Obtaining Maximum Performance on Silicon Graphics Prism™ Visualization Systems

This document provides guidance about how to get the best performance from a Silicon Graphics Prism Visualization system. It covers the following topics:

Rendering

The Silicon Graphics Prism graphics subsystem has two or more graphics pipes, each with a single GPU and 256MB of local memory. Each graphics pipe has its own full-speed AGP8x interface, supplying bus bandwidths of up to 2.1GB/s and providing the connection between host-system main memory and local graphics memory. All graphics data is sent over this AGP8x bus to the graphics pipe, where it is rendered and then displayed.

Programming Model

The use of a “retained-mode” OpenGL programming model will usually provide the best graphics performance on a Prism graphics system. This retained-mode model is in contrast to the immediate-mode model widely used on previous SGI graphics systems.

A retained-mode OpenGL programming model is one in which the geometry or pixel data resides in the local memory of the graphics pipe, rather than in the main memory of the host system. OpenGL commands that create objects (e.g., display lists, vertex buffer objects, vertex array objects, or texture objects) are retained-mode commands. All of these OpenGL commands return an identifier that allows the association of the object to the data that is resident on the graphics pipe. Other OpenGL commands that send data to the graphics pipe generally keep the data in the Prism system's main memory and send it to the graphics pipe each time it is needed.

Note: The Prism system supports both vertex buffer objects and vertex array objects. Vertex buffer objects (the ARB_vertex_buffer_object extension) are recommended. Vertex array objects (the ATI_vertex_array_object extension) are ATI®-specific, and are supported for purposes of porting and compatibility.

Data Commands and Formats

After the selection of a programming model, the next most important consideration for high graphics performance is the data format used when sending data to the graphics pipes.

Some data formats need special software handling, while others are on the “fast path,” that is, they move from Prism main memory through the graphics pipes very quickly.

Once the data is in local memory on the graphics pipes these fastpath formats will also generally render more quickly than other formats.

Geometry fastpaths are primarily dictated by using display lists, vertex array objects, or vertex buffer objects. Within a display list, it is most beneficial to provide uniform data per vertex. This means that if color or normals are provided, they should be provided per vertex rather than per facet.

Pixel fastpaths are more variable than geometry fastpaths. Pixel data can be rendered either as a texture or directly via `glDrawPixels()`. Generally, using a texture to render the data is faster than using `glDrawPixels()`. Though there is some overhead associated with creating textures, using texture objects is generally faster than using immediate mode for textures. When the texture data changes every frame, this overhead can become significant. In this case, the overhead can be minimized by creating a single texture object and using one of the `glTexSubImage*()` family of commands to download the texture data into that object.

Besides the command structure used to download data to the card, the format of the data can make a significant difference in graphics performance. Because the performance of a given format can be increased as the driver software is improved, the general guidelines given here may not apply to later releases of the Prism graphics driver.

RGBA formats run faster than RGB formats, and BGRA / BGR formats run slightly faster than RGBA / RGB formats.

The fastest data types are the `GL_UNSIGNED_BYTE` and `GL_BYTE` types, followed by the `GL_FLOAT` types. The one anomaly is that single-channel formats are faster with `GL_FLOAT` data than with `GL_UNSIGNED_BYTE` data.

Generally, `GL_UNSIGNED_BYTE` is faster than the other signed or multi-byte formats, although some of the `GL_FLOAT` formats are also on the fastpath.

Sending a texture in any format other than the one used internally by the graphics card requires the card to convert the texture to its native format (an operation sometimes called a “swizzle”). Performing this operation will degrade performance.

Overall performance may improve by spending additional time in software to reformat pixel data, thus allowing that data to be sent to the graphics pipe in the most appropriate format. The trade-off between CPU time spent reformatting data and performance lost by exiting the fastpath must be considered individually for each application.

Readback

When reading pixel data back from the graphics pipe, the fastpaths are similar to those that are on the download fastpaths, although the readback fastpath set is much smaller. The color buffer can be read back quickly using the RGBA and BGRA `GL_UNSIGNED_BYTE` formats, the BGRA format being slightly faster than the RGBA.

