components, and the number of components is communicated through the *format* argument to *DrawPixels*. The formats that may be specified include RED, GREEN, BLUE, ALPHA, RGB, RGBA, LUMINANCE, and LUMINANCE_ALPHA.

The actual data type of each pixel component is specified using the *type* argument to *DrawPixels*. The type can be one of BITMAP, FLOAT, or signed and unsigned versions of BYTE, SHORT, and INT. The combination of these eight types and eight formats results in 64 possible organizations for a pixel value in host memory.

The pixels in memory can be thought of as a rectangle that is arranged in a series of rows. The pointer passed to *DrawPixels* points to the first component in the first pixel of the first row of image data. If all attributes, including the pixel unpacking attributes, are in their default state, the first pixel in host memory becomes the lower left corner of the displayed image.

Additional flexibility is available if the *PixelStore* routine is used to specify pixel unpacking attributes. The UNPACK_ROW_LENGTH attribute can be used to override the number of pixels in each row. If UNPACK_ROW_LENGTH is 0, the number of pixels in each row is assumed to be equal to the *width* parameter passed to *DrawPixels*, otherwise the number of pixels in each row is UNPACK_ROW_LENGTH. This attribute would typically be used to indicate the extraction of a subimage with rows that are shorter than the real image stored in memory. This attribute indicates the number of pixels in each row, not the number of bytes.

You can also skip a number of rows before reading the first row, and skip a number of pixels in that row (and each subsequent row) prior to the first pixel that is to be transferred to the display. The number of rows to be skipped is established by setting UNPACK_SKIP_ROWS and the number of pixels to be skipped in each row is specified with UNPACK_SKIP_PIXELS. Figure 2 shows how the arguments to *DrawPixels* are used together with these three pixel unpacking attributes in order to extract image data from host memory.

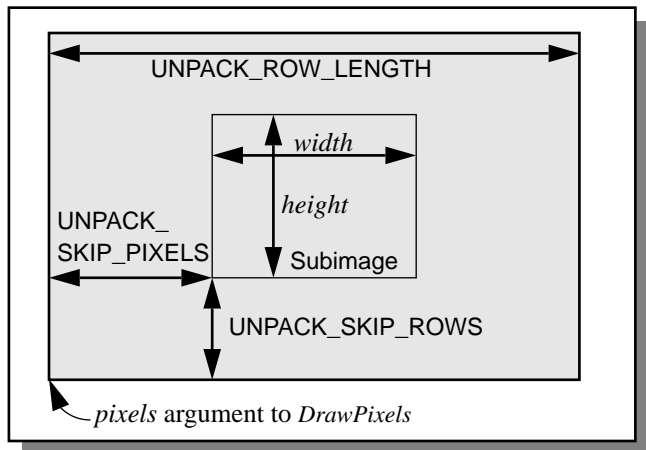


Figure 2: Effect of pixel unpacking attributes

Another pixel unpacking attribute is UNPACK_ALIGNMENT. This attribute allows you to specify whether each row begins at a memory address that is a multiple of 1, 2, 4, or 8. On most architectures, multiple word transfers are more efficient if they begin on a word or double-word boundary. Since image data is arbitrary width, this attribute lets you pad out each row of image data with some unused bytes in order to allow for more efficient data transfer. This unpacking alignment attribute should be set to 8 if each row begins at an address that is a multiple of 8, to 4 if each row begins at an address that is a multiple of 4, and so on.

The UNPACK_LSB_FIRST attribute is a boolean value that controls the interpretation of bitmap image data. If TRUE, the 8 single-bit elements in each byte of image data are ordered from least significant bit to the most significant bit.

The final pixel unpacking attribute is UNPACK_SWAP_BYTES. This attribute is used to automatically reorder the bytes of an image in host memory that was created on a machine with byte ordering that is different from the current machine.

The operations that occur during the “Unpack Pixels” stage of the pixel processing pipeline are shown in detail in Figure 3. There are three main steps that go into unpacking pixels. First, the arguments to *DrawPixels* together with the pixel unpacking attributes are used to extract pixels from host memory as described above.

In order to make it easier to specify subsequent operations on these pixel values, the next step is to convert all pixel components to floating point values in the range [0,1]. This conversion is shown as the second box in Figure 3. (Note that this is a conceptual step in order to make it easier to specify the semantics of subsequent imaging operations. Real implementations can be optimized to avoid this conversion step if it is unnecessary.) After completing this step, all pixel components are the same type and can be treated equally.

The final step in unpacking pixels is to convert all values to RGBA. Luminance values are converted to RGB values by

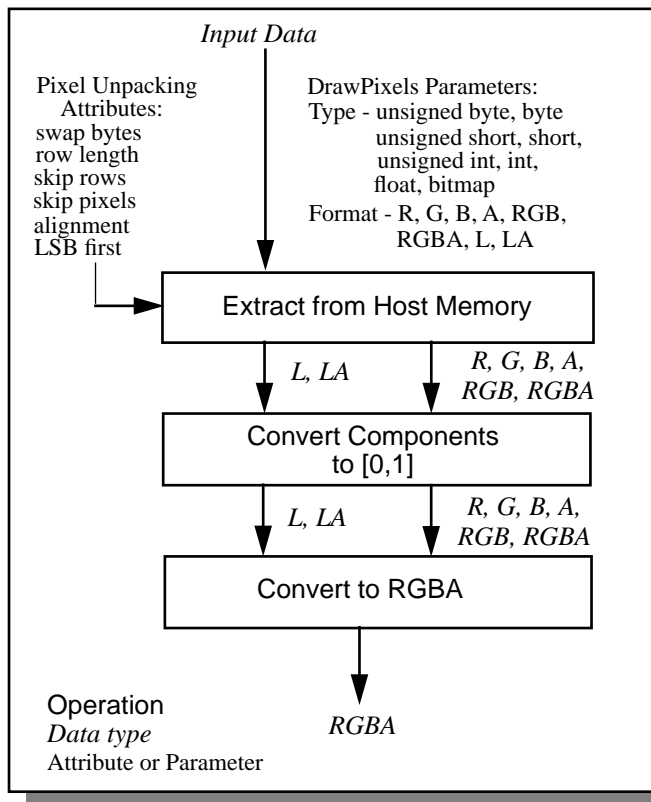


Figure 3: Details of the pixel unpacking operation

assigning the luminance value to each of the red, green, and blue pixel components. If no alpha value has been provided, it is added and assigned a value of 1.0. If any of red, green, and blue are missing, each missing element is added and assigned a value of 0.0. As shown in Figure 3, the result of pixel unpacking is a stream of pixels that are uniform in type and format. (The expansion to RGBA is also a conceptual step that makes it easier to succinctly describe the operations that follow. It can be optimized or eliminated entirely in a real implementation if the expansion is unnecessary given the current state.)

