Chapter 9

Extensions to GLX

GLX is an extension to the X Window System that makes OpenGL available in an X Window System environment. All GLX functions and other elements have the prefix **glX** (just as all OpenGL elements have the prefix **gl**) For more information about GLX, see “The GLX Extension to the X Window System” on page 2 in this document, and Appendix D, “The OpenGL Extension to the X Window System,” in the *OpenGL Programming Guide*.

This chapter explains how to use extensions to GLX. You learn about:

- “The Make Current Read Extension” on page 164
- “The Visual Info Extension” on page 165
- “The Visual Rating Extension” on page 167
- “The Video Synchronization Extension” on page 167
- “The Swap Control Extension” on page 168
- “The Import Context Extension” on page 169

The following sections describe extensions that are experimental:

- “The Framebuffer Configuration Extension” on page 171
- “The Pixel Buffer Extension” on page 179
- “The Video Source Extension” on page 183

Note: Using OpenGL in an X Window System environment is discussed in the following chapters of this guide:

- Chapter 2, “OpenGL and X: Getting Started”
- Chapter 3, “OpenGL and X: Examples”
- Chapter 4, “OpenGL and X: Advanced Topics”

The Make Current Read Extension

The make current read extension, SGI_make_current_read, allows you to attach separate read and write drawables to a GLX context by calling **glXMakeCurrentReadSGI()**.

In GLX 1.1, you associate a GLX context with one drawable (window or pixmap) calling **glXMakeCurrent()**. **glXMakeCurrentReadSGI()** lets you attach a GLX context to two drawables: The first is the one you draw to, the second serves as a source for pixel data.

Having both a read and write drawable is useful under various circumstances; for example, to copy the contents of a window to another window, to stream video to a window, and so on.

The write drawable is used for all OpenGL operations. Accumulation buffer operations fetch data from the write drawable and are not allowed when the read and write drawable are not identical.

The read drawable is used for any color, depth, or stencil values that are retrieved by **glReadPixels()**, **glCopyPixels()**, or any OpenGL extension that sources depth images from the framebuffer in the manner of **glReadPixels()** and **glCopyPixels()**.

Here's some additional information about the two drawables:

- The two drawables don't need to have the same ancillary buffers.
- The read drawable doesn't have to contain a buffer corresponding to the current GL_READ_BUFFER of a GLX context. For example, the current GL_READ_BUFFER may be GL_BACK, and the read drawable may be single-buffered.

  If a subsequent command sets the read buffer to a color buffer that doesn't exist on the read drawable—even if set implicitly by **glPopAttrib()**—or if an attempt is made to source pixel values from an unsupported ancillary buffer, a GL_INVALID_OPERATION error is generated.
- If the current GL_READ_BUFFER does not exist in the read drawable, pixel values extracted from that drawable are undefined, but no error is generated.
- Operations that query the value of GL_READ_BUFFER use the value set last in the context, regardless of whether the read drawable has the corresponding buffer.

Possible Match Errors

When **glXMakeCurrentReadSGI()** associates two GLX drawables with a single GLX context, a BadMatch X protocol error is generated if either drawable was not created with the same X screen and visual as the context.

The color, depth, stencil, and accumulation buffers of the two drawables don't need to match. Certain implementations may impose additional constraints. For example, the current RealityEngine implementation requires that the color component resolution of both drawables is the same.

New Functions

glXMakeCurrentReadSGI(), glXGetCurrentReadDrawableSGI().

The Visual Info Extension

The visual info extension, EXT_visual_info, enhances the standard GLX visual mechanism as follows:

- You can request that a particular X visual type be associated with a GLX visual.
- You can query the X visual type underlying a GLX visual.
- You can request a visual with a transparent pixel.
- You can query whether a visual supports a transparent pixel value and query the value of the transparent pixel.

  Note that the notions of level and transparent pixels are orthogonal as both level 1 and level 0 visuals may or may not support transparent pixels.

Using the Visual Info Extension

To find a visual that best matches specified attributes, call **glXChooseVisual()**:

`XVisualInfo* glXChooseVisual( Display *dpy, int screen, int *attrib_list )`

- If GLX_RGBA is in *attrib_list*, you can choose a TrueColor or DirectColor visual by setting GLX_X_VISUAL_TYPE_EXT to GLX_TRUE_COLOR_EXT or GLX_DIRECT_COLOR_EXT, respectively.

  - If you set GLX_X_VISUAL_TYPE_EXT to PseudoColor, StaticColor, GrayScale, or StaticGray, visual selection will fail.
  - If GLX_X_VISUAL_TYPE_EXT is not in *attrib_list*, and if all other attributes are equivalent, a TrueColor visual is chosen in preference to a DirectColor visual.
- If GLX_RGBA is not in *attrib_list*, you can choose a PseudoColor or StaticColor visual by setting GLX_X_VISUAL_TYPE_EXT to GLX_PSEUDO_COLOR or GLX_STATIC_COLOR, respectively.
  - If you set GLX_X_VISUAL_TYPE_EXT to TrueColor, DirectColor, GrayScale, or StaticGray, visual selection will fail.
  - If GLX_X_VISUAL_TYPE_EXT is not in *attrib_list*, and if all other attributes are equivalent, a PseudoColor visual is chosen in preference to a StaticColor visual.
- If an undefined GLX attribute, or an unacceptable enumerated attribute value is encountered, NULL is returned.

More attributes may be specified in the attribute list. If a visual attribute is not specified, a default value is used. See the **glXChooseVisual()** reference page for more detail.

To free the data returned, use **XFree()**.

Using Transparent Pixels

If you want a visual with transparent pixels, and GLX_RGBA is in *attrib_list*, call **glXChooseVisual()** and specify the GLX_TRANSPARENT_TYPE_EXT value GLX_TRANSPARENT_RGB_EXT.

If you want a transparent visual and GLX_RGBA is not in *attrib_list*, call **glXChooseVisual()** and specify the GLX_TRANSPARENT_TYPE_EXT value GLX_TRANSPARENT_INDEX_EXT.

Don't specify one of the following values in *attrib_list* because typically only one transparent color or index value is supported:

GLX_TRANSPARENT_INDEX_VALUE_EXT,
GLX_TRANSPARENT_{RED|GREEN|BLUE|ALPHA}_VALUE_EXT

Once you have a transparent visual, you can query the transparent color value by calling **glXGetConfig()**. To get the transparent index value for visuals that support index rendering, use GLX_TRANSPARENT_INDEX_VALUE_EXT. For visuals that support RGBA rendering, use GLX_TRANSPARENT_{RED | GREEN | BLUE}_VALUE_EXT. The visual attribute GLX_TRANSPARENT_ALPHA_VALUE_EXT is included in the extension for future use.

