# OpenGL™ 1.1.1 For Solaris™ Implementation and Performance Guide

## Solaris™ Version

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 805-3145-10
February 1998, Revision A

Please
Recycle

Adobe PostScript

# Contents

# Figures

# Tables

# Preface

*OpenGL 1.1.1 For Solaris Implementation and Performance Guide* provides information on the Solaris™ OpenGL™ software.

## Who Should Use This Book

This book is intended for application developers who are using the Solaris OpenGL software to port OpenGL applications to Sun hardware. It assumes familiarity with OpenGL functionality and with the principles of 2D and 3D computer graphics.

## How This Book Is Organized

This book is organized as follows:

**Chapter 1 "Introduction to the OpenGL for Solaris Software,"** provides a description of the OpenGL for Solaris software.

**Chapter 2 "OpenGL for Solaris Architecture,"** presents information on the OpenGL for Solaris architecture.

**Chapter 3 "Performance,"** presents specific information on using Sun's OpenGL library for specific hardware platforms.

**Chapter 4 "X Visuals for the OpenGL for Solaris Software,"** presents information on visuals for the OpenGL for Solaris product.

**Chapter 5 "Tips and Techniques,"** contains information that may make using the OpenGL for Solaris library easier.

# Related Books

For information on the OpenGL library, refer to the following books:

- Neider, Jackie, Tom Davis, Mason Woo, *OpenGL Programming Guide*, Reading, Mass., Addison-Wesley, 1993.
- OpenGL Review Board, OpenGL Reference Manual, Reading, Mass., Addison-Wesley, 1992.
- Kilgard, Mark, *OpenGL Programming for X Window Systems*, Reading, Mass., Addison-Wesley, 1996.

# Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

# Accessing Sun Documentation Online

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`.

# Sun Welcomes Your Comments

Sun is interested in improving our documentation and welcome your comments and suggestions. You can email or fax your comments to us. Please include the part number of your document in the subject line of your email or fax message.

- Email:—`smcc-docs@sun.com`
- Fax:—SMCC Document Feedback, 1-650-786-6443

# What Typographic Changes Mean

The following table describes the typographic changes used in this book.

**TABLE P-1** Typographic Conventions

| Typeface or Symbol | Meaning | Example |
|---|---|---|
| `AaBbCc123` | The names of commands, files, and directories; on-screen computer output | Edit your `.login` file.<br>Use `ls -a` to list all files.<br>`machine_name% You have mail.` |
| **`AaBbCc123`** | What you type, contrasted with on-screen computer output | `machine_name% su`<br>`Password:` |
| *AaBbCc123* | Command-line placeholder: replace with a real name or value | To delete a file, type `rm` *filename*. |
| *AaBbCc123* | Book titles, new words or terms, or words to be emphasized | Read Chapter 6 in *User's Guide*. These are called *class* options.<br>You *must* be root to do this. |

# Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

**TABLE P-2** Shell Prompts

| Shell | Prompt |
|---|---|
| C shell prompt | `machine_name%` |
| C shell superuser prompt | `machine_name#` |
| Bourne shell and Korn shell prompt | `$` |
| Bourne shell and Korn shell superuser prompt | # |

CHAPTER 1

# Introduction to the OpenGL for Solaris Software

The OpenGL for Solaris software is Sun's native implementation of the OpenGL application programming interface (API). The OpenGL API is an industry-standard, vendor-neutral graphics library. It provides a small set of low-level geometric primitives and many basic and advanced 3D rendering features, such as modeling transformations, shading, lighting, anti-aliasing, texture mapping, fog, and alpha blending.

## OpenGL 1.1.1 for Solaris Product Functionality

The OpenGL 1.1.1 for Solaris software is a functionally conforming implementation based on the OpenGL 1.1, GLX 1.2, and GLU 1.2 standard specifications. The OpenGL 1.1.1 for Solaris software incorporates the new features in OpenGL and includes support for the `SERVER_OVERLAY_VISUALS` property.

### OpenGL 1.1.1 Library

The OpenGL 1.1.1 library is a superset of OpenGL 1.1, including all OpenGL 1.1 functionality and additional features that were available as extensions to OpenGL 1.0. The added extensions, which are listed in TABLE 1-1 on page 2, have become part of base OpenGL functionality; however, the semantics or syntax may have changed

in OpenGL 1.1.1 for Solaris. For detailed information on the extensions incorporated into the OpenGL 1.1 specification, see Appendix C in *The OpenGL Graphics System: A Specification, Version 1.1*.