The operations described in this section are only applied when pixels are transferred from host memory to frame buffer memory (e.g., when *DrawPixels* is called). They are not applied when pixels are read from the frame buffer or copied within the frame buffer.

PIXEL TRANSFER

Although the term “pixel transfer” is somewhat ambiguous, it is used to refer to the set of operations illustrated in Figure 4. These operations are applied whenever pixels are drawn

(*DrawPixels*), read (*ReadPixels*), or copied (*CopyPixels*). Most applications will use these operations only when drawing pixels (transferring pixels from host memory to the frame buffer). It is important to remember to disable any of the capabilities that are not needed before performing a pixel read or copy operation.

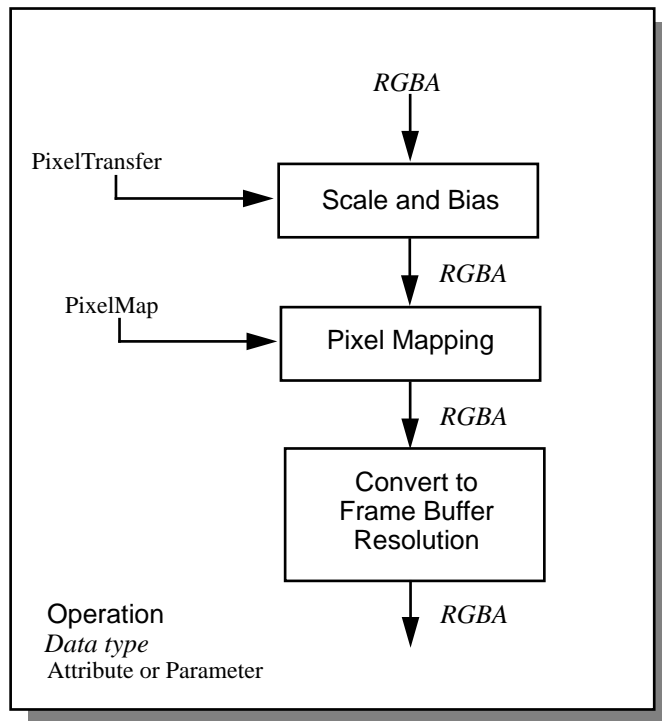


Figure 4: Details of the pixel transfer operation

The first step in the pixel transfer operation is to scale and bias each component. The *PixelTransfer* routine can be used to specify separate scale and bias factors for red, green, blue, and alpha. In addition, this routine can be used to specify whether the scale/bias is enabled or disabled. If the scale/bias is enabled, each component is multiplied by the appropriate signed scale factor and added to the appropriate signed bias value to produce a new value for the component.

This step has a number of useful applications. If the dynamic range of an image is known *a priori*, this operation can be used to enlarge or reduce the image's dynamic range as it is rendered. One common use is to convert signed pixel values in the range [-1,1] into unsigned pixel values in the range [0,1]. Also, in some imaging markets, 12-bit components are widely used. If 12-bit pixel values are stored in the low-order 12 bits of unsigned shorts (16 bits on a typical workstation), the conversion steps in the pixel unpacking stage will result in all the pixels being in the range [0,0.0625] (since only one-sixteenth of the dynamic range of the unsigned short will be used). A scale factor of 16.0 and a bias of 0.0 for each component can be used to map the component values to the full [0,1] range.

This is important for lookup tables and other operations that follow.

The second step in the pixel transfer operation is a lookup table operation. The *PixelMap* routine can be used to specify arrays of component values for each of red, green, blue, and alpha using the *PixelMap* routine. It can also be used to specify whether this mapping operation is enabled or disabled. If it is enabled, each incoming component value is used as an index into the corresponding array in order to produce a new component value. To accomplish this, each component is first clamped to the range [0,1]. Next, each component is multiplied by an integer one less than the size of the corresponding array. The result is rounded to the nearest integer value. This computed index value is used as an index into the corresponding array, and the addressed value becomes the new component value. Pixel mapping is useful for changing the contrast in an image in order to emphasize subtle details or to compensate for artifacts in the image acquisition system. It also provides a powerful mechanism for performing nonlinear pointwise image operations.

The final step of the pixel transfer stage involves converting pixel values from their internal floating point representation into values that map to the entire range supported by the destination color buffer. First, pixel components are clamped to the range [0, 1]. Next, if the destination buffer has m bits, each component c is converted to the fixed point value k where $k \in \{0, 1, \dots, 2^m - 1\}$ and the fraction $k / (2^m - 1)$ is closer to c than any other value.

Another way of looking at this is with a concrete example. If your destination color buffer is 8 bits, then you have $2^8 = 256$ different values that are possible. The floating point values in the range [0, 1] are mapped onto the 256 frame buffer values with 0.0 being mapped to frame buffer value 0 and 1.0 mapped to frame buffer value 255. The range [0, 1] is effectively divided into 256 bands, and any floating point value in band k will get mapped to a frame buffer value of k .

As we'll see in a later section, much has been added to the pixel transfer operation by extensions to OpenGL.

PIXEL RASTERIZATION

Rasterization is the process of converting a rendering primitive into a collection of window coordinate values and the pixel values to be written at those locations. You can think of rasterization as consisting of two steps: first you must determine all of the pixel locations in a window that will be affected by rendering the primitive, and then you must determine what value is to be written at each of the affected locations. The data structures that are generated by the rasterization process are called fragments. Each fragment consists of a frame buffer location and the color, depth, and tex-

ture values that will be used to determine how that frame buffer location will be modified.