"How to Create Overlays" on page 60 presents an example program that uses a transparent visual for the overlay window.

The Visual Rating Extension

The visual rating extension, EXT_visual_rating, allows servers to export visuals with improved features or image quality, but lower performance or greater system burden, without having to have these visuals selected preferentially. It is intended to ensure that most—but possibly not all—applications get the "right" visual.

You can use this extension during visual selection, keeping in mind that while you will get a good match for most systems, you may not get the best match for all systems.

Using the Visual Rating Extension

To determine the rating for a visual, call **glXGetConfig()** with *attribute* set to GLX_VISUAL_CAVEAT_EXT. **glXGetConfig()** returns the rating of the visual in the parameter *value*: GLX_NONE_EXT or GLX_SLOW_EXT.

If the GLX_VISUAL_CAVEAT_EXT attribute is not specified in the *attrib_list* parameter of **glXChooseVisual()**, preference is given to visuals with no caveats (that is, visuals with the attribute set to GLX_NONE_EXT). If the GLX_VISUAL_CAVEAT_EXT attribute is specified, then **glXChooseVisual()** matches the specified value exactly. For example, if the value is specified as GLX_NONE_EXT, only visuals with no caveats are considered.

The Video Synchronization Extension

The video synchronization extension, SGI_video_sync, allows an application to synchronize drawing with the vertical retrace of a monitor (in the case of an interlaced monitor, the synchronization is actually with the field rate instead). Using the video synchronization extension, an application can put itself to sleep until a counter corresponding to the number of screen refreshes reaches a desired value. The application can also query the current value of the counter.

The system maintains a video sync counter (an unsigned 32-bit integer) for each screen in a system. The counter is incremented upon each vertical retrace.

The counter runs as long as the graphics subsystem is running; it is initialized by the */usr/gfx/gfxinit* command.

Note: A process can query or sleep on the counter only when a direct context is current; otherwise, an error code is returned. See the reference page for more information.

New Functions

glXGetVideoSyncSGI(), glXWaitVideoSyncSGI().

The Swap Control Extension

The swap control extension, SGI_swap_control, allows an application to set a minimum period for buffer swaps (counted in display retrace periods). This is similar to the **swapinterval()** command in IRIS GL.

To set the buffer swap interval, call **glXSwapIntervalSGI()**, specifying the minimum number of retraces between buffer swaps in the *interval* parameter. For example, a value of 2 means that the color buffer is swapped at most every other display retrace.

glXSwapIntervalSGI() affects only buffer swaps for the GLX write drawable for the current context. Note that **glXSwapBuffers()** may be called with a *drawable* parameter that is not the current GLX drawable; in this case **glXSwapIntervalSGI(),** has no effect on that buffer swap.

New Functions

glXSwapIntervalSGI().

The Import Context Extension

The import context extension, EXT_import_context, allows multiple X clients to share an indirect rendering context. The extension also adds some query routines to retrieve information associated with the current context.

To understand this extension, you must first understand direct and indirect rendering. See "Direct and Indirect Rendering" on page 79 for some background information.

Importing a Context

You can use the extension to import another process OpenGL context as follows:

- To retrieve the XID for a GLX context, call **glXGetGLXContextIDEXT()**:

  ```
  GLXContextID glXGetGLXContextIDEXT(const GLXContext ctx)
  ```

  This function is client-side only. No round trip is forced to the server; unlike most X calls that return a value, **glXGetGLXContextIDEXT()** does not flush any pending events.

- To create a GLX context, given the XID of an existing GLX context, call **glXImportGLXContextEXT()**. You can use this function in place of **glXCreateContext()**, to share another process' indirect rendering context:

  ```
  GLXContext glXImportGLXContextEXT( Display *dpy,
                                     GLXContextID contextID)
  ```

  Only the server-side context information can be shared between X clients; client-side state, such as pixel storage modes, cannot be shared. Thus, **glXImportGLXContextEXT()** must allocate memory to store client-side information.

  This call does not create a new XID. It merely makes an existing XID available to the importing client. The XID goes away when the creating client drops its connection or the ID is explicitly deleted. The object goes away when the XID goes away and the context is not current to any thread.

- To free the client-side part of a GLX context that was created with **glXImportGLXContextEXT()**, call **glXFreeGLXContextEXT()**.

  ```
  void glXFreeGLXContextEXT(Display *dpy, GLXContext ctx)
  ```

  glXFreeGLXContextEXT() does not free the server-side context information or the XID associated with the server-side context.

Retrieving Display and Context Information

Use the extension to retrieve the display of the current context, or other information about the context, as follows:

- To retrieve the current display associated with the current context, call **glxGetCurrentDisplayEXT()**. If there is no current context, NULL is returned. No round trip is forced to the server; unlike most X calls that return a value, **glXGetCurrentDisplayEXT()** does not flush any pending events.
- To obtain the value of a context's attribute, call **glXQueryContextInfoEXT()**. The values and types corresponding to each GLX context attribute are listed in Table 9-1.

Table 9-1 Type and Context Information for GLX Context Attributes

GLX context attribute	Type	Context Information
GLX_SHARE_CONTEXT_EXT	XID	XID of the share list context
GLX_VISUAL_ID_EXT	XID	visual ID
GLX_SCREEN_EXT	int	screen number

New Functions

glXGetCurrentDisplayEXT(), glXGetGLXContextIDEXT(), glXImportGLXContextEXT(), glXFreeGLXContextEXT().

The Framebuffer Configuration Extension

The framebuffer configuration extension, SGIX_fbconfig, provides three new features:

- It introduces a new way to describe the capabilities of a GLX drawable, that is, to describe the resolution of color buffer components and the type and size of ancillary buffers by providing a GLXFBConfig construct.
- It relaxes the "similarity" requirement when associating a current context with a drawable.
- It supports RGBA rendering to one- and two-component windows and GLX pixmaps as well as pbuffers.

Caution: This extension is an SGIX (experimental) extension. The interface may change, or some other details of the extension may change.

Why Use the Framebuffer Configuration Extension?

Use this extension

- if you want to use pbuffers (see "The Pixel Buffer Extension" on page 179)
- if you want to do RGBA rendering to a PseudoColor or GrayScale window
- instead of **glXChooseVisual()**, since it provides visual selection for all GLX drawables, including pbuffers, and incorporates the visual info and visual rating extensions.