**TABLE 1-1** OpenGL 1.0 Extensions That Have Become Apart of Base OpenGL 1.1 Functionality

| OpenGL 1.1 Name | Extension Name | Changed Syntax or Semantics |
|---|---|---|
| Vertex arrays | `GL_EXT_vertex_array` | Yes |
| Polygon offset | `GL_EXT_polygon_offset` | Yes |
| RGBA logical operations | `GL_EXT_blend_logic_op` | No |
| Internal texture image formats | `GL_EXT_texture` | No |
| Texture replace environment | `GL_EXT_texture` | No |
| Texture proxies | `GL_EXT_texture` | Yes |
| Copy texture and subtexture | `GL_EXT_copy_texture`<br>`GL_EXT_subtexture` | No |
| Texture objects | `GL_EXT_texture_object` | Yes |

**Note –** Because the OpenGL 1.1.1 for Solaris software is based on a more current version of the OpenGL specifications (OpenGL 1.1, GLX 1.2, GLU 1.2) than the OpenGL 1.0 version, customers currently using the OpenGL 1.0 extensions syntax should be alert for software changes required to support the updated OpenGL specifications.

# Supported OpenGL 1.1.1 Extensions

The OpenGL 1.1.1 for Solaris software supports the following OpenGL extensions:

- 3D texture mapping extension – `GL_EXT_texture3D`
- ABGR reverse-order color format extension – `GL_EXT_abgr`
- Texture color table extension – `GL_SGI_texture_color_table`
- SGI color table extension – `GL_SGI_color_table`
- Sun geometry compression extension – `GL_SUNX_geometry_compression`
- Rescale normal extension – `GL_EXT_rescale_normal`
- Histogram extension – `GL_EXT_histogram`

- Postconvolution color table extension – `GL_SGI_postconvolution_color_table`
- Convolution extension – `GL_EXT_convolution`
- Blend color extension – `GL_EXT_blend_color`
- Blend minmax extension – `GL_EXT_blend_minmax`
- Blend suptract extension – `GL_EXT_blend_subtract`
- Pixel transform extension – `GL_EXT_pixel_transform`
- Multidrawarrays extension – `GL_SUN_multi_draw_arrays`
- Convolution border mode extension – `GL_HP_convolution_border_modes`
- Convolution border mode extension – `GL_SUN_convoution_border_modes`
- Pixel transformation extension – `GL_EXT_pixel_transform`

---

**Note –** `glBlendEquationEXT(GL_LOGIC_OP)` is not supported. Instead of using the code sequence:

```
glLogicOpEXT(GL_XOR)
glBlendEquationEXT(GL_LOGIC_OP)
glEnable(GL_BLEND)
```

use the following code sequence:

```
glLogicOp(GL_XOR)
/* GL_LOGIC_OP not supported
    in glBlendEquationEXT with
    OpenGL for Solaris */
#ifdef GL_VERSION_1_1
     glEnable(GL_COLOR_LOGIC_OP)
#endif
```

---

The OpenGL 1.1.1 for Solaris software also supports the following GLX extensions:

- Return the transparent pixel index for an overlay/underlay window pair – `GLX_SUN_get_transparent_index`. See the `glXGetTransparentIndexSUN`(3gl) man page.
- fbconfig extension – `GLX_SGIX_fbconfig`
- pbuffer extension – `GLX_SGIX_pbuffer`
- make current read extension – `GLX_SGI_make_current_read`
- Multithread support extension – `GLX_SUN_init_threads`

**Note –** To determine what extensions, if any, your application uses, search for command-name patterns such as `gl`*Procedure*`EXT(`3gl`)`. If your application uses extensions, you will need to ensure that it also handles the functionality in an OpenGL 1.1-compliant manner. To determine what extensions an OpenGL implementation supports, use `glXQueryExtensionString(`3gl). Because the OpenGL 1.1.1 for Solaris software is based on a more current version of the OpenGL specifications (OpenGL 1.1, GLX 1.2, GLU 1.2) than the OpenGL 1.0 version, customers currently using the OpenGL 1.0 extensions syntax should be alert for software changes required to support the updated OpenGL specifications.

# Compatibility Issues

Applications compiled with the previous OpenGL for Solaris libraries will run unchanged with the Solaris OpenGL 1.1.1 implementation. However, note the following backward compatibility issues:

- To reduce function call overhead and improve performance for vertex calls in immediate mode, vertex commands such as `glVertex, glColor, glNormal, glTexCoord` and `glIndex` have been redefined as macros in the OpenGL 1.1.1 for Solaris software. Therefore, by default, applications compiled with the OpenGL 1.1.1 for Solaris library will not run on the 1.0 library. To compile an application with the OpenGL 1.1.1 for Solaris library and maintain compatibility with 1.0, use the flag `-DSUN_OGL_NO_VERTEX_MACROS` when compiling the application. See the `glVertex` (3gl) man page for further information.
- If your application uses the features in the OpenGL 1.1.1 for Solaris library, these are not available in the previous release for OpenGL It will not be backward compatible with the previous OpenGL libraries.

# MT-Safe

The OpenGL for Solaris library is multithread safe (MT-safe). Multiple rendering threads are allowed in a single process. See man page `glXInitThreadsSUN(3gl)`.