One piece of information that is needed in order to display an image is the location on the screen at which to display it. The location at which to draw an image is called the raster position. The raster position is stored as current state and is specified with the *RasterPos* command. The value passed to *RasterPos* is treated like as if it were a vertex. It is transformed by the current model-view matrix and projection matrices. The resulting coordinates are passed to the OpenGL clipping operation just as if they represented a point. If the coordinates represent a point that would not be clipped, a flag is set indicating that the raster position is valid.

This method of specifying the position at which to draw an image may seem complex, but it does provide the capability for using the raster position to do things in OpenGL besides positioning images. The default model-view and projection matrices are the identity matrix, so if you never modify these matrices, you can simply specify the raster position in window coordinates and it will always be valid.

OpenGL defines a rudimentary scaling operation that affects the rasterization of images. The *PixelZoom* routine can be used to specify scale values for enlarging or shrinking an image. The image can be reflected about the current raster position if negative scale factors are used. However, applications are given no control over the reconstruction method used after scaling. For this and other reasons, its usefulness is limited for true imaging applications.

FRAGMENT OPERATIONS

In the OpenGL rendering process, a lot of work remains even after fragments are generated by rasterization. It is worth pointing out that, although the operations performed to get to this point in the rendering pipeline are different for 3D geometry than for image data, from this point on the fragments generated by either pipeline are treated exactly the same. Figure 5 shows the details of the steps that are collectively referred to as “fragment operations.” Some of the fragment operations that are not typically relevant to the pixel processing pipeline (texturing, fog, depth testing) have been omitted from this diagram of the OpenGL fragment operations.

Clearing Buffers

OpenGL maintains a current “clear” value for each type of buffer it defines. The two clear values of interest to imaging applications are the values used when clearing the color and stencil buffers. The value to be used when clearing the color buffers is set by calling *ClearColor*. The value to be used when clearing the stencil buffer is set by calling *ClearStencil*. Buffers are cleared by calling *Clear* with a bitmask indicating which of the buffers should be cleared.

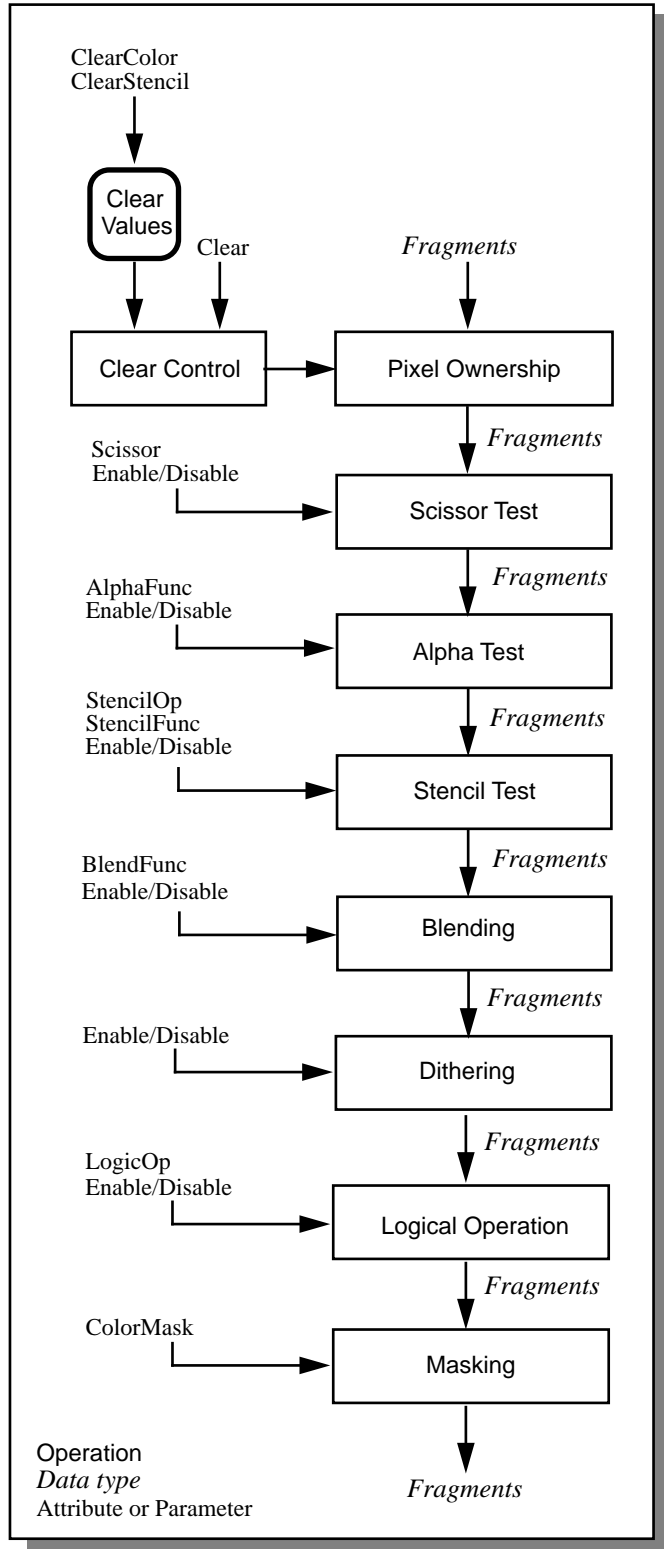


Figure 5: Details of fragment operations

The behavior of the *Clear* call is not represented accurately by the diagram in Figure 5. The fragments generated by this call will be subjected to only the pixel ownership test, the scissor

test, dithering, and masking. Furthermore, the scissor test and dithering will be applied only if they are enabled at the time of the call to *Clear*.

Pixel Ownership

Fragments that are generated by rasterization or by a call to *Clear* are subjected to the pixel ownership test. This test determines whether the location at which the fragment is to be written can be modified by the current OpenGL context. It may be that another window is obscuring the location that is to be modified. If the location cannot be modified, the window system will decide the fate of the incoming fragment. (Most implementations will discard the fragment.) In the X Window System, this is called window clipping. The pixel ownership test cannot be disabled, since it ensures that applications will behave as “good citizens” within the current windowing environment.