Graphics Memory Usage

In most cases using graphics memory is significantly faster than using main memory. Exceeding the capacity of graphics memory, however, will cause memory thrashing, and can drastically reduce performance. This section provides guidance regarding the memory capacity of the Prism graphics pipes.

The 256 Megabytes of graphics memory on the Prism graphics cards is divided into two partitions: 128 Megabytes of framebuffer memory and 128 Megabytes of retained-object memory. The framebuffer memory partition size is a hard limit, and may not be exceeded. The retained-object memory partition size, however, is a softer limit. When the retained-object memory partition fills, the driver will automatically use any available framebuffer memory. If framebuffer memory is not available, or once it has been used up, the driver will then swap to main (system) memory (an operation that can cause thrashing, and thus reduce graphics performance).

The framebuffer partition is used for front buffer, back buffer, depth buffer, accumulation buffer, stereo buffers, multisample buffers, driver housekeeping, and various other purposes (including overflow from the retained-object memory partition).

The retained-object memory partition is used for display lists, vertex buffer objects, vertex array objects, and textures.

Information about graphics card memory usage may be found in the file:

```
/proc/dri/*/umm
```

where `*` is the pipe number of the card in question. The output will look something like:

```
-bash-2.05b$ cat /proc/dri/0/umm
free  AGP = 1065091072
max   AGP = 1065091072
free  LFB = 116391936
max   LFB = 116391936
free  Inv = 134217728
max   Inv = 134217728
total Inv = 134217728
total TIM = 0
-bash-2.05b$
```

This file provides a rough snapshot of how memory on the card is being used, and may be helpful in program design.

Note: In many cases a simple calculation of the size of the data being sent to the card will differ from the amount of memory actually used. This is due to factors such as driver overhead, format conversions, and compression.

The contents of the `/proc/dri/*/umm` file can be interpreted as follows:

- "Free" indicates the total amount of free memory in that partition.
- "Max" indicates the size of the largest available memory block within that partition.
- The "AGP" listing represents the portion of main (system) memory mapped to that card's AGP bus.
- The "LFB" listing represents framebuffer memory (called LFB for Local Frame Buffer).

  Note: A small portion of the framebuffer memory is always reserved for driver use, leaving slightly less than 128 Megabytes for application use.

- The "INV" listing represents the retained-object memory (called INV for "invisible"). Any retained-mode objects will therefore reduce the amount of available INV memory.
- The "TIM" listing represents TIMMO local frame buffer, and is a portion of invisible memory. As this number increases, INV will decrease. (TIMMO is described in "Disabling Turbo Immediate Mode (TIMMO)" on page 10.)

Typically the best performance will be obtained when data is stored on the graphics card. When the available retained-object memory on the graphics card reaches zero, however, the system will swap to main memory. This memory swapping can reduce performance.

Environment Variables

There are a number of environment variables that may be used to tune graphics performance on a Prism system. These variables adjust the behavior of the graphics driver and can be used to optimize display list and immediate mode performance, among other functions. Though the default settings will typically provide the best performance, this section describes those variables that might help improve performance in some situations.

Other systems often have GUI control panels that adjust the settings of many OpenGL-related variables, as well as application control panels that can be used to choose a group of settings that work best with specific applications. Instead of such a GUI, the SGI Prism system exposes those same OpenGL variables, as well as many others, allowing finer control of graphics system settings and performance.

Miscellaneous Environment Variables

Table 1 details a number of miscellaneous environment variables.