This section briefly explores the three new features provided by the extension.

Describing a Drawable With a GLXFBConfig Construct

Currently GLX overloads X visuals so they have additional buffers and other characteristics needed for OpenGL rendering. This extension packages GLX drawables by defining a new construct, a GLXFBConfig, that encapsulates GLX drawable capabilities and has the following properties:

- It may or may not have an associated X visual. If it does have an associated X visual, then it is possible to create windows that have the capabilities described by the GLXFBConfig.

- A particular GLXFBConfig does not need to work with all GLX drawables. For example, it's possible for implementations to export GLXFBConfigs that work only with GLX pixmaps.

Less-Rigid Similarity Requirements When Matching Context and Drawable

In OpenGL 1.0, when you associate a drawable with a GLX context by calling **glXMakeCurrent()**, the two have to be "similar"; that is, they must have been created with the same visual. This extension relaxes the requirement; it only requires the context and drawable to be compatible. This is less restrictive and implies the following:

- The *render_type* attribute for the context must be supported by the GLXFBConfig that the drawable was created with. For example, if the context was created with *render_type* GLX_RGBA_TYPE_SGIX, then the GLX_RENDER_TYPE_SGIX attribute of the GLXFBConfig must have the GLX_RGBA_BIT_SGIX bit set.
- All color buffers and ancillary buffers that exist in both GLXFBConfigs must have the same size. For example, a GLX drawable that has a front left buffer and a back left buffer with red, green, and blue sizes of 4 is not compatible with a GLXFBConfig that has only a front left buffer with red, green, and blue sizes of 8. However, it is compatible with a GLXFBConfig that has only a front left buffer if the red, green, and blue sizes are 4.

Note that when a context is created, it has an associated rendering type: GLX_RGBA_TYPE_SGIX or GLX_COLOR_INDEX_TYPE_SGIX.

Less-Rigid Match of GLX Visual and X Visual

The current GLX specification requires that the GLX_RGBA visual attribute be associated only with TrueColor and DirectColor X visuals. This extension makes it possible to do RGBA rendering to windows created with visuals of type PseudoColor, StaticColor, GrayScale, and StaticGray. In each case, the red component is used to generate the display, and the green and blue components, if present, are ignored for display purposes.

The OpenGL RGBA rendering semantics are more powerful than the OpenGL index rendering semantics. By extending the X visual types that can be associated with an RGBA color buffer, this extension allows RGBA rendering semantics to be used with pseudo-color and gray-scale displays. A particularly useful application of this extension is support of one- and two-component RGBA drawables (a two-component visual has zero-size green and blue components; a one-component visual has, in addition, a zero-size alpha component).

The GLXFBConfig Construct

A GLXFBConfig describes the format, type, and size of the color buffer and ancillary buffers for a GLX drawable. When the GLX drawable is a window, then the GLXFBConfig that describes it has an associated X visual; for a GLXPixmap or GLXPbuffer there may or may not be an X visual associated with the GLXFBConfig.

Choosing a GLXFBConfig Construct

Use **glXChooseFBConfigSGIX()** to get GLXFBConfig constructs that match a list of attributes or to get the list of GLXFBConfig constructs (GLXFBConfigs) that are available on the specified screen.

```
GLXFBConfig *glXChooseFBConfigSGIX(Display *dpy, int screen,
                          const int *attrib_list, int *nitems)
```

If *attrib_list* is NULL, **glXChooseFBConfigSGIX()** returns an array of GLXFBConfigs that are available on the specified screen; otherwise this call returns an array of GLXFBConfigs that match the specified attributes. Table 9-2 shows only attributes specified by this extension; additional attributes are listed on the **glXChooseVisual()** reference page.

Table 9-2 Attributes Introduced by GLXFBConfig

Attribute	Type	Description
GLX_DRAWABLE_TYPE_SGIX	bitmask	Mask indicating which GLX drawables are supported. Valid bits are GLX_WINDOW_BIT_SGIX and GLX_PIXMAP_BIT_SGIX.
GLX_RENDER_TYPE_SGIX	bitmask	Mask indicating which OpenGL rendering modes are supported. Valid bits are GLX_RGBA_BIT_SGIX and GLX_COLOR_INDEX_BIT_SGIX.
GLX_X_RENDERABLE_SGIX	Boolean	True if X can render to drawable.
GLX_FBCONFIG_ID_SGIX	XID	XID of GLXFBConfig.

The attributes are matched in an attribute-specific manner. Some attributes, such as GLX_LEVEL, must match the specified value exactly; others, such as GLX_RED_SIZE, must meet or exceed the specified minimum values. The sorting criteria are defined as follows:

smaller — GLXFBConfigs with an attribute value that meets or exceeds the specified value are matched. Precedence is given to smaller values (when a value is not explicitly requested, the default is implied).

larger — When the value is requested explicitly, only GLXFBConfigs with a corresponding attribute value that meets or exceeds the specified value are matched. Precedence is given to larger values. When the value is not requested, explicitly behaves exactly like the "smaller" criterion.

exact — Only GLXFBConfigs whose corresponding attribute value exactly matches the requested value are considered.

mask — For a config to be considered, all the bits that are set in the requested value must be set in the corresponding attribute. (Additional bits might be set in the attribute.)

Note also that "don't care" means that the default behavior is to have no preference when searching for a matching GLXFBConfig.

Table 9-3 illustrates how each attribute is matched.

Table 9-3 GLXFBConfig Attribute Defaults and Sorting Criteria

Attribute	Default	Sorting Criteria
GLX_BUFFER_SIZE	0	smaller
GLX_LEVEL	0	smaller
GLX_DOUBLEBUFFER	don't care	smaller
GLX_STEREO	False	exact
GLX_AUX_BUFFERS	0	smaller
GLX_RED_SIZE	0	larger
GLX_GREEN_SIZE	0	larger
GLX_BLUE_SIZE	0	larger
GLX_ALPHA_SIZE	0	larger
GLX_DEPTH_SIZE	0	larger
GLX_STENCIL_SIZE	0	larger
GLX_ACCUM_RED_SIZE	0	larger