If an application needs only one rendering thread, MT-safe mode is not recommended. MT-safe mode incurs some performance overhead which can be avoided for single threaded rendering. Some multithread cases may contain computation or GUI threads. For these cases an application can create one OpenGL rendering thread and separate GUI or computational threads.

Multithread safe allows OpenGL parallelism. This parallelism supports single to multiple processors as well as single to multiple screens.

Multithread safe does not allow:

- More than one thread using the same context
- More than one current context per thread

The maximum number of OpenGL rendering threads supported is 64.

The following MT-Safe patches are required:

- Solaris 2.5.1 +patch 103566-27
- Solaris 2.6 +patch 105633-02 (if using Creator patch 105360-04 also needed)

---

**Caution –** When the OpenGL renderer (see `glGetString(GL_RENDERER)`) is a graphics accelerator (not a software renderer), multiple rendering threads to the same screen might perform slower than single threaded rendering. If possible, avoid multithreaded rendering to a single graphics accelerated screen.

---

# Supported Platforms

The OpenGL 1.1.1 for Solaris software supports the following devices:

- Creator Graphics and Creator3D Graphics – OpenGL functionality is accelerated in hardware.
- SX, ZX, GX, GX+, TGX, TGX+, S24 – OpenGL functionality is performed in software.
- All SMCC SPARC™ systems equipped with the following frame buffers are supported on the OpenGL 1.1.1 for Solaris software: the TCX, SX, GX, ZX and Creator families of frame buffers. This includes Ultra™ desktop, Ultra Enterprise™ and all the legacy SPARCstation™ family.

---

**Note –** The PGX frame buffer family is not supported by the OpenGL 1.1.1 for Solaris software.

---

# Where to Look for Information on OpenGL Programming

For information on how to write an OpenGL application, see the following books:

- *OpenGL Programming Guide* by Neider, Davis, and Woo
- *OpenGL Reference Manual* by the OpenGL Architecture Review Board
- *OpenGL Programming for X Windows System* by Mark Kilgard

These books are published by Addison-Wesley and are available through your local bookstore.

For more information on OpenGL, you may want to refer to "The Design of the OpenGL Interface" written by Mark Segal and Kurt Akeley. A PostScript copy of this document is included in the `SUNWgldoc` package or the OpenGL 1.1.1 for Solaris CD-ROM. For the complete specification of what constitutes OpenGL, see *The OpenGL Graphics System: A Specification, Version 1.1*, also written by Mark Segal and Kurt Akeley. An online version of this specification is located at `http://www.sgi.com/Technology/OpenGL/glspec1.1/glspec.html`).

Finally, for a good source of answers to questions you may have about OpenGL, see the OpenGL information center at `http://www.sgi.com/Technology/OpenGL/opengl.html`.

CHAPTER 2

# OpenGL for Solaris Architecture

The purpose of designing a graphics system architecture is to enable performance within the constraints of cost and functionality goals. Hardware design places various stages of the graphics pipeline into hardware accelerators. Software design uses the hardware features and complements the hardware by providing complete coverage of functionality.

Understanding the hardware and software architecture of a particular system will help you determine whether a feature is accelerated in the graphics hardware or implemented in software. This will enable you to identify which path through the system your application uses for the feature. With this information, you can project your application's performance. Given knowledge of performance versus functionality tradeoffs, you can make informed choices about how to use the system to maximize your application's interactivity.

This chapter describes the OpenGL for Solaris architecture. First it defines two terms commonly used when discussing hardware and software performance.

## Acceleration vs. Optimization

When discussing performance, understanding how the hardware implementor, software implementor, and application programmer define and differentiate the terms *hardware acceleration* and *software optimization* is helpful.

- To the hardware designer, hardware accelerating OpenGL means implementing logic in the form of gates and data paths for OpenGL functions.
- To the OpenGL software implementor, accelerating OpenGL functions means writing software to use the graphics hardware features. In addition, the software implementor can *optimize* OpenGL features that are not accelerated in hardware by writing highly tuned code to make the performance of those features as efficient as possible.

- To the OpenGL application programmer, acceleration typically means the speed at which various combinations of geometry and OpenGL state render, with the goal generally being interactive performance.

With these definitions in mind, the next sections describe the OpenGL architecture and the implementation of this architecture in the Solaris OpenGL software.

# A Quick Review of the OpenGL Architecture

As a first step in examining the OpenGL for Solaris architecture, Figure 2-1 shows the basic architecture of the OpenGL library.