Table 1 Miscellaneous Environment Variables

Variable	Values	Description
GLERRORABORT	yes, YES	Forces application to exit on any GL error condition. [Off by default]
GLFORCEDIRECT	yes, YES no, NO	Forces direct rendering when DISPLAY specifies localhost, regardless of the glXCreateContext allowDirect parameter. Forces indirect rendering regardless of the glXCreateContext allowDirect parameter.
DECOUPLE_SWAPBUF	any value	Disables swapbuffers sync to vertical blank.
__GL_SYNC_TO_VBLANK	any value	Enables swapbuffers sync to vertical blank.
FGL_DISABLE_DYNAMIC_FSAASCALE	any value	Disables reduced-sample FSAA retries (see “Full-Scene Anti-Aliasing” on page 14)

Display List Optimizer

The graphics driver contains a display list optimizer, which in most cases improves display list performance. There are some cases, however, where the best performance is obtained by fully or partially disabling the optimizer, such as when an application defines many individual small begin-end primitives. Table 2 describes how to disable the display-list optimizer in these cases.

Table 2 Display-List Optimizer Environment Variables

Variable	Values	Description
FGL_DLOPT_FLAGS	bitwise value	Enables individual display-list optimization stages with bit flags (identified using FGL_DLOPT_INFO or from Table 3 on page 9).
	0	Disables all display-list optimization.
	1319	Sets display-list optimizations to default values.
FGL_DLOPT_INFO	1, 2	Details which display-list optimizer stages are enabled. 1: Basic 2: Verbose

Table 3 shows the individual display-list optimizer stages, the values to which they may be set, the bit value that can be used to enable or disable each one using the FGL_DLOPT_FLAGS environment variable, and a description of the function of each.

Table 3 Display-List Optimizer Environment Variables (the Bolded Bit Values are on by Default)

Variable	Values	Bit Value	Display-List Optimization
FGL_DLOPT_REORDER_COLOR_MATERIAL	0, 1[a]	**1**	ColorMaterial
FGL_DLOPT_CONV_TO_DRAWARRAY	0, 1[a]	**2**	Convert to DrawArray
FGL_DLOPT_CONNECT_DRAWARRAYS	0, 1[a]	**4**	Connect DrawArrays
FGL_DLOPT_COMPRESS_DRAWARRAY	0, 1[a]	8	Compress DrawArray
FGL_DLOPT_CONV_TO_MULTI	0, 1[a]	16	Convert to MultiDrawArrays
FGL_DLOPT_BOUNDING_TREE	0, 1[a]	**32**	Insert bounding tree
FGL_DLOPT_GLOBAL_BOUNDING_BOX	0, 1[a]	64	Insert global bounding box
FGL_DLOPT_LOCAL_BOUNDING_BOX	0, 1[a]	128	Insert local bounding box
FGL_DLOPT_CONV_TO_HW	0, 1[a]	**256**	Cache display-list data in graphics memory
FGL_DLOPT_CONNECT_TRISTRIP_ARRAYS	0, 1[a]	**1024**	Connect DrawArrays for tri-strip primitives only

a. Each of the ten individual display-list optimization stages may be enabled by setting them to 1, “yes”, or “YES”; disabled by setting them to 0, “no”, or “NO”.

Disabling Turbo Immediate Mode (TIMMO)

Turbo Immediate Mode (sometimes referred to as TIMMO) performs vertex caching, typically increasing performance in immediate mode, especially when geometry is consistent across frames. There are some cases, however, where the best performance is obtained by disabling this feature.

Note: The `FGL_DISABLE_TIMMO` environment variable will be included in a future release of the Prism software. Until this variable is available, TIMMO may be disabled on a global basis in the `XF86Config` file, as detailed in "Disabling Turbo Immediate Mode (TIMMO)" on page 16.

Table 4 details the Turbo Immediate Mode environment variable.

Table 4 Turbo Immediate Mode Environment Variable

Variable	Values	Description
`FGL_DISABLE_TIMMO`	`any value`	`Disables Turbo Immediate Mode.`

Pixel Operation Environment Variables

Table 5 details a number of environment variables related to pixel operations.