Table 9-3 (continued) GLXFBConfig Attribute Defaults and Sorting Criteria

Attribute	Default	Sorting Criteria
GLX_ACCUM_GREEN_SIZE	0	larger
GLX_ACCUM_BLUE_SIZE	0	larger
GLX_ACCUM_ALPHA_SIZE	0	larger
GLX_SAMPLE_BUFFERS_SGIS	0 if GLX_SAMPLES_ SGIS = 0, 1 otherwise	smaller
GLX_SAMPLES_SGIS	0	smaller
GLX_X_VISUAL_TYPE_EXT	don't care	exact
GLX_TRANSPARENT_TYPE_EXT	GLX_NONE_EXT	exact
GLX_TRANSPARENT_INDEX_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_RED_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_GREEN_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_BLUE_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_ALPHA_VALUE_EXT	don't care	exact
GLX_VISUAL_CAVEAT_EXT	GLX_NONE_EXT	exact, if specified, otherwise minimum.
GLX_DRAWABLE_TYPE_SGIX	GLX_WINDOW_BIT_ SGIX	mask
GLX_RENDER_TYPE_SGIX	GLX_RGBA_BIT_SGIX	mask
GLX_X_RENDERABLE_SGIX	don't care	exact
GLX_FBCONFIG_ID_SGIX	don't care	exact

There are several uses for the **glXChooseFBConfigSGIX()** function:

- Retrieve all GLXFBConfigs on the screen (*attrib_list* is NULL).
- Retrieve a GLXFBConfig with a given ID specified with GLX_FBCONFIG_ID_SGIX.
- Retrieve the GLXFBConfig that's the best match for a given list of visual attributes.

- Retrieve first a list of GLXFBConfigs that match some criteria, for example, each GLXFBConfig available on the screen or all double-buffered visuals available on the screen. Then call **glXGetFBConfigAttribSGIX()** to find their attributes and choose the one that best fits your needs.

Once the GLXFBConfig is obtained, you can use it to create a GLX pixmap, window, or pbuffer (see "The Pixel Buffer Extension" on page 179). In the case of a window, you must first get the associated X visual by calling **glXGetVisualFromFBConfigSGIX().**

Below is a description of what happens when you call **glXChooseFBConfigSGIX()**:

- If no conforming GLXFBConfig exists, or if an error occurs (that is, an undefined GLX attribute is encountered in *attrib_list*, *screen* is invalid, or *dpy* does not support the GLX extension) then NULL is returned.
- If *attrib_list* is not NULL and more than one GLXFBConfig is found, then an ordered list is returned with the GLXFBConfigs that form the "best" match at the beginning of the list. ("How a GLXFBConfig Is Selected" on page 178 describes the selection process.) Use **XFree()** to free the memory returned by **glXChooseFBConfigSGIX()**.
- If GLX_RENDER_TYPE_SGIX is in *attrib_list*, the value that follows is a mask indicating which types of drawables will be created with it. For example, if GLX_RGBA_BIT_SGIX | GLX_COLOR_INDEX_BIT_SGIX is specified as the mask, then **glXChooseFBConfigSGIX()** searches for GLXFBConfigs that can be used to create drawables that work with both RGBA and color index rendering contexts. The default value for GLX_RENDER_TYPE_SGIX is GLX_RGBA_BIT_SGIX.

  The attribute GLX_DRAWABLE_TYPE_SGIX has as its value a mask indicating which drawables to consider. Use it to choose GLXFBConfigs that can be used to create and render to a particular GLXDrawable. For example, if GLX_WINDOW_BIT_SGIX | GLX_PIXMAP_BIT_SGIX is specified as the mask for GLX_DRAWABLE_TYPE_SGIX then **glXChooseFBConfigSGIX()** searches for GLXFBConfigs that support both windows and GLX pixmaps. The default value for GLX_DRAWABLE_TYPE_SGIX is GLX_WINDOW_BIT_SGIX.

If a GLXFBConfig supports windows it has an associated X visual. Use the GLX_X_VISUAL_TYPE_EXT attribute to request a particular type of X visual.

Note that RGBA rendering may be supported for any of the six visual types, but color index rendering can be supported only for PseudoColor, StaticColor, GrayScale, and StaticGray visuals (that is, single-channel visuals). The GLX_X_VISUAL_TYPE_EXT attribute is ignored if GLX_DRAWABLE_TYPE_SGIX is specified in *attrib_list* and the mask that follows does not have GLX_WINDOW_BIT_SGIX set.

GLX_X_RENDERABLE_SGIX is a Boolean indicating whether X can be used to render into a drawable created with the GLXFBConfig. This attribute is always True if the GLXFBConfig supports windows and/or GLX pixmaps.

Retrieving GLXFBConfig Attribute Values

To get the value of a GLX attribute for a GLXFBConfig, call

```
int glXGetFBConfigAttribSGIX(Display *dpy, GLXFBConfig config,
                             int attribute, int *value)
```

If **glXGetFBConfigAttribSGIX()** succeeds, it returns Success, and the value for the specified attribute is returned in *value*; otherwise it returns an error.

Note: A GLXFBConfig has an associated X visual if and only if the GLX_DRAWABLE_TYPE_SGIX value has the GLX_WINDOW_BIT_SGIX bit set.

To retrieve the associated visual, call:

```
XVisualInfo *glXGetVisualFromFBConfigSGIX(Display *dpy,
                                          GLXFBConfig config)
```

If *config* is a valid GLXFBConfig and it has an associated X visual, then information describing that visual is returned; otherwise NULL is returned. Use **XFree()** to free the returned data.

It is also possible to get a GLXFBConfig, given visual information:

```
GLXFBConfig glXGetFBConfigFromVisualSGIX(Display *dpy, XVisualInfo *vis)
```

If the visual is valid and supports OpenGL rendering (that is, if the GLX visual attribute GLX_USE_GL is True) then the associated GLXFBConfig is returned; otherwise NULL is returned.

To create a GLX rendering context or a GLX pixmap using a GLXFBConfig, call **glXCreateContextWithConfigSGIX()** or **glXCreateGLXPixmapWithConfigSGIX()**. The functions are similar to **glXCreateContext()** and **glXCreateGLXPixmap()**. See the reference page for detailed information.