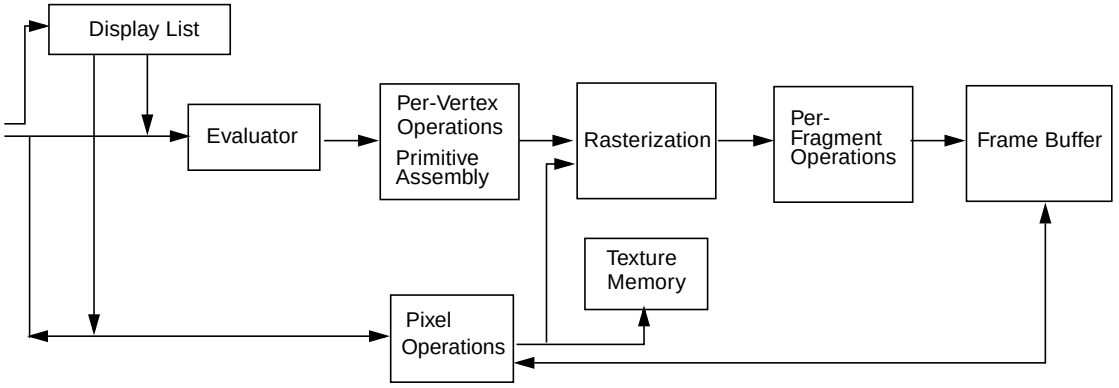


**FIGURE 2-1** OpenGL Architecture[1]

In the first stage of the OpenGL pipeline, vertex data enters the pipeline, and curve and surface geometry is evaluated. Next, colors, normals, and texture coordinates are associated with vertices, and vertices are transformed and lit. Vertices are then assembled into geometric primitives.

The rasterization stage converts geometric primitives into frame buffer addresses and values, or fragments. Each fragment may be altered by per-fragment operations, such as blending. Per-fragment operations store updates into the frame buffer based on incoming and previously stored Z values (for Z buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations.

1. From Segal, Mark, and Kurt Akeley, "The OpenGL Graphics System: A Specification," Mountain View, CA, 1995.

Pixel data is processed in the pixel operation stage. The resulting data is stored as texture memory, or rasterized and processed as fragments before being written to the frame buffer.

The task of the hardware and software implementors at Sun was to implement the OpenGL functionality. The remainder of this chapter describes this implementation.

# Graphics Hardware Architecture

Graphics hardware architectures can be designed to meet varying constraints of cost and CPU performance. High-performance model coordinate (MC) devices typically implement vertex processing and transformations in hardware. A model coordinate device may perform lighting, coordinate transformations, clipping, and culling as well as rasterization and fragment processing in hardware, thereby providing very fast performance.

At a different performance level, rasterization devices typically use the host CPU to perform vertex processing and use the rasterization hardware to convert device coordinate geometry into pixel values. The Ultra Creator and Creator3D systems are examples of device coordinate (DC) devices. The graphics hardware architecture of the Creator3D graphics system is designed as follows:

- Primitive assembly and vertex processing are performed on the UltraSPARC™ CPU. Texturing operations are also performed on the CPU.
- Rasterization and fragment processing are performed in the Creator3D Graphics hardware subsystem. The Creator3D graphics system accelerates rasterization of lines, points, and triangles, and also accelerates per-fragment operations such as the pixel ownership test, scissor test, depth buffer test, blending, logical operations, line anti-aliasing, line stippling, and polygon stippling.

The benefit of building custom hardware for graphics is that when operations are parallelized in hardware circuits, turning on features (like both Z-buffering and blending) has a very small performance cost. If a feature is provided in hardware, the hardware is usually designed to allow sustained throughput for that feature. Thus, you can make full use of features that have been implemented in hardware without experiencing performance degradation.

The benefit of putting graphics functions in software is that since the CPU is a required and shared computing resource, using it for graphics operations imposes no additional financial cost. The disadvantage is that each additional graphics operation requires CPU cycle time. When an application asks more of the CPU, the CPU may perform more slowly.

# Solaris OpenGL Software Architecture

Once the hardware designers have determined what the hardware will accelerate, all other decisions regarding performance fall to the software implementors. Software implementors need to consider the following questions:

What hardware features will be used?

1. What features that are not accelerated in hardware can the software optimize?
2. How will the software implement all functionality?

In response to these questions, the Solaris OpenGL software developers implemented OpenGL as follows:

- Accelerated OpenGL by using using all features of the Creator and Creator3D graphics subsystems.
- For the Creator and Creator3D systems optimized line and point transformation and clip test, and a subset of texture lookup and filtering.
- Implemented OpenGL to its complete specification by writing code for primitive assembly and vertex processing, including:
  - Coordinate transformations
  - Texture coordinate generation
  - Clipping
- Implemented two forms of software rasterization for OpenGL features not rasterized in hardware:
  - Optimized software rasterizer for many texturing functions and pixel operations. Software rasterization is done by the CPU using an optimized implementation. On an UltraSPARC CPU, some features, such as texturing rasterization, may be handled using software code employing the VIS instruction set.
  - A software rasterizer for all features not handled by the hardware or by the VIS software.

This implementation of the OpenGL for Solaris library allows devices with varying capabilities to run efficiently on the OpenGL software. It enables OpenGL for Solaris applications to run on the following types of devices:

- Model coordinate device – Handles most OpenGL functionality in hardware, including vertex processing, primitive assembly, rasterization, and fragment operations.
- Device coordinate device (Creator or Creator3D graphics system) – Performs vertex processing. Rasterization and fragment processing is handled in hardware.