Scissor Test

There are several OpenGL features that lend themselves to the definition of a “region of interest.” One simple mechanism is the scissor box. This is simply a rectangular region defined in window coordinates. Only pixels that are within the scissor box are eligible for modification when the scissor test is enabled. The scissor test may be enabled or disabled by passing the value `SCISSOR_TEST` to the functions *Enable* and *Disable*. The values for the scissor box are set by calling *Scissor*.

Alpha Test

OpenGL defines an alpha test that is applied after the destination pixel is determined to be writable via the pixel ownership and scissor tests. The alpha test compares the alpha value associated with the incoming fragment to an alpha value that is stored as part of the current OpenGL state. The result of this comparison determines whether the incoming pixel value is rejected or continues on for further processing. The comparison to be used is specified by calling *AlphaFunc* with one of the constants `NEVER`, `LESS`, `EQUAL`, `LEQUAL`, `GREATER`, `NOTEQUAL`, `GEQUAL`, or `ALWAYS`. The alpha test itself may be enabled or disabled by calling *Enable* or *Disable* with the constant `ALPHA_TEST`. This capability allows an application to easily draw an image to the frame buffer with a mask, if that mask is stored in the image’s alpha planes.

Stencil Test

The most powerful method for specifying a region of interest in OpenGL is through the use of the stencil buffer. Values in the stencil buffer may be modified as a side effect of drawing into a color buffer, or modified directly through the *DrawPixels* function. If the stencil test is enabled (by calling *Enable*

with the constant `STENCIL_TEST`), then operations such as drawing and copying pixels will conduct the test specified by the currently defined stencil function (which consists of a comparison function, a reference value, and a mask). Depending on the outcome of the stencil test, values in the stencil buffer are modified according to the currently defined stencil operation (which is capable of clearing, incrementing, decrementing, replacing, inverting, or keeping the current stencil value).

Some very complex region of interest operations can be defined with this mechanism. For instance, it is possible to use OpenGL geometry rendering routines to draw a complex shape in such a way that the only effect is to place a value of 1 in the stencil buffer wherever drawing has occurred. It is then possible to perform an operation (such as redrawing the image with a different set of attributes) and overwrite only those pixels where there is a value of 1 in the stencil buffer.

Blending

Once the incoming fragment has passed all other tests, its red, green, blue, and alpha values may be combined with those in the frame buffer at the target location. This blending operation depends on the alpha value of the incoming fragment, the alpha value in the frame buffer, a source blending function and a destination blending function. The source blending function is applied to the incoming fragment, and the destination blending function is applied to the pixel already in the frame buffer. The two results are added to compute the new value to be written into the frame buffer. Blending is enabled by calling *Enable* with the constant `BLEND`.

The blending operation combines the incoming fragment with the pixel value in the frame buffer according to the following equation:

$$C_s S + C_d D$$

where C_s is the pixel value of the incoming fragment, C_d is the pixel value stored in the frame buffer, S is the source blending function and D is the destination blending function. The source blending function can be one of `ZERO`, `ONE`, `DST_COLOR`, `ONE_MINUS_DST_COLOR`, `SRC_ALPHA`, `ONE_MINUS_SRC_ALPHA`, `DST_ALPHA`, `ONE_MINUS_DST_ALPHA`, or `SRC_ALPHA_SATURATE`. The destination blending function can be one of `ZERO`, `ONE`, `SRC_COLOR`, `ONE_MINUS_SRC_COLOR`, `SRC_ALPHA`, `ONE_MINUS_SRC_ALPHA`, `DST_ALPHA`, or `ONE_MINUS_DST_ALPHA`.

Blending can be used as a transition effect between two images; it can be used to make an image “semitransparent” with respect to a background image; and it can be used to achieve a wide range of image compositing effects.

Dithering

Frame buffers with limited color resolution (*e.g.*, 8 bits) may allow applications to trade spatial resolution for color resolution by dithering. Dithering is enabled by calling *Enable* with the constant *DITHER*. The precise mechanics of dithering are not specified by OpenGL, but it is specified that a value produced by dithering must depend only on the incoming fragment value and its x and y window coordinates. When dithering is enabled, applications must realize that pixel values written to the frame buffer and read back may differ from those in the original image. This may make it unsuitable for use in some imaging applications.

Logical Operations

The next fragment operation performs a logical operation between the incoming fragment value and the value currently stored in the frame buffer. Logical operations are enabled by calling *Enable* with the constant *COLOR_LOGIC_OP*. The current logical operation is established by calling *LogicOp()* with a constant specifying the desired logical operation (the usual list of 16 is supported).

Masking

OpenGL maintains a separate flag for each of the red, green, blue, and alpha channels to denote whether the channel is writable. The flags that enable writing R, G, B, and A values are set by calling *ColorMask*. There is no bit-level control over writing color values; each channel is either writable or not.

Frame Buffer Control

The current drawable in OpenGL always has some number of bitplanes that comprise the color buffer for that drawable. The color buffer may consist of 1, 2, or 4 buffers depending on whether it is single buffered, double buffered, stereo, or stereo double buffered. The four possible buffers are named front left, front right, back left, and back right. Single buffered color buffers contain only front buffers. Double buffered color buffers contain both front and back buffers. Monoscopic color buffers contain only left buffers. Stereoscopic color buffers contain both left and right buffers.

The current draw buffer is the buffer that is the target of all subsequent pixel write operations. It can be set by calling *DrawBuffer* with a value of *NONE*, *FRONT_LEFT*, *FRONT_RIGHT*, *BACK_LEFT*, *BACK_RIGHT*, *FRONT*, *BACK*, *LEFT*, *RIGHT*, or *FRONT_AND_BACK*. Names that omit reference to *LEFT* or *RIGHT* refer to both left and right buffers. If more than one buffer is selected for drawing, blending and logical operations are computed and applied independently for each selected buffer.

Color buffers that contain both a front and a back buffer can be used to eliminate annoying rendering artifacts caused by clearing and redrawing an image in a visible buffer. The front buffer is displayed while rendering is occurring in the back buffer. When rendering is completed, the *SwapBuffers* routine can be called in order to promote the contents of the back buffer to the front buffer. The contents of the back buffer is undefined after this call.