Table 5 Pixel Operation Environment Variables

Variable	Values	Description
FGL_DISABLE_FAST_BLIT	any value	Disables fast blit for drawpixels, readpixels and accumbuffer operations.
FGL_DISABLE_SGI_FASTBLIT	any value	Disables SGI fast blit optimization for drawpixels.
SGI_FASTBLIT_INFO	any value	Provides feedback when SGI fast blit path is being employed.
FGL_MACRO_TILE_FB	any value	Macro tile all pbuffer and private framebuffers.
FGL_MACRO_TILE_SZ	any value	Macro tile private depth and stencil buffers.
FGL_MACRO_TILE_RGBA	any value	Macro tile pbuffer and private colorbuffers.

Texture Operation Environment Variables

Table 6 details a number of environment variables related to texture operations. Also see "Texture Compression" on page 23.

Table 6 Texture Operation Environment Variables

Variable	Values	Description
FGL_DYNAMIC_TEXIMAGE	any value	Causes TexImage2D & TexSubImage2D commands to copy texture images directly from user memory to graphics memory, without maintaining a host-side driver copy (when possible). This option can improve performance when a texture will not be reused, as in streaming video.
FGL_NO_UNCOMPRESSED_TEXTURE	any value	Causes the driver not to maintain an uncompressed copy of textures that it compresses.
OGLEnableTextureCompression	0, **1**	Support for compressed textures. [Enabled by default]
OGLForceTextureCompressionRGB	0, **1**	Forces compression of RGB textures. [Enabled by default]
OGLForceTextureCompressionRGBA	**0**, 1	Forces compression of RGBA textures. [Disabled by default]
OGLTextureOpt	0-3	Texture quality (controls texture compression & trilinear filtering): 0: High quality 1: Quality 2: Performance 3: High performance

Table 6 (continued) Texture Operation Environment Variables

Variable	Values	Description
OGLLODBias	0-3	Adjusts level of detail (LOD) setting: 0: High quality 1: Quality 2: Performance 3: High performance
OGLMaxAnisotropy	0, 2, 4, 8, 16	Sets maximum anisotropy level: 0: Application preferred 2-16: 2x through 16x
OGLAnisoType	0, 1, 2	Sets anisotropy type: 0: Application default 1: Performance 2: Quality

XFree86™ Configuration

Prism Visualization Systems use the XFree86 windowing system. Much of the configuration of this windowing system is done in the `/etc/X11/XF86Config` file. A few such configuration options are described here. Information about other options may be found in the *Silicon Graphics Prism Visualization System User's Guide.*

Full-Scene Anti-Aliasing

Full-scene anti-aliasing (FSAA) may be configured by editing the relevant "Device" section of the `/etc/X11/XF86Config` file to include the following lines:

```
Option "FSAAScale" "n"
```

where *n* is `0`, `1`, `2`, `4`, or `6`.

Note: Full-scene anti-aliasing is disabled by setting "`FSAAScale`" to `0`. Per-window full-scene anti-aliasing is accomplished by setting "`FSAAScale`" to `1`, in which case the anti-aliasing level may be set by the appropriate selection of visuals. Global anti-aliasing is accomplished by setting "`FSAAScale`" to `2`, `4`, or `6`, in which case the setting will apply to all OpenGL windows, regardless of the visual being displayed.

FSAA requests can quickly exceed the available framebuffer memory. When this happens, the default behavior is for the graphics driver to retry the request once at the requested sample count, then downgrade the request to the next smaller sample count and try again. This process is repeated until either all buffers can be allocated or the request has been downgraded to use no FSAA and still fails. If this happens, the driver gives up and reports a failure.

Since the driver may significantly downgrade the FSAA request without reporting a failure, you may wish to use `glGetInteger(GL_MULTISAMPLE_ARB, X)` to verify the results of the request.

Alternately, reduced-sample retries may be disabled using the environment variable `FGL_DISABLE_DYNAMIC_FSAASCALE`.

Stereo

Stereo may be enabled by editing the relevant “Device” section of the `/etc/X11/XF86Config` file to include the following lines:

```
Option "Stereo"           "on"
Option "StereoSyncEnable" "1"
```

You must also ensure that the “Monitor” section contains a suitable stereo mode.