- Memory mappable devices (SX, ZX, GX, GX+, TGX, TGX+, TCX) – Vertex processing, primitive assembly, rasterization, and fragment processing are performed in software, and the results are written to the memory-mapped frame buffer.

FIGURE 2-2 on page 12 illustrates the graphics software architecture of the OpenGL for Solaris product. This figure shows the paths that application data can take through the OpenGL system, depending on the type of hardware device the application is running on. TABLE 2-1 summarizes the data paths with reference to several hardware platforms.

**TABLE 2-1** Data Paths Through the OpenGL for Solaris System

| Platform | Vertex Processing | Rasterization | Performance |
|---|---|---|---|
| MC device | Hardware vertex processing | Hardware rasterizer | Fastest path |
| | Software vertex processing | Hardware rasterizer | Fast path |
| | Software vertex processing | Software rasterizer | Slow path |
| DC device (Creator3D or Creator) | Software vertex processing | Hardware rasterizer | Fast path |
| | Software vertex processing | Software rasterizer | Slow path |
| Memory map (ZX, GX, SX) | Software vertex processing | Software rasterizer | Only path |

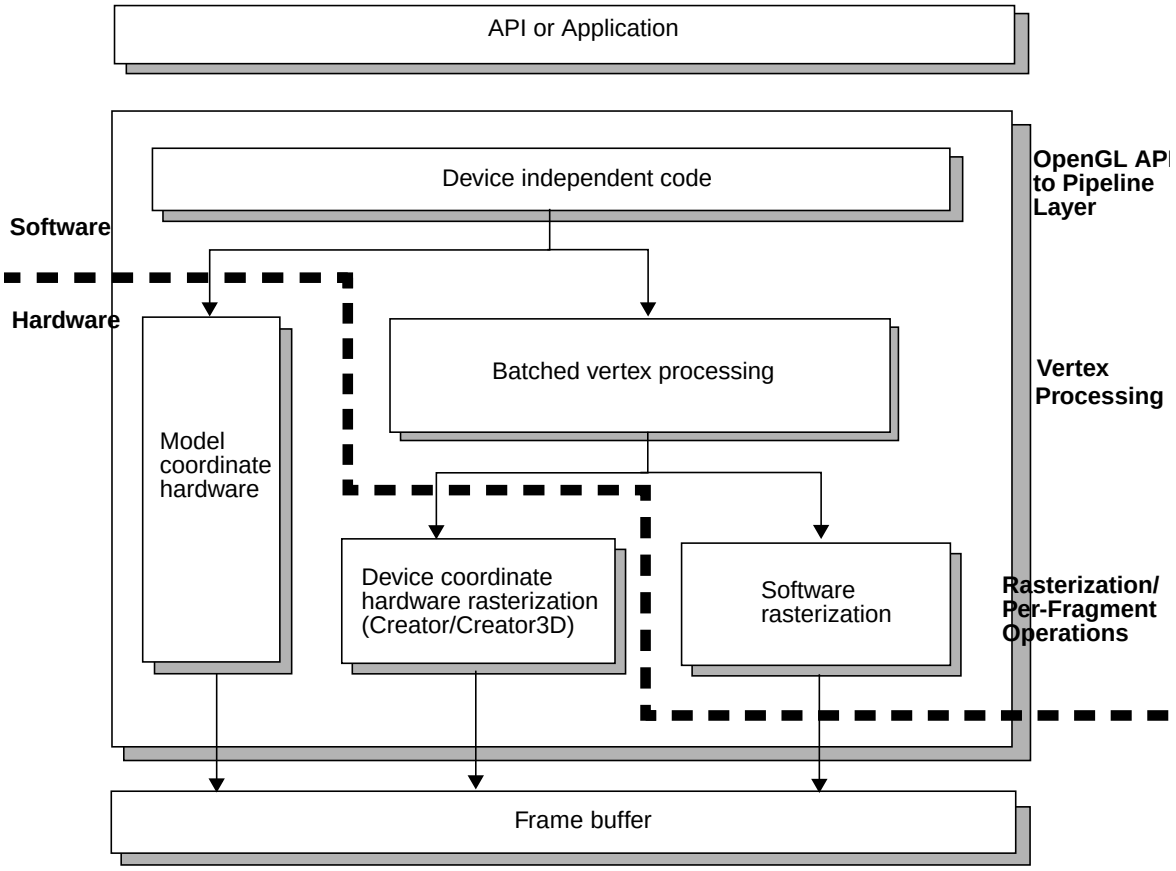


**FIGURE 2-2** Solaris OpenGL Software Architecture

# Vertex Processing Architecture

As Figure 2-2 shows, Sun's OpenGL implementation handles vertex processing in several ways:

- Hardware vertex processing – On model coordinate devices, vertex processing is done via the hardware. In addition to hardware acceleration, the model coordinate (MC) pipeline is optimized for vertex arrays and display list mode. The model coordinate pipeline also recognizes consistent data types within `glBegin`/`glEnd` pairs. If the data is consistent, the software is able to use hardware resources efficiently.