OTHER OPENGL CAPABILITIES FOR IMAGING

Copying Pixels

The capabilities described so far have dealt entirely with transferring pixels from host memory to the frame buffer. Pixels can be transferred from one location in the frame buffer to another with the *CopyPixels* routine. As shown in Figure 6, the operations that occur during a copy pixels operation are nearly identical to those that occur during a draw pixels operation. The biggest differences are that the source of pixels is the frame buffer rather than host memory, and that the pixel unpacking operation shown in Figure 1 is replaced by a similar operation that extracts pixel values from frame buffer memory. The remainder of the pipeline is the same, including pixel transfer, pixel rasterization, and fragment operations.

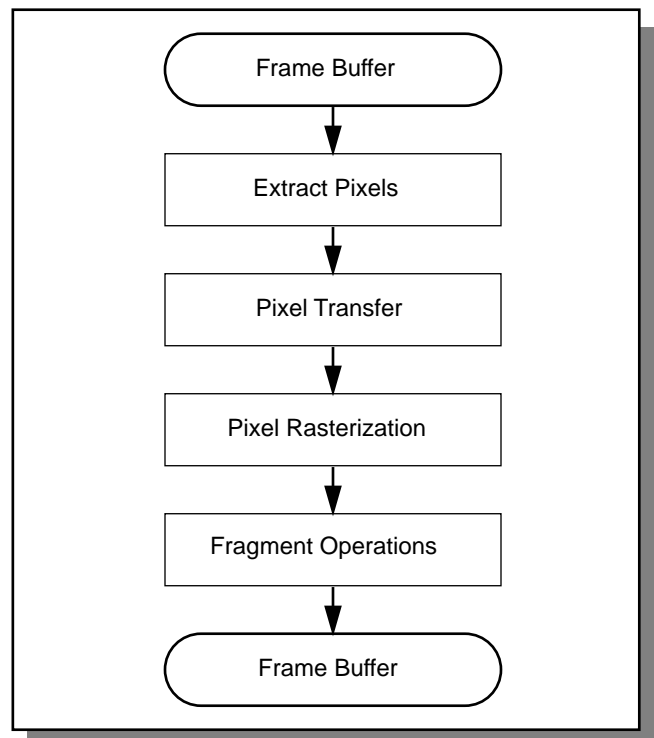


Figure 6: Block diagram of the operations in *CopyPixels*

Reading Pixels

Figure 7 shows the sequence of operations that occur when *ReadPixels* is called. Pixel values are extracted from frame buffer memory using the same operation as is used in a copy pixels operation. Pixels thus extracted are subjected to the pixel transfer operations as described previously. The resulting pixels are packed into host memory in an operation that is the analog of the pixel unpacking operation described earlier. When reading or copying pixels, values obtained for pixels that lie outside the current drawable are undefined.

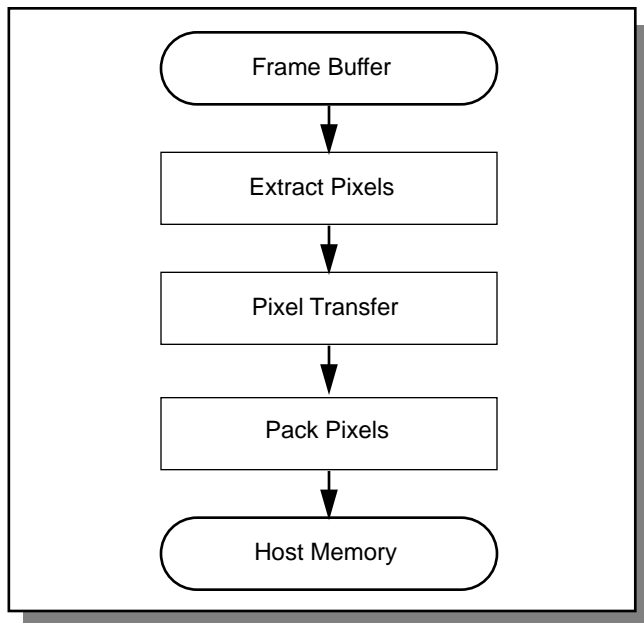


Figure 7: Block diagram of the operations in *ReadPixels*

Color Index Rendering

This paper has been limited to discussing the semantics of RGBA rendering in OpenGL. A second rendering mode, color index mode, is also available, but a full discussion of this mode is outside the scope of this paper. However, the *PixelMap* routine allows you to define a mapping from color index values to RGBA values that will occur as part of the pixel mapping operation. This mechanism can be used to add a variety of false color display capabilities to single channel data.

Texture Mapping

One of the apparent deficiencies in the OpenGL pixel processing pipeline is the lack of any capability for high quality scaling and/or rotation of images. However, OpenGL supports image scaling and rotation through its 2D texture mapping capabilities. Applications can control the resampling of a texture image using the *TexParameter* call to select between NEAREST, LINEAR, and several mipmapping filters. An

image can be loaded into texture memory and then textured onto a polygon. A variety of image transformations, including warping, are possible with this method. The pixel transfer operations described earlier are also applied to pixels as they are stored in texture memory. An extension which adds image transformation capabilities directly to the pixel processing pipeline has been proposed as an alternative to this approach and is described in the next section.

PIXEL TRANSFER EXTENSIONS

This section describes some of the most relevant imaging extensions to OpenGL. Some of the extensions described are supported by more than one vendor, others by a single vendor. They are all described in the context of the OpenGL pixel processing pipeline as defined thus far. As of this writing, no one who supports OpenGL has a complete implementation of the pipeline as described in this section.

Color Table

The EXT_color_table extension defines the entry points needed to create and modify color lookup tables similar to the pixel mapping table defined by OpenGL. The type of tables that are created through this extension are somewhat more capable than the ones created through the *PixelMap* call. Applications are given more control over the exact internal format of these lookup tables, and a new query mechanism is provided.

This extension defines a new lookup table which occurs immediately after the standard OpenGL pixel mapping table (see Figure 4). Because of the benefits of using the new color lookup tables, people will be encouraged to use the more capable color table and discouraged from using the existing OpenGL pixel mapping table. Two additional color lookup tables are defined: one that occurs immediately after the convolution operation and one that occurs immediately after the color matrix operation. The image transform extension defines yet another color lookup table that can be used for the purpose of window level mapping.