How a GLXFBConfig Is Selected

If more than one GLXFBConfig matches the specification, they are prioritized as follows:

- Preference is given to GLXFBConfigs with the largest GLX_RED_SIZE, GLX_GREEN_SIZE, and GLX_BLUE_SIZE.
- If the requested GLX_ALPHA_SIZE is zero, preference is given to GLXFBConfigs that have GLX_ALPHA_SIZE set to zero; otherwise preference is given to GLXFBConfigs that have the largest GLX_ALPHA_SIZE value.
- If the requested number of GLX_AUX_BUFFERS is zero, preference is given to GLXFBConfigs that have GLX_AUX_BUFFERS set to zero; otherwise preference is given to GLXFBConfigs that have the smallest GLX_AUX_BUFFERS value.
- If the requested size of a particular ancillary buffer is zero (for example, GLX_DEPTH_BUFFER is zero), preference is given to GLXFBConfigs that also have that size set to zero; otherwise preference is given to GLXFBConfigs that have the largest size.
- If the requested value of either GLX_SAMPLE_BUFFERS_SGIS or GLX_SAMPLES_SGIS is zero, preference is given to GLXFBConfigs that also have these attributes set to zero; otherwise preference is given to GLXFBConfigs that have the smallest size.
- If GLX_X_VISUAL_TYPE_EXT is not specified but there is an X visual associated with the GLXFBConfig, the visual type is used to prioritize the GLXFBConfig.
- If GLX_RENDER_TYPE_SGIX has GLX_RGBA_BIT_SGIX set, the visual types are prioritized as follows: TrueColor, DirectColor, PseudoColor, StaticColor, GrayScale, and StaticGray.
- If only the GLX_COLOR_INDEX_SGIX is set in GLX_RENDER_TYPE_SGIX, visual types are prioritized as PseudoColor, StaticColor, GrayScale, and StaticGray.
- If GLX_VISUAL_CAVEAT_EXT is set, the implementation for the particular system on which you run determines which visuals are returned. See “The Visual Rating Extension” on page 167 for more information.

New Functions

glXGetFBConfigAttribSGIX(), glXChooseFBConfigSGIX(), glXCreateGLXPixmapWithConfigSGIX(), glXCreateContextWithConfigSGIX(), glXGetVisualFromFBConfigSGIX(), glXGetFBConfigFromVisualSGIX().

The Pixel Buffer Extension

You can use the pixel buffer extension, SGIX_pbuffer, to define a pixel buffer (GLXPbuffer or pbuffer for short).

Caution: This extension is an SGIX (experimental) extension. The interface or other aspects of the extension may change.

A GLXPbuffer is an additional non-visible rendering buffer for an OpenGL renderer. A pbuffer is equivalent to a GLXPixmap with the following exceptions:

- There is no associated X pixmap. Also, since pbuffers are a GLX resource, it may not be possible to render to them using X or an X extension other than GLX.
- The format of the color buffers and the type and size of associated ancillary buffers for a pbuffer can only be described with a GLXFBConfig; an X visual cannot be used.
- It is possible to create a pbuffer whose contents may be arbitrarily and asynchronously lost at any time.
- A pbuffer works with both direct and indirect rendering contexts.

A pbuffer is allocated in non-visible framebuffer memory, that is, areas for which hardware-accelerated rendering is possible. Applications include additional color buffers for rendering or image processing algorithms.

Pbuffers can be either “volatile,” that is, their contents can be destroyed by another window or pbuffer, or “preserved,” that is, their contents are guaranteed to be correct and are swapped out to virtual memory when other windows need to share the same framebuffer space. The contents of a preserved pbuffer are swapped back in when the pbuffer is needed. The swapping operation incurs a performance penalty, so preserved pbuffers should be used only if re-rendering the contents is not feasible.

A pbuffer is intended to be a “static” resource: a program typically allocates it only once, rather than as a part of its rendering loop. The framebuffer resources that are associated with a GLXPbuffer are also static. They are deallocated only when the GLXPbuffer is destroyed, or, in the case of volatile pbuffers, as the result of X server activity that changes framebuffer requirements of the server.

To create a GLXPbuffer, call **glXCreateGLXPbufferSGIX()**:

```
GLXPbuffer glXCreateGLXPbufferSGIX(Display *dpy, GLXFBConfig config,
                    unsigned int *width, unsigned int *height, int attrib_list)
```

This creates a single GLXPbuffer and returns its XID.

- *width* and *height* specify the pixel width and height of the rectangular GLXPbuffer.
- *attrib_list* specifies a list of attributes for the GLXPbuffer. (Note that the attribute list is defined in the same way as the list for **glXChooseFBConfigSGIX()**: attributes are immediately followed by the corresponding desired value and the list is terminated with None.)

  Currently only two attributes can be specified in *attrib_list*: GLX_CONTENTS_PRESERVED_SGIX and GLX_GET_LARGEST_PBUFFER_SGIX.

  – Use GLX_GET_LARGEST_PBUFFER_SGIX to get the largest available GLXPbuffer when the allocation of the pbuffer would otherwise fail. The width and height of the pbuffer (if one was allocated) are returned in *width* and *height*. Note that these values can never exceed the *width* and *height* that were initially specified. By default, GLX_GET_LARGEST_PBUFFER_SGIX is False.

  – If the GLX_CONTENTS_PRESERVED_SGIX attribute is set to False in *attrib_list*, a "volatile" GLXPbuffer is created and the contents of the pbuffer may be lost at any time. If this attribute is not specified, or if it is specified as True in *attrib_list*, the contents of the pbuffer are preserved, most likely by swapping out portions of the buffer to main memory when a resource conflict occurs. In either case, the client can register to receive a "buffer clobber" event and be notified when the pbuffer contents have been swapped out or have been damaged.

The resulting GLXPbuffer contains color buffers and ancillary buffers as specified by *config*. It is possible to create a pbuffer with back buffers and to swap the front and back buffers by calling **glXSwapBuffers()**. Note that a pbuffer uses framebuffer resources, so applications should deallocate it when not in use.

If **glXCreateGLXPbufferSGIX()** fails to create a GLXPbuffer due to insufficient resources, a BadAlloc X protocol error is generated and NULL is returned. If *config* is not a valid GLXFBConfig then a GLXBadFBConfigSGIX error is generated; if *config* does not support pbuffers, a BadMatch X protocol error is generated.

Rendering to a GLXPbuffer

Any GLX rendering context created with a GLXFBConfig or X visual that is compatible with a GLXFBConfig may be used to render into the pbuffer. For the definition of "compatible," see the description of **glXCreateContextWithConfigSGIX()**, **glXMakeCurrent()** and **glXMakeCurrentReadSGI()**.

If a GLXPbuffer is created with GLX_CONTENTS_PRESERVED_SGIX set to False, the storage for the buffer contents—or a portion of the buffer contents—may be lost at any time. It is not an error to render to a GLXPbuffer that is in this state, but the effect of rendering to it is undefined. It is also not an error to query the pixel contents of such a GLXPbuffer, but the values of the returned pixels are undefined.