- Software vertex processor – This is the fully optimized path from the software implementor's point of view. The principal optimization is that the model coordinate software pipeline recognizes consistent data types within `glBegin`/`glEnd` pairs: if the data is consistent, the software pipeline is able to use CPU resources efficiently.

The OpenGL vertex array commands result in the best performance for vertex processing on all hardware platforms. For repeated rendering of the same geometry, display lists provide significant performance benefits over immediate mode rendering.

## Rasterization and Fragment Processing Architecture

Rasterization and fragment processing is handled in one of the following ways:

- Hardware rasterizer – The graphics subsystem handles lines, points, and triangles, and does simple fragment processing, such as blending and the depth-buffer test.
- Optimized software rasterizer – The CPU does software rasterization using an optimized implementation. On an UltraSPARC CPU, some features, such as texturing rasterization, may be handled by the UltraSPARC CPU using software code employing the VIS instruction set.
- Software rasterizer – The CPU does software rasterization using a generic, unoptimized implementation. The generic software rasterizer is approximately one-sixth the speed of the optimized software rasterizer.

## Solaris OpenGL Interface Layers

The OpenGL for Solaris implementation has three layers of interfaces with the hardware, each requiring successively more processing by the host CPU. These interface layers correspond to the stages of the OpenGL pipeline. The rendering interface is determined by the value of the current OpenGL attributes, and in a small number of cases by the geometry itself. In general, the more host processing needed, the slower the resulting rendering, so an application should avoid attributes that force the slower rendering layers to be used.

FIGURE 2-3 on page 15 shows the interface layers and their relationship to data paths through the OpenGL for Solaris system. In this illustration, the filled boxes represent the hardware-specific device pipeline (DP) components and show the hardware data paths. The white boxes represent the device-independent (DI) software components and show the software data paths.

The more efficiently an application can reach a filled box, the better the application's performance will be. For example, for an application running on a model coordinate device, the fast data paths are those that result in rendering in hardware at the vertex processing layer. Setting an attribute that causes the use of the software pipeline for model coordinate processing can result in a significant drop in performance. Setting an attribute that results in the use of software rasterizing can cause an even more significant drop in performance.

On a device coordinate device such as the Creator3D system, hardware rasterization is about three times faster than the VIS (optimized) rasterizer. The VIS rasterizer is about five-to-six times faster than the generic software rasterizer. Thus, the best way to increase rasterization and fragment processing performance on a DC device is to stay in the hardware rasterizer whenever possible.

Memory-mappable devices without hardware support use the software pipeline for model coordinate operations and the software rasterizer for rasterization. Examples of this device are the single-buffered GX, and TGX. For devices that do not allow memory access, the OpenGL for Solaris architecture provides a pixel--rendering interface layer. However, at this time no Sun hardware devices use this interface layer.

For detailed information on attributes that result in slower rendering paths, see **Chapter 3 "Performance**."