Enabling stereo doubles the amount of framebuffer memory required, and—especially when combined with full-scene anti-aliasing—can easily exceed the available framebuffer memory.

Note: Overlay planes, stereo, and dual-channel operation are mutually exclusive. A Prism graphics pipe may use at most one of these three features at any one time.

Overlay Planes

Overlay planes allow rendering to take place in a separate layer, distinct from the main framebuffer plane, and without affecting that main plane. This may be particularly useful when the data in the main frame buffer is very complex.

Overlay planes may be enabled by editing the relevant “Device” section of the `/etc/X11/XF86Config` file to include the following lines:

```
Option "OpenGLOverlay"           "on"
```

Note: Overlay planes are limited to pseudo-color (also called color index mode), which provides access to a limited set of colors selected from a larger color palette.

Note: Overlay planes, stereo, and dual-channel operation are mutually exclusive. A Prism graphics pipe may use at most one of these three features at any one time.

Disabling Turbo Immediate Mode (TIMMO)

As described in “Disabling Turbo Immediate Mode (TIMMO)” on page 10, there are cases where it may be desirable to disable Turbo Immediate Mode (TIMMO). This can be done by editing the relevant “Device” section of the `/etc/X11/XF86Config` file to include the following lines:

```
Option "Configuration"          "0x00008000"
```

Note: The environment variable `FGL_DISABLE_TIMMO` will be included in a future release of the Prism software and will allow finer control over TIMMO. Until this variable is available, however, TIMMO may be disabled on a global basis in the `XF86Config` file, as described in this section.

Window Manager Considerations

This section describes configuring and using your window manager to get the best graphics performance from your Prism system. Some of the configurations described here are initially set in global configuration files, but may be overridden in individual users' configuration files. Such overriding may significantly impact graphics performance for that user.

Avoid Complex Window Schemes

The default window scheme for Prism systems is intentionally of minimal complexity. This simpler scheme allows for greater graphics performance. If you change your desktop scheme, be aware that curved or shaped window borders can significantly reduce graphics performance when these windows are placed on top of a window containing OpenGL.

Avoid Stacked Windows

Placing any window in front of a window containing OpenGL can significantly reduce the graphics speed in the OpenGL window. This problem is exacerbated when multiple windows are stacked over one with OpenGL, and is further exacerbated when those windows have curved or shaped borders.

Desktop Icons on Non-Primary Screen

The default KDE window scheme for Prism systems disables the display of desktop icons on all but the primary screen (i.e., screen 0, or the first X screen). Re-enabling display of these icons could cause minor visual artifacts, and is therefore not recommended.

Platform Parameters

There are a number of areas not directly related to graphics, but which nevertheless have an impact on graphics performance. This section addresses some of those areas.

Ensuring That Jobs Run Node-Local

Since Prism systems use a CC-NUMA (cache-coherent non-uniform memory access) architecture, the CPU on which a process runs will determine the location of the memory used for that process. Graphics processes that use memory 'closer' to their graphics pipe will perform better than those that use memory 'farther' from that pipe.

This section describes tools that may be used to determine which CPUs are closest to a particular graphics pipe, and other tools that can ensure that a particular process runs on those closest CPUs (i.e., that the process runs "node-local").

For additional information on this topic, search for "NUMA tools" in the Linux portion of the SGI Tech Pubs Library (`http://techpubs.sgi.com`).

The gfxtopology Script

The script `gfxtopology` may be used to determine which CPUs are closest to a particular graphics pipe. Typically you will know which graphics pipe you want to use for a particular process (assume pipe 0 for the example below). Then, using the `gfxtopology` script (note the use of the `-v` option), you can determine the closest CPUs.