Convolution

Convolution is a common image processing operation that can be used to filter an image. The filtering is accomplished by computing the sum of products between the source image and a smaller image called the convolution filter or convolution kernel. The convolution filter can be loaded with different values to achieve effects like sharpening, blurring, and edge detection.

The EXT_convolution extension adds a convolution operator immediately after the new color lookup table defined by the color table extension. It provides facilities for creating and manipulating convolution filters, modifying convolution filter

parameters, changing parameters of the convolution operation (such as a mode that defines how borders are to be treated), support for 1D and 2D general convolution operations, and support for separable 2D convolution operations. Post-convolution scale and bias factors are provided to make it easier to map the resulting values into a different range. A post-convolution color table is also provided for additional flexibility in remapping the results of the convolution operation.

This extension defines only one way to treat the borders of an image during convolution. The HP_convolution_border_modes extension defines some additional border modes that are commonly used.

Image Transformation

The HP_image_transform extension adds support for scaling, rotation, and translation of images to the pixel processing pipeline. The 2D transformation attributes are specified as individual values in order to make it easier for implementations to detect and optimize important special cases. This extension also provides control over the resampling process that occurs after the transformation. If the image is magnified, the resampling method can be one of NEAREST, LINEAR, or CUBIC_HP. If the image is minified, AVERAGE_HP is a fourth method that may be chosen. Finally, this extension defines a color table that operates immediately after the image transformation operation. Except for its position in the pixel processing pipeline, this color table is identical to the other color lookup tables that are defined on top of the EXT_color_table extension.

Color Matrix

A method for reassigning and duplicating color components and performing simple color space conversions is defined by the SGI_color_matrix extension. This extension adds a 4×4 matrix stack to the pixel processing path. The matrix transforms each RGBA pixel group by the matrix on the top of the color matrix stack. After the matrix operation is performed, a linear transform is applied using scale and bias values defined for each of the red, green, blue, and alpha channels. The resulting pixel values are then passed through a color lookup table of the type defined by the EXT_color_table extension.

Histogram

Statistics about the number of occurrences of each pixel value in an image are useful for a variety of image analysis and display techniques. The EXT_histogram extension defines an operation that counts the occurrences of specific pixel component values and tracks the minimum and maximum component values. An optional mode allows the pixels to be discarded after the histogram and min/max operations have been performed. This extension provides entry points for both querying and resetting the histogram table and the min/max values.

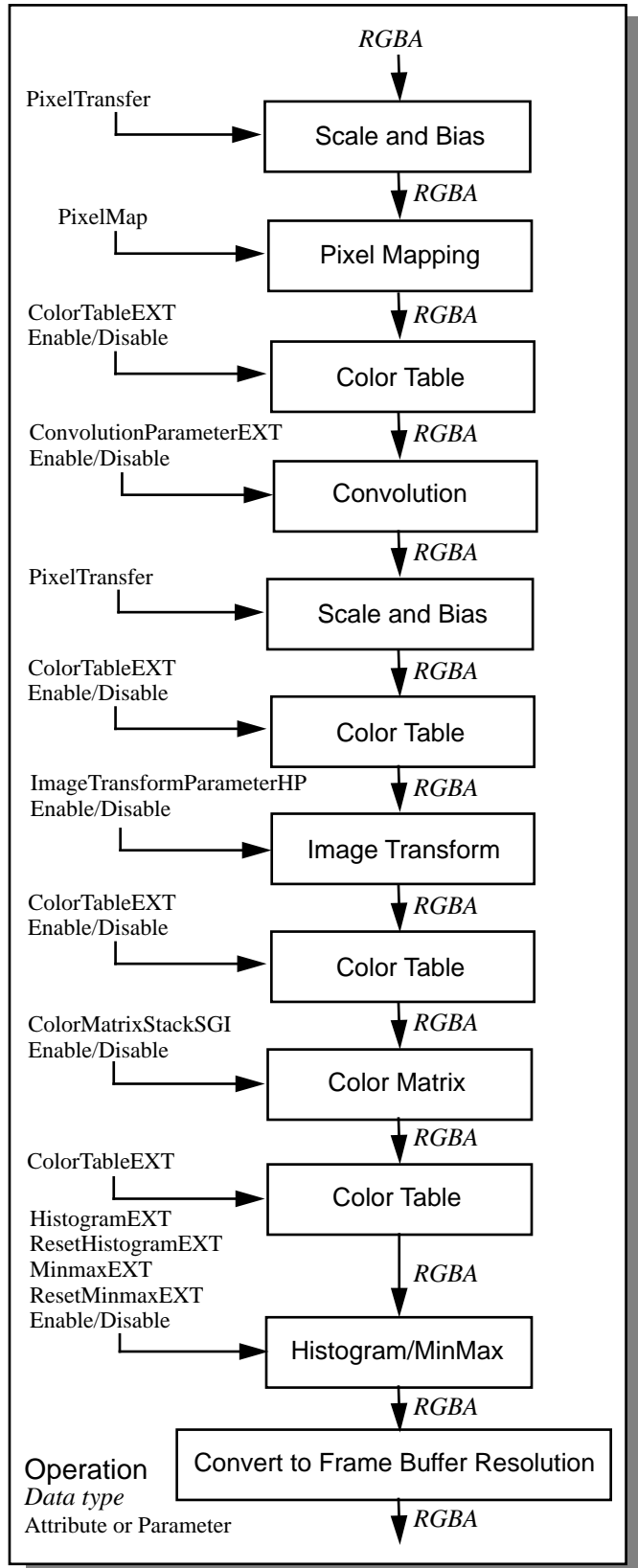


Figure 8: Details of the extended pixel transfer operation

Each of the extensions discussed so far in this section adds capabilities to the pixel transfer operation shown in detail in Figure 4. If all of these extensions were available on a single implementation, the details of the pixel transfer operation would be as shown in Figure 8.

OTHER IMAGING EXTENSIONS

This section describes imaging extensions to areas other than the pixel transfer operation.