Because the contents of a volatile GLXPbuffer can be lost at any time with only asynchronous notification (via the "buffer clobber" event), the only way a client can guarantee that valid pixels are read back with **glReadPixels()** is by grabbing the X server. (Note that this operation is potentially expensive and you should not do it frequently. Also, since this locks out other X clients, you should do it only for short periods of time.) Clients that don't wish to do this can check if the data returned by **glReadPixels()** is valid by calling **XSync()** and then checking the event queue for "buffer clobber" events (assuming that these events were pulled off of the queue before the **glReadPixels()** call).

To destroy a GLXPbuffer call **glXDestroyGLXPbufferSGIX()**:

```
void glXDestroyGLXPbufferSGIX(Display *dpy, GLXPbuffer pbuf)
```

To query an attribute associated with a GLXPbuffer, call **glXQueryGLXPbufferSGIX()**.

```
void glXQueryGLXPbufferSGIX(Display *dpy, GLXPbuffer pbuf, int attribute
                             unsigned int *value)
```

To get the GLXFBConfig for a GLXPbuffer, first retrieve the ID for the GLXFBConfig and then call **glXChooseFBConfigSGIX()**. See "The Framebuffer Configuration Extension" on page 171.

Directing the Buffer Clobber Event

An X client can ask to receive GLX events on a window or GLXPbuffer by calling **glXSelectEventSGIX()**:

```
void glXSelectEventSGIX(Display *dpy, GLXDrawable drawable,
                        unsigned long mask)
```

Currently you can only select the GLX_BUFFER_CLOBBER_BIT_SGIX GLX event as the *mask*. The event structure is

```
typdef struct {
  int event_type;            /* GLX_DAMAGED_SGIX or GLX_SAVED_SGIX */
  int draw_type;             /* GLX_WINDOW_SGIX or GLX_PBUFFER_SGIX */
  unsigned long serial;      /* # of last request processed by server */
  Bool send_event;           /* true if it came for SendEvent request */
  Display *display;          /* display the event was read from */
  GLXDrawable drawable;      /* i.d. of Drawable */
  unsigned int mask;    /* mask indicating which buffers are affected*/
  int x, y;
  int width, height;
  int count;                 /* if nonzero, at least this many more */
} GLXBufferRestoreEvent;
```

A single X server operation can cause several buffer clobber events to be sent, for example, a single GLXPbuffer may be damaged and cause multiple buffer clobber events to be generated. Each event specifies one region of the GLXDrawable that was affected by the X server operation.

A *mask* argument to **glXSelectEventSGIX()** indicates which color and ancillary buffers were affected. All the buffer clobber events generated by a single X server action are guaranteed to be contiguous in the event queue. The conditions under which this event is generated and the event type varies, depend on the type of the GLXDrawable:

- For a preserved GLXPbuffer, a buffer clobber event, with type GLX_SAVED_SGIX, is generated whenever the contents of the GLXPbuffer are swapped out to host memory. The event(s) describes which portions of the GLXPbuffer were affected. Clients who receive many buffer clobber events, referring to different save actions, should consider freeing the GLXPbuffer resource to prevent the system from thrashing due to insufficient resources.
- For a volatile GLXPbuffer, a buffer clobber event with type GLX_DAMAGED_SGIX is generated whenever a portion of the GLXPbuffer becomes invalid. The client may wish to regenerate the invalid portions of the GLXPbuffer.
- For windows, buffer clobber events with type GLX_DAMAGED_SGIX or GLX_SAVED_SGIX occur whenever an ancillary buffer associated with the window is clobbered or swapped out to host memory. The event contains information indicating which color and ancillary buffers—and which portions of those buffers—were affected.

Calling **glXSelectEventSGIX()** overrides any previous event mask that was set by the client for the drawable. Note that it does not affect the event masks that other clients may have specified for a drawable, since each client rendering to a drawable has a separate event mask for it.

To find out which GLX events are selected for a window or GLXPbuffer, call **glXGetSelectedEventSGIX()**:

```
void glXSelectEventSGIX(Display *dpy, GLXDrawable drawable,
                        unsigned long mask)
```

New Functions

glXCreateGLXPbufferSGIX(), glXDestroyGLXPbufferSGIX()
glXGetGLXPbufferStatusSGIX(), glXGetGLXPbufferConfigSGIX(),
glXGetLargestGLXPbufferSGIX().

The Video Source Extension

The video source extension, SGIX_video_source, lets you source pixel data from a video stream to the OpenGL renderer. The video source extension is available only for system configurations that have direct hardware paths from the video hardware to the graphics accelerator. On other systems, you need to transfer video data to host memory and then call **glDrawPixels()** or **glTex{Sub}Image()** to transfer data to the framebuffer or to texture memory.

The video source extension introduces a new type of GLXDrawable—GLXVideoSourceSGIX—that is associated with the drain node of a Video Library (VL) path. A GLXVideoSourceSGIX drawable can only be used as the *read* parameter to **glXMakeCurrentReadSGI()** to indicate that pixel data should be read from the specified video source instead of the framebuffer.

Caution: This extension is an SGIX (experimental) extension. The interface may change, or it may not be supported in future releases.

The remainder of this section presents two examples: Example 9-1 demonstrates the video to graphics capability of the Sirius video board using OpenGL. Example 9-2 is a code fragment for how to use the video source extension to load video into texture memory.