Note: Prism software prior to SGI ProPack™ 3, Service Pack 3 included an earlier version of the `gfxtopology` script which did not provide as much information as the version documented here.

```
-bash-2.05b$ gfxtopology -v
pipe 0  g @ 001c02/1/agp/0a -> c @ 001c02/0 (cpu 2,3)    (PCI:23:0:0)   SG2
pipe 1  g @ 001c02/2/agp/0a -> c @ 001c03/0 (cpu 4,5)    (PCI:27:0:0)   SG2
pipe 2  g @ 001c03/1/agp/0a -> c @ 001c03/0 (cpu 4,5)    (PCI:39:0:0)   SG2
pipe 3  g @ 001c03/2/agp/0a -> c @ 001c01/0 (cpu 0,1)    (PCI:43:0:0)   SG2
```

The output of the `gfxtopology` script is detailed in Figure 1 and Table 7.

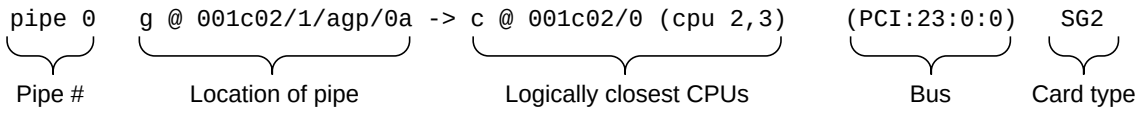

Figure 1 Output of the gfxtopology Script

Table 7 Output of the gfxtopology Script

Description	Explanation
Pipe #	The pipe number described by this line in the gxftopology output (one line for each graphics pipe in the system). Note: By default, this pipe number is used as the screen number for the Xserver (thus pipe 3 would be server connection :0.3).
Location of pipe	g @ 001c02 The brick in which this graphics pipe is located.[a] /1/agp/0a The port within that brick, the bus type, and the device number.
Logically closest CPUs	c @ 001c02/0 The brick containing the closest CPUs[a] and the port within that brick. (cpu 2,3) The CPU numbers located in that brick.
Bus	(PCI:23:0:0) The bus type for XFree86[b] (in this case, PCI), the system-wide bus number (in this case, 23), the device number (in this case, 0), and the function number (in this case, 0).
Card type	The type of graphics card used for that pipe.

a. Bricks may be identified by looking at the level 1 system controller LCD panel on the front of the brick. The brick in rack #1, slot #2 would show “001c02” on its display.

b. Both PCI and AGP buses will be identified here as “PCI” for compatibility with XFree86 conventions.

In the above example, CPU 2 and CPU 3 are closest to pipe 0. This information can then be used with `dplace` or `runon`, described in the next section, to ensure that the application to be displayed on pipe 0 runs node-local.

Note: Due to system architecture factors, a graphics pipe may actually be farther ("more hops") from CPUs in its own brick than it is from the CPUs in a different brick.

The dplace and runon Commands

After using `gfxtopology` (described in "The gfxtopology Script" on page 18) to determine which CPU you want your process to run on, you can then use `dplace` or `runon` to start the process running on that CPU. For example:

`dplace -c`*X command*

[where *command* should be run on CPU *X*].

More information about `dplace` may be found by typing **`man dplace`** at the Linux prompt.

More information about `runon` may be found by typing **`man runon`** at the Linux prompt.

The dlook Command

Another tool which may be useful is `dlook`, which displays memory maps and CPU usage for a particular process.

More information about `dlook` may be found by typing **`man dlook`** at the Linux prompt.

CPU Version Sensitivity

In addition to clock-speed differences, the various CPU versions used in Prism systems have significantly different cache sizes. These cache-size differences can have a pronounced effect on graphics performance. When looking for the best graphics performance choose the highest CPU speed and cache size available. When comparing

performance between two Prism systems, ensure that both systems have the same CPU speed and cache size.

Maximizing Bus Throughput

The best graphics performance on Prism systems will be obtained when primitives are coalesced into the largest groups practical. This practice helps to take full advantage of the system's extremely high I/O bandwidth, while minimizing the impact of its latencies.

This is typically accomplished by using a retained-mode programing model (as described in "Programming Model" on page 2), and avoiding switching contexts (for example, avoid using X rendering within OpenGL windows and use full-screen applications when possible).