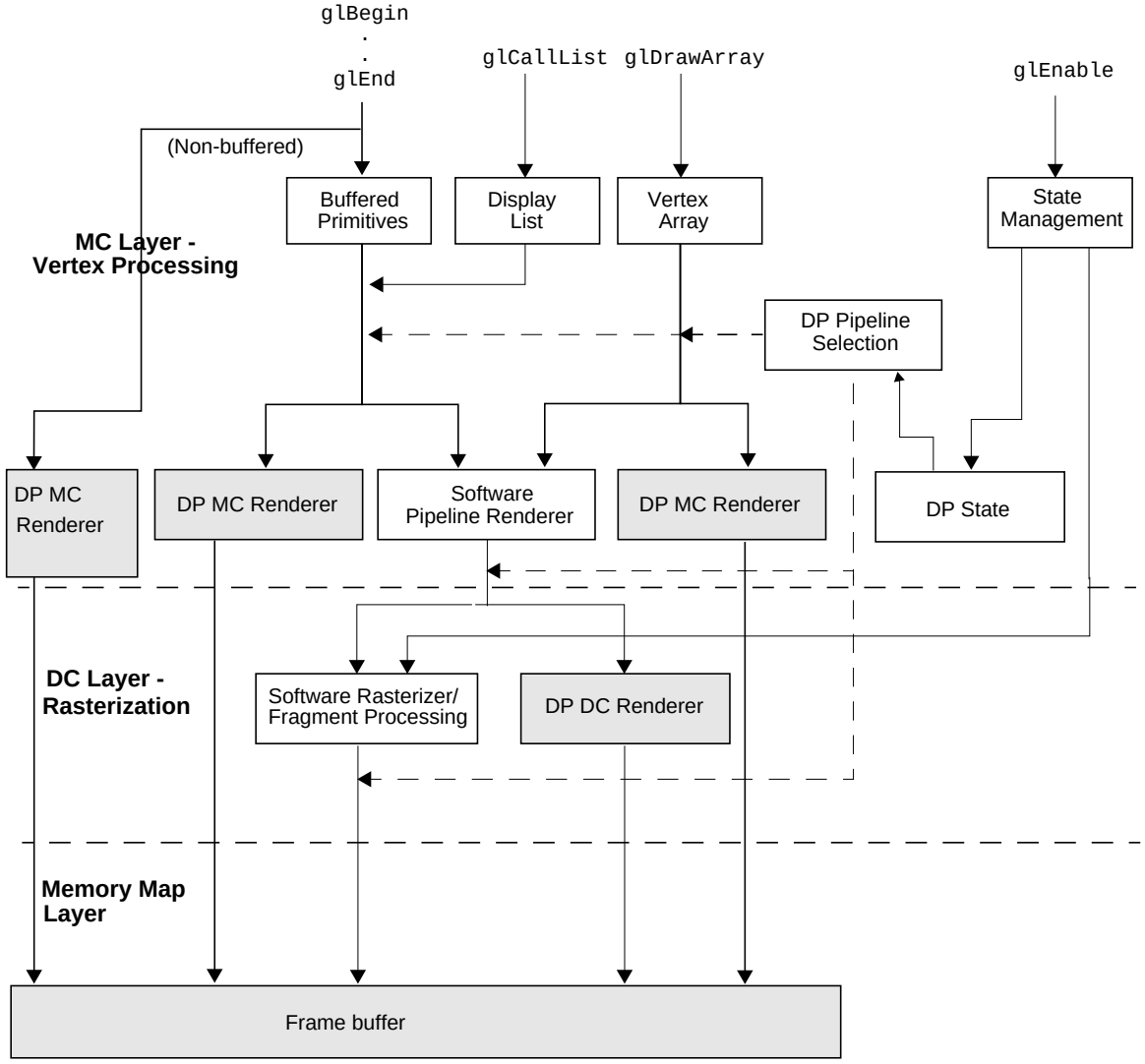


**FIGURE 2-3** Solaris OpenGL Data Paths

CHAPTER 3

# Performance

This chapter provides performance information that you can use to tune your application to make the best use of Sun hardware graphics accelerators. The first section provides general advice on how to optimize vertex processing performance for a variety of platforms. The subsequent sections provide specific techniques to ensure maximum performance on the Creator3D and Creator graphics accelerators.

## General Tips on Vertex Processing

To achieve the best vertex processing performance on all Sun platforms, follow these guidelines:

1. Use vertex arrays or display list mode rather than immediate mode whenever rendering data repeatedly.
2. Use consistent patterns of data types between `glBegin`(3gl) and `glEnd`(3gl). Consistent data types are described in “Consistent Data Types” on page 18.
3. If you must use immediate mode, try to include as many primitives of the same type as possible between one `glBegin` and the corresponding `glEnd`.
4. If vertex array is used, try to stay in vertex array mode, rather than switching between vertex array and immediate mode.

These guidelines are discussed in the sections that follow.

# Vertex Arrays

Vertex array commands provide the best performance for vertex processing of big primitives because they avoid the function call overhead of passing one vertex, color, and normal at a time. Instead of calling an OpenGL command for each vertex, you can pre-specify arrays of vertices, colors, and normals, and use them to define a primitive or set of primitives of the same type with a single command. Interleaved vertex arrays may enable even faster performance, since the application passes the data packed in a single array.

## MultiDrawArrays

OpenGL for Solaris contains the extension `glMultiDrawArraysSUN`(). This function allows multiple strips of primitives to be rendered with one call to OpenGL. Because of reduced function call and setup overhead, this function can provide significant spee when an object contains many short strips. For some implementations of this function, there may be additional performance gains if the strips are contiguous in the vertex array. As with the standard `glDrawArrays`(), using interleaved vertex arrays gives even better performance.

# Consistent Data Types

For the OpenGL for Solaris implementation on all Sun platforms, vertex processing is optimized if the application provides consistent, supported data types within a `glBegin`/`glEnd` pair. Data types are consistent when the commands between one vertex call, such as `glVertex3fv`, and the next vertex call include identical patterns of data types in the identical order. In other words, consistent data is data for which the pattern is the same for each vertex, except when `glCallList` or `glEval*` is included. For example, the following set of commands is consistent because the primitive is defined by the repeating set of calls `glColor3fv`(3gl); `glVertex3fv`(3gl).

```
glBegin(GL_LINES);
       glColor3fv(...);
       glVertex3fv(...);
       glColor3fv(...);
       glVertex3fv(...);
       glColor3fv(...);
       glVertex3fv(...);
glEnd();
```

As another example, the following set of commands is consistent since each vertex contains the same data- a color, normal, and vertex in repeating order.

```
glBegin(GL_LINES);
       glColor3f(...);
       glNormal3f(...);
       glVertex3f(...);
       glColor3f(...);
       glNormal3f(...);
       glVertex3f(...);
glEnd();
```