Example 9-1 Use of the Video Source Extension

```
/*
 * vidtogfx.c
 *  This VL program demonstrates the Sirius Video board video->graphics
 *  ability using OpenGL.
 *  The video arrives as fields of an interlaced format.  It is
 *  displayed either by interlacing the previous and the current
 *  field or by pixel-zooming the field in Y by 2.
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vl/vl.h>
#include <vl/dev_sirius.h>
#include <GL/glx.h>
#include "xwindow.h"
#include <X11/keysym.h>

/* Video path variables */
VLServer svr;
VLPath path;
VLNode src;
VLNode drn;
/* Video frame size info */
VLControlValue size;

int F1_is_first;                /* Which field is first */

/* OpenGL/X variables */
Display *dpy;
Window window;
GLXVideoSourceSGIX glxVideoSource;
GLXContext ctx;
GLboolean interlace = GL_FALSE;
/*
 * function prototypes
 */
void usage(char *, int);
void InitGfx(int, char **);
void GrabField(int);
void UpdateTiming(void);
void cleanup(void);
void ProcessVideoEvents(void);
static void loop(void);
```

```
int
main(int argc, char **argv)
{
    int         c, insrc = VL_ANY;
    int         device = VL_ANY;
    short       dev, val;
    /* open connection to VL server */

    if (!(svr = vlOpenVideo(""))) {
        printf("couldn't open connection to VL server\n");
        exit(EXIT_FAILURE);
    }

    /* Get the Video input */
    src = vlGetNode(svr, VL_SRC, VL_VIDEO, insrc);
    /* Get the first Graphics output */
    drn = vlGetNode(svr, VL_DRN, VL_GFX, 0);

    /* Create path   */
    path = vlCreatePath(svr, device, src, drn);
    if (path < 0) {
        vlPerror("vlCreatePath");
        exit(EXIT_FAILURE);
    }
    /* Setup path */
    if (vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE,
                           VL_SHARE) < 0) {
        vlPerror("vlSetupPaths");
        exit(EXIT_FAILURE);
    }
    UpdateTiming();
    if (vlSelectEvents(svr, path,VLStreamPreemptedMask |
                            VLControlChangedMask ) < 0) {
            vlPerror("Select Events");
            exit(EXIT_FAILURE);
    }
    /* Open the GL window for gfx transfers */
    InitGfx(argc, argv);
    /* Begin Transfers */
    vlBeginTransfer(svr, path, 0, NULL);
    /* The following sequence grabs each field and displays it in
     * the GL window.
     */
    loop();
}
```

```
void
loop()
{
  XEvent event;
  KeySym key;
  XComposeStatus compose;
  GLboolean clearNeeded = GL_FALSE;

  while (GL_TRUE) {
    /* Process X events */
    while(XPending(dpy)) {
      XNextEvent(dpy, &event);
      /* Don't really need to handle expose as video is coming at
       * refresh speed.
       */
      if (event.type == case KeyPress) {
        XLookupString(&event.xkey, NULL, 0, &key, NULL);
        switch (key) {
         case XK_Escape:
          exit(EXIT_SUCCESS);
         case XK_i:
          if (hasInterlace) {
            interlace = !interlace;
            if (!interlace) {
              if (!glXMakeCurrentReadSGI(dpy, window,
                                         glxVideoSource, ctx)) {
                fprintf(stderr,
                        "Can't make current to video\n");
                exit(EXIT_FAILURE);
              }
            } else if (!glXMakeCurrent(dpy, window, ctx)) {
              fprintf(stderr,
                      "Can't make window current to context\n");
              exit(EXIT_FAILURE);
            }
            printf("Interlace is %s\n", interlace ? "On" : "Off");
            /* Clear both buffers */
            glClear(GL_COLOR_BUFFER_BIT);
            glXSwapBuffers(dpy, window);
            glClear(GL_COLOR_BUFFER_BIT);
            glXSwapBuffers(dpy, window);
            glRasterPos2f(0, size.xyVal.y - 1);
          } else {
            printf("Graphics interlacing is not supported\n");
          }
```

```
          break;
        }
      }
    }
    ProcessVideoEvents();
    GrabField(0);
    glXSwapBuffers(dpy, window);
    GrabField(1);
    glXSwapBuffers(dpy, window);
  }
}

/*
 * Open an X window of appropriate size and create context.
 */
void
InitGfx(int argc, char **argv)
{
  int i;
  XSizeHints hints;
  int visualAttr[] = {GLX_RGBA, GLX_DOUBLEBUFFER, GLX_RED_SIZE, 12,
                      GLX_GREEN_SIZE, 12, GLX_BLUE_SIZE, 12,
                      None};
  const char *extensions;

  /* Set hints so window size is exactly as the video frame size */
  hints.x = 50; hints.y = 0;
  hints.min_aspect.x = hints.max_aspect.x = size.xyVal.x;
  hints.min_aspect.y = hints.max_aspect.y = size.xyVal.y;
  hints.min_width = size.xyVal.x;
  hints.max_width = size.xyVal.x;
  hints.base_width = hints.width = size.xyVal.x;
  hints.min_height = size.xyVal.y;
  hints.max_height = size.xyVal.y;
  hints.base_height = hints.height = size.xyVal.y;
  hints.flags = USSize | PAspect | USPosition | PMinSize | PMaxSize;
  createWindowAndContext(&dpy, &window, &ctx, 50, 0, size.xyVal.x,
                size.xyVal.y, GL_FALSE, &hints, visualAttr, argv[0]);

  /* Verify that MakeCurrentRead and VideoSource are supported */
  ....
  glxVideoSource = glXCreateGLXVideoSourceSGIX(dpy, 0, svr, path,
                                              VL_GFX, drn);
  if (glxVideoSource == NULL) {
    fprintf(stderr, "Can't create glxVideoSource\n");
```

```
    exit(EXIT_FAILURE);
  }
  if (!glXMakeCurrentReadSGI(dpy, window, glxVideoSource, ctx)) {
    fprintf(stderr, "Can't make current to video\n");
    exit(EXIT_FAILURE);
  }
  /* Set up the viewport according to the video frame size */
  glLoadIdentity();
  glViewport(0, 0, size.xyVal.x, size.xyVal.y);
  glOrtho(0, size.xyVal.x, 0, size.xyVal.y, -1, 1);
  /* Video is top to bottom */
  glPixelZoom(1, -2);
  glRasterPos2f(0, size.xyVal.y - 1);
  glReadBuffer(GL_FRONT);
  /* Check for interlace extension. */
  hasInterlace = ... /* Interlace is supported or not */
}
/*
 * Grab a field. A parameter of  1 = odd Field, 0 = Even Field.
 * Use the global F1_is_first variable to determine how to
 * interleave the fields.
 */
void
GrabField(int odd_field)
{
  /* copy pixels from front to back buffer */
  if (interlace) {
    /* Restore zoom and transfer mode */
    glRasterPos2i(0, 0);
    glPixelZoom(1, 1);
    glCopyPixels(0, 0, size.xyVal.x, size.xyVal.y, GL_COLOR);

    /* Copy the field from Sirius Video to GFX subsystem */
    if (!glXMakeCurrentReadSGI(dpy, window, glxVideoSource, ctx)) {
      fprintf(stderr, "Can't make current to video\n");
      exit(EXIT_FAILURE);
    }
    if (odd_field) {
      if (F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      }
```

```
    } else {
      if (F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      }
    }
#ifdef GL_SGIX_interlace
    if (hasInterlace)
      glEnable(GL_INTERLACE_SGIX);
#endif
    /* video is upside down relative to graphics */
    glPixelZoom(1, -1);
    glCopyPixels(0, 0, size.xyVal.x, size.xyVal.y/2, GL_COLOR);
    if (!glXMakeCurrent(dpy, window, ctx)) {
      fprintf(stderr, "Can't make current to original window\n");
      exit(EXIT_FAILURE);
    }
#ifdef GL_SGIX_interlace
    if (hasInterlace)
      glDisable(GL_INTERLACE_SGIX);
#endif
  } else {
    /* Not deinterlacing */
    glPixelZoom(1, -2);
    if (!odd_field) {
      if (!F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      }
    } else {
      if (!F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      }
    }
```

```
glCopyPixels(0, 0, size.xyVal.x, size.xyVal.y/2, GL_COLOR);
  }
}

/*
 * Get video timing info.
 */
void
UpdateTiming(void)
{
  int is_525;
  VLControlValue timing, dominance;

  /* Get the timing on selected input node */
  if (vlGetControl(svr, path, src, VL_TIMING, &timing) <0) {
    vlPerror("VlGetControl:TIMING");
    exit(EXIT_FAILURE);
  }
  /* Set the GFX Drain to the same timing as input src */
  if (vlSetControl(svr, path, drn, VL_TIMING, &timing) <0) {
    vlPerror("VlSetControl:TIMING");
    exit(EXIT_FAILURE);
  }
  if (vlGetControl(svr, path, drn, VL_SIZE, &size) <0) {
    vlPerror("VlGetControl");
    exit(EXIT_FAILURE);
  }
  /*
   * Read the video source's field dominance control setting and
   * timing, then set a variable to indicate which field has the first
   * line, so that we know how to interleave fields to frames.
   */
  if (vlGetControl(svr, path, src,
                   VL_SIR_FIELD_DOMINANCE, &dominance) < 0) {
    vlPerror("GetControl(VL_SIR_FIELD_DOMINANCE) on video source
                                                         failed");
    exit(EXIT_FAILURE);
  }

  is_525 = ( (timing.intVal == VL_TIMING_525_SQ_PIX) ||
             (timing.intVal == VL_TIMING_525_CCIR601) );

  switch (dominance.intVal) {
    case SIR_F1_IS_DOMINANT:
```

```
      if (is_525) {
        F1_is_first = 0;
      } else {
        F1_is_first = 1;
      }
      break;
    case SIR_F2_IS_DOMINANT:
      if (is_525) {
        F1_is_first = 1;
      } else {
        F1_is_first = 0;
      }
      break;
  }
}

void
cleanup(void)
{
  vlEndTransfer(svr, path);
  vlDestroyPath(svr, path);
  vlCloseVideo(svr);
  exit(EXIT_SUCCESS);
}

void
ProcessVideoEvents(void)
{
  VLEvent ev;

  if (vlCheckEvent(svr, VLControlChangedMask|
                   VLStreamPreemptedMask, &ev) == -1) {
    return;
  }
  switch(ev.reason) {
    case VLStreamPreempted:
      cleanup();
      exit(EXIT_SUCCESS);
    case VLControlChanged:
      switch(ev.vlcontrolchanged.type) {
        case VL_TIMING:
        case VL_SIZE:
        case VL_SIR_FIELD_DOMINANCE:
          UpdateTiming();
          /* change the gl window size */
```

```
            XResizeWindow(dpy, window, size.xyVal.x, size.xyVal.y);
            glXWaitX();
            glLoadIdentity();
            glViewport(0, 0, size.xyVal.x, size.xyVal.y );
            glOrtho(0, size.xyVal.x, 0, size.xyVal.y, -1, 1);
            break;
          default:
            break;
        }
        break;
      default:
        break;
  }
}
```