VTune, Thread Checker, and Thread Profiler

A suite of tools from Intel®, VTune™ Performance Analyzer for Linux, Thread Checker, and Thread Profiler, help to locate performance bottlenecks.

VTune Performance Analyzer for Linux (sometimes referred to as "VTL") helps improve software performance by finding bottlenecks and hotspots through advanced profiling technologies.

Thread Checker and Thread Profiler help to show how thread overhead and thread synchronization impact your application's performance.

Further information about these tools, as well as purchase information, is available on Intel's website (`http://www.intel.com`).

Note: Thread Profiler and Thread Checker run only on Microsoft® Windows® systems. Although a native Linux® version of VTune is available, the remote-agent version simplifies analysis by using the Windows VTune GUI and the other Windows-only tools.

SGI Histx

Another useful group of tools, collectively called SGI Histx, may be downloaded free from SGI at the following URL:

```
http://www.sgi.com/products/evaluation/altix_histx/
```

These tools assist with application performance analysis. They contain a profiling tool that can sample either instruction pointer or call stack on either timer interrupts or performance monitor counter overflows, two tools for reporting performance monitor event counts, and three filters.

Miscellaneous Issues

A number of other factors that can have an effect on performance are described in this section.

Texture Compression

The SGI Prism graphics pipes can accept textures compressed in S3 Texture Compression (S3TC®) formats (also known as DX Texture Compression).

The Prism graphics pipes can decompress textures, but can not compress them. Therefore in order to use texture compression the textures must be compressed before being sent to the pipe. If your textures have not been compressed previously, you can compress them by calling `glTexImage*()` with an internal texture format of one of the `GL_COMPRESSED_*` family of enumerants.

You may wish to use `glGetCompressedTexImage()` to get the compressed texture back from the driver. This can save CPU time that would otherwise be required to perform the compression operation again if you later reuse the same texture.

A number of environment variables related to texture compression are detailed in “Texture Operation Environment Variables” on page 12.

Creating MIPmaps

When using MIPmaps, it may save time to use the `GL_SGIS_generate_mipmap` extension. This extension sends the texture to the graphics pipe, which then creates the MIPmaps locally, thereby reducing texture download time.

Swap Barriers

There are some special considerations regarding the use of swap barriers, addressed below.

Swap Barriers and Multiple Applications

If an application asserts control over the swap rate on a pipe, any other applications running on that same pipe will be affected.

This is due to the fact that the hardware control for swapping is global, rather than per context, and is expected behavior.

Graphic Pipe Lockup with Swap Barriers

It is possible to trigger a condition that will lock up graphics pipes when using applications that make use of swap barriers and other dependent application calls (e.g., to X in a window update). For example, take the case of an application running a swap barrier while the user drags a window on another pipe:

1. The application on Pipe 0 has called SwapBuffers and is waiting for the swapready line to rise, in addition it has taken a lock on the graphics device in order to guarantee exclusive access.
2. An application on Pipe 1 has called SwapBuffers and realizes that a window invalidate has happened and it has to contact the Xserver for the new clip rectangles.

The Xserver wants to lock graphics down to prevent any window rectangle changes before it responds. However, it needs to wait for the graphics lock to be free before it can issue its swapbuffers. Hence a deadlock exists.

This situation does not occur very often. The deadlock can be broken by killing the application on the hung pipe from another pipe or from the console.

The best way to minimize this problem is to run swap barriered applications in full-screen mode. Then at least it is a single application holding the lock and managing the swapbuffers.

There is no short-term fix for this problem. In the example of the application / X deadlock, this is inherently a problem with the Xserver's single-threaded nature.

Written by Eric Zamost

Acknowledgments: Major portions of this document were contributed by Alan Commike. Many others helped, including: Terrence Crane, Bill Feth, Brad Grantham, Alpana Kaulgud, Bob Kozdemba, Eric Kunze, Yaron Lachman, Jon Leech, Martin McDonald, Dan McLachlan, Shrijeet Mukherjee, Peter Ostrin, Arthur Raefsky, Dave Shreiner, and Andrew Spray