Packed Pixels

Although the arguments to *DrawPixels* provide applications with a lot of flexibility as far as specifying how pixels are stored in host memory, the EXT_packed_pixels extension goes even further. This extension provides support for pixel formats that are packed into a native machine type (unsigned byte, unsigned short, unsigned int). Among the pixel types added by this extension are RGB-3-3-2, RGBA-4-4-4-4, RGBA-5-5-5-1, RGBA-8-8-8-8, and RGBA-10-10-10-2. These additional pixel types provide more flexibility to applications that need to make difficult storage vs. color resolution tradeoffs.

Blending

An area that has received a lot of attention in terms of extension development is blending. As described previously, blending is a fragment operation that occurs just prior to writing a pixel value in the frame buffer. While extremely useful, the blending extensions that have been developed may gain acceptance slowly, since hardware support is almost essential for obtaining reasonable performance.

The EXT_blend_color extension augments OpenGL's blending capabilities by defining a constant color that can be used in the blending equation. The new source and destination blending factors that are added are CONSTANT_COLOR_EXT, ONE_MINUS_CONSTANT_COLOR_EXT, CONSTANT_ALPHA_EXT, and ONE_MINUS_CONSTANT_ALPHA_EXT. One common use of this capability is blending two RGB images without the need for an alpha channel for either image.

A second blend extension, EXT_blend_minmax, adds a way to select a different blending equation. The *BlendEquation-EXT* routine can be used to specify the blending equation as FUNC_ADD_EXT (the default blending equation as described earlier), MIN_EXT, or MAX_EXT. When MIN_EXT is selected, the value written into the frame buffer will be the lesser of the incoming fragment's pixel value and the value already in the frame buffer. When MAX_EXT is selected, the value written will be the greater of the two. This capability is essential for implementing both minimum and maximum intensity projections.

A third blend extension is called EXT_blend_subtract. Two new blending equations are added to support image differencing operations. The FUNC_SUBTRACT_EXT equation is:

$$C_s S - C_d D$$

and the FUNC_REVERSE_SUBTRACT_EXT equation is:

$$C_d D - C_s S$$

where C_s , C_d , D , and S are as defined previously.

The OpenGL Imaging State Block Diagram

The complete state block diagram of the OpenGL imaging pipeline, including the extensions that have been discussed, is shown in Figure 9. This diagram illustrates how image data is moved around the system as well as the operations that affect pixels as they are being moved. The arrows represent the flow of data within the system. Boxes outlined with thin lines and square corners are operations that are applied to pixel values. Boxes with bold outlines and rounded corners represent storage locations for pixel values. The *CopyPixels* operation is not shown, but the operations that occur as a result of a *DrawPixels* or a *ReadPixels* operation are represented.

Other Extensions

Two extensions have been proposed to address other limitations of OpenGL with respect to imaging. Both of these extensions are still considered experimental as vendors try to determine the best way to expose the desired capabilities.

The SGIX_FBConfig extension introduces a new way to describe the capabilities of a GLX drawable (e.g., depth of color components and type and size of all ancillary buffers). Currently, GLX describes drawables by adding information to the list of visuals as reported by X. This method has some inherent limitations that are overcome by the more flexible scheme proposed by the SGIX_FBConfig extension. This extension also removes certain restrictions regarding similarity between rendering contexts and drawables, and it allows X visual types other than DirectColor and TrueColor to support the RGBA rendering semantics.

The SGIX_pbuffer extension defines an additional type of non-visible rendering buffer that supports hardware accelerated rendering and provides a useful intermediate storage area for multipass imaging operations. GLXPbuffers are equivalent to GLXpixmap except that (a) there is no associated X pixmap, (b) the format of a GLXPbuffer can only be described with a GLXFBConfig, and (c) GLXPbuffers must work with both direct and indirect rendering contexts. There are two types of GLXPbuffers: volatile and secure. Volatile GLXPbuffers may be created in a volatile area of the frame buffer (such as in the usually unused bits of deep pixels), and their contents may be arbitrarily and asynchronously lost at any