Example 9-2 Loading Video Into Texture Memory

```
Display *dpy;
Window win;
GLXContext cx;
VLControlValue size, texctl;
int tex_width, tex_height;
VLServer svr;
VLPath path;
VLNode src, drn;

static void init_video_texturing(void)
{
    GLXVideoSourceSGIX videosource;
    GLenum intfmt;
    int scrn;
    float s_scale, t_scale;

    /* set video drain to texture memory */
    drn = vlGetNode(svr, VL_DRN, VL_TEXTURE, 0);

    /* assume svr, src, and path have been initialized as usual */

    /* get the active video area */
    if (vlGetControl(svr, path, src, VL_SIZE, &size) < 0) {
        vlPerror("vlGetControl");
    }
    /* use a texture size that will hold all of the video area */
    /* for simplicity, this handles only 1024x512 or 1024x1024 */
```

```
    tex_width = 1024;
    if (size.xyVal.y > 512) {
        tex_height = 1024;
    } else {
        tex_height = 512;
    }
    /* Set up a texture matrix so that texture coords in 0 to 1    */
    /* range will map to the active video area.  We want           */
    /* s' = s * s_scale                                            */
    /* t' = (1-t) * t_scale  (because video is upside down).       */
    s_scale = size.xyVal.x / (float)tex_width;
    t_scale = size.xyVal.y / (float)tex_height;
    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();
    glScalef(s_scale, -t_scale, 1);
    glTranslatef(0, t_scale, 0);

    /* choose video packing mode */
    texctl.intVal = SIR_TEX_PACK_RGBA_8;
    if (vlSetControl(svr, path, drn, VL_PACKING, &texctl) <0) {
        vlPerror("VlSetControl");
    }
    /* choose internal texture format; must match video packing mode */
    intfmt = GL_RGBA8_EXT;

    glEnable(GL_TEXTURE_2D);
    /* use a non-mipmap minification filter */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    /* use NULL texture image, so no image has to be sent from host */
    glTexImage2D(GL_TEXTURE_2D, 0, intfmt, tex_width, tex_height, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, NULL);

    if ((videosource = glXCreateGLXVideoSourceSGIX(dpy, scrn, svr,
                                     path, VL_TEXTURE, drn)) == None) {
        fprintf(stderr, "can't create video source\n");
        exit(1);
    }
    glXMakeCurrentReadSGI(dpy, win, videosource, cx);
}

static void draw(void)
{
    /* load video into texture memory */
    glCopyTexSubImage2DEXT(GL_TEXTURE_2D, 0, 0, 0, 0, 0,
                           size.xyVal.x, size.xyVal.y);
```

```
    /* draw the video frame */
    glBegin(GL_POLYGON);
    glTexCoord2f(0,0); glVertex2f(0, 0);
    glTexCoord2f(1,0); glVertex2f(size.xyVal.x, 0);
    glTexCoord2f(1,1); glVertex2f(size.xyVal.x, size.xyVal.y);
    glTexCoord2f(0,1); glVertex2f(0, size.xyVal.y);
    glEnd();

}
```

New Functions

glXCreateGLXVideoSourceSGIX(), glXDestroyGLXVideoSourceSGIX().