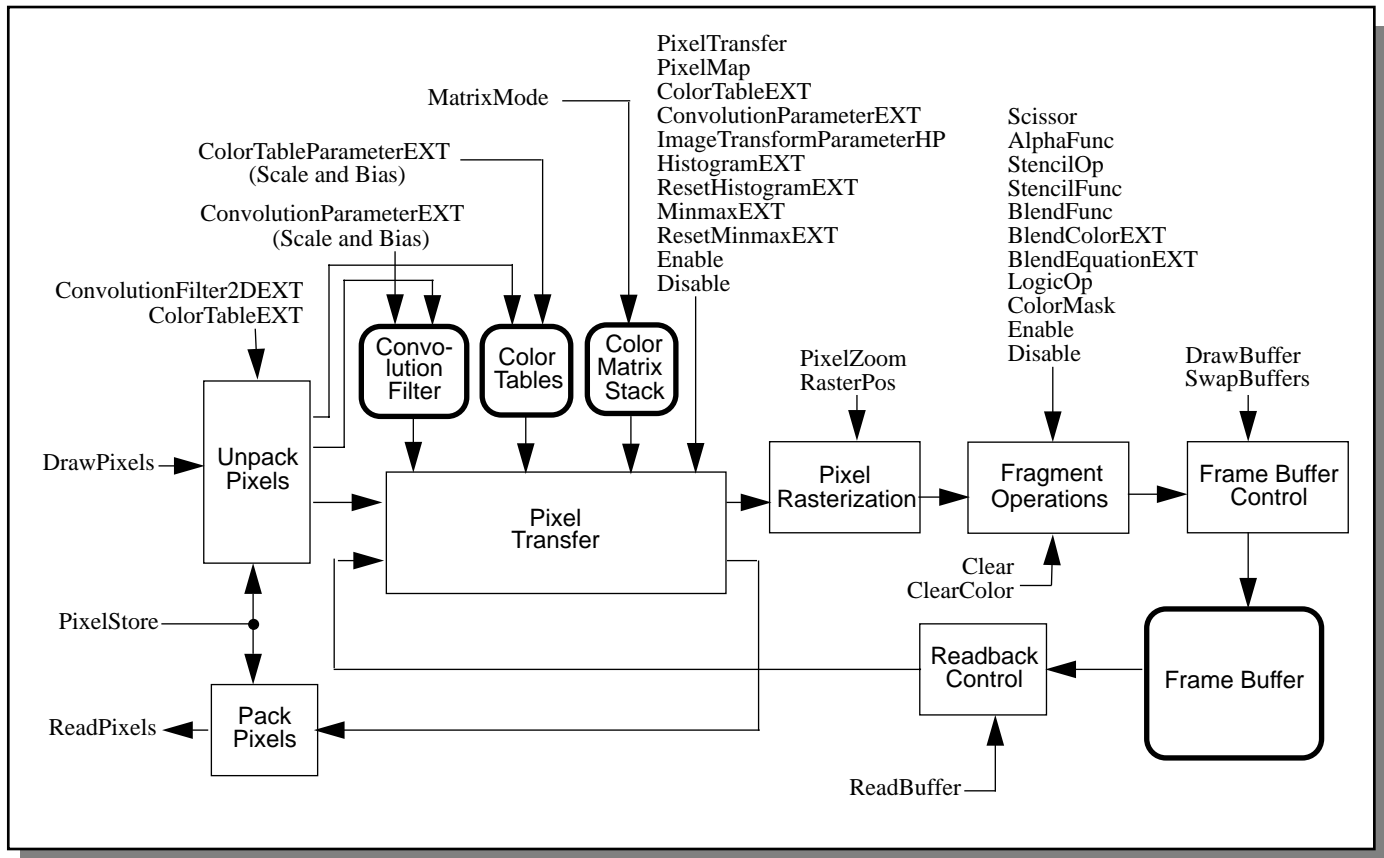


Figure 9: State block diagram of OpenGL1.1 imaging pipeline, including imaging extensions

time. Secure GLXPbuffers may also be created in a volatile region of the frame buffer, but they must be able to save and restore their contents if resource contention occurs. Secure GLXPbuffers are provided for ease-of-use, but volatile GLXP-buffers may provide better performance.

TOOLKITS AND MIDDLEWARE

OpenGL is a low-level interface, in that it is intended to provide access to the frame buffer with the highest possible performance and the lowest possible overhead. At the OpenGL API level there is little to get in the way of applications having direct control over the behavior and the performance of the underlying frame buffer hardware. Like X, OpenGL follows the philosophy of providing mechanism, not policy.

These design choices make OpenGL a powerful and flexible API, but one that may be tedious to program. Because different implementation choices lead to different system performance on each OpenGL platform, it can also be difficult to develop software that runs efficiently on a range of platforms.

For these reasons, OpenGL is a good foundation on which to build toolkits or "middleware" for imaging applications. Such toolkits are already available for OpenGL environments from some vendors and from third party software developers. Applications that want the conveniences offered by a toolkit can use one of these higher-level libraries. Applications that need the ultimate in performance or flexibility can use the OpenGL API directly.

CONCLUSION

Because of OpenGL's current capabilities and broad industry support, vendors developing imaging applications need to carefully consider whether to use it as their imaging API. Extensions to support advanced imaging and volume rendering features are currently being defined. Toolkits and middleware libraries based on OpenGL are available from hardware vendors and third party software vendors. Especially with the addition of some of the imaging extensions that have been defined, OpenGL provides a powerful foundation on which to build high-performance imaging applications.

ACKNOWLEDGEMENTS

This paper was produced with the support of Hewlett-Packard through the Graphics Software Lab in Fort Collins, CO. OpenGL is a registered trademark of Silicon Graphics, Inc. The X Window System is a trademark of Massachusetts Institute of Technology. Microsoft Windows is a trademark of Microsoft Corporation. OS/2 is a trademark of International Business Machines.

The author would like to extend his sincerest thanks to Kurt Akeley of Silicon Graphics, Inc. for patiently answering numerous questions about OpenGL's pixel processing pipeline and the imaging extensions developed by SGI. Reviewers who made valuable comments include Barthold Lichtenbelt, Dave Desormeaux, Antonio Palacios, and Mike McCarthy of Hewlett-Packard; and Kurt Akeley, Paula Womack, Nancy Cam, Ziv Gigus, and Lesley Kalmin of Silicon Graphics, Inc.

REFERENCES

- [1] Akeley, Kurt, and Mark Segal. *The OpenGL[®] Graphics System: A Specification (Version 1.1)*. December 2, 1995, Silicon Graphics, Inc., Mountain View, CA.
- [2] *EXT_blend_color Specification*, September 16, 1994, Silicon Graphics, Inc.
- [3] *EXT_blend_minmax Specification*, September 16, 1994, Silicon Graphics, Inc.
- [4] *EXT_blend_subtract Specification*, September 16, 1994, Silicon Graphics, Inc.
- [5] *EXT_convolution Specification*, June 13, 1995, Silicon Graphics, Inc.
- [6] *EXT_histogram Specification*, March 9, 1995, Silicon Graphics, Inc.
- [7] *EXT_packed_pixels Specification*, September 16, 1994, Silicon Graphics, Inc.
- [8] *HP_image_transform Specification*, June 1, 1995, Hewlett-Packard Company.
- [9] *HP_convolution_border_modes Specification*, April 25, 1995, Hewlett-Packard Company.
- [10] Karlton, Phil. *OpenGL[™] Graphics with the X Window System[®] (Version 1.1)*. January 1, 1995, Silicon Graphics, Inc., Mountain View, CA.
- [11] Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL[™] Programming Guide*, Addison-Wesley, 1993.
- [12] OpenGL Architecture Review Board. *OpenGL[™] Reference Manual*, Addison-Wesley, 1993.
- [13] *The OpenGL Graphics Interface*, Randi Rost, et. al. SIGGRAPH Tutorial 30 Course Notes, SIGGRAPH '94, July 24-29, 1994, Orlando, FL.
- [14] Scheifler, Bob and Jim Gettys. *X Window System, 3rd Edition*. Digital Press, 1992.
- [15] *SGI_color_matrix Specification*, September 16, 1994, Silicon Graphics, Inc.
- [16] *SGI_color_table Specification*, May 3, 1995, Silicon Graphics, Inc.
- [17] *SGIX_FBConfig Specification*, May 10, 1995, Silicon Graphics, Inc.
- [18] *SGIX_pbuffer Specification*, May 10, 1995, Silicon Graphics, Inc.