---

**Note –** The `*f` versions of the calls may be used interchangeably with the `*fv` versions.

---

Inconsistent data types do not follow a repeating, supported pattern. In the first example below, the data is inconsistent because the first vertex has a normal, but the second vertex doesn't. In the second example, the order is reversed in the second set of commands, although both vertices have a color and a normal.

```
glBegin(GL_LINES);
       glNormal3fv(...);
       glColor3fv(...);
       glVertex3fv(...);
       glColor3fv(...);
       glVertex3fv(...);
glEnd();

glBegin(GL_LINES);
       glColor3fv(...);
       glNormal3fv(...);
       glVertex3fv(...);
       glNormal3fv(...);
       glColor3fv(...);
       glVertex3fv(...);
glEnd();
```

For general information on the vertex data that can be specified between `glBegin`(3gl) and `glEnd`(3gl) calls, see the `glBegin`(3gl) reference page.

## Low Batching

OpenGL for Solaris performs best when given big primitives. If small primitives are sent to the library, the library will try to batch these primitives together, providing that the primitives are of the same primitive type, with the same consistent data pattern, and there are no attribute state changes outside the `glBegin` call.

For example, the following primitives will be batched together by the library.

```
glBegin(GL_TRIANGLES);
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
glEnd();

glBegin(GL_TRIANGLES);
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
   glNormal3fv(...);
   glVertex3fv(...);
glEnd();
```

The following example shows that the primitives are not batched together because the `glColor3fv` call outside the `glBegin` call breaks the batching of the primitives.

```
glBegin(GL_LINES);
   glVertex3fv(...);
   glVertex3fv(...);
glEnd();
```

```
glColorfv(...);
glBegin(GL_LINES);
  glVertex3fv(...);
  glVertex3fv(...);
glEnd();
```

## Optimized Data Types

On any platform that uses the software pipeline for model coordinate rendering, your application will get better performance if it can pass vertex data in patterns for which the software pipeline has optimized code. Optimized data patterns are consistent data patterns which contain none of the following:

- glEdgeFlag*()
- glMaterial*()
- glEvalCoord*()
- glCallList() or glCallLists()
- both glColor*() and glIndex*()
- both glTexCoord*() and glIndex*()

# Creator3D Graphics and Creator Graphics Performance

The Ultra Creator and Creator 3D Graphics systems accelerate rasterization of lines, points, and triangles as well as most per-fragment operations. Vertex processing and texturing operations are performed on the UltraSPARC CPU. The OpenGL for Solaris implementation for the Creator and Creator3D frame buffers uses all features of the Creator graphics subsystem.

Rasterization and fragment processing is handled in one of three ways:

- Creator3D hardware rasterizer – Handles lines, points, and triangles, and does simple fragment processing.
- Optimized software rasterizer – UltraSPARC VIS (Visual Instruction Set) handles many texturing functions and pixel operations.
- Generic software rasterizer – Performs rasterization for all features not handled by the hardware or by the VIS software.

To find out more about the Creator and Creator3D hardware platforms, refer to the Architecture Technical White paper at `http://www.sun.com/desktop/products/Ultra2/`.

The following sections provide specific information on attribute use and pixel operations on these platforms.

# Attributes Affecting Creator3D Performance

Primitive-attribute settings affect performance; therefore, you will get a better level of performance if you can avoid setting the attributes listed below. In some cases, the listed attributes simply increase the amount of processing in the hardware or optimized software data paths. In other cases, setting these attributes forces the use of the software rasterizer, resulting in slow performance.

## Attributes That Increase Vertex Processing Overhead

Attributes that that result in more vertex processing overhead include:

- Enabling lighting.
- Turning on user specified clip planes (`GL_CLIP_PLANE[i]`).
- Enabling color material (`GL_COLOR_MATERIAL`).
- Enabling non-linear fog (`glFog(GL_FOG_MODE, GL_EXP{2})`). An exception to this is using `RGBA` mode on Creator3D Series 2.
- Enabling `GL_NORMALIZE`.
- Turning on polygon offset. However, polygon offset is optimized for the case when the factor parameter of the `glPolygonOffset` call is set to 0.0. Users may have to adjust the units parameter accordingly to avoid stitching for this case.

## Primitive Types and Vertex Data Patterns That Increase Vertex Processing Overhead

Types and patterns that result in more vertex processing overhead are:

- Using a surface primitive type as an argument to `glBegin`. The surface primitive types are: `GL_TRIANGLES`, `GL_TRIANGLE_STRP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP` and `GL_POLYGON`.

- Using a vertex data pattern for `GL_POINTS, GL_LINES, GL_LINE_STRIP,` and `GL_LINE_LOOP`, *other than* one of the following repeating patterns. These are the patterns that are maximally accelerated.

  V3F:

  glVertex3f(...);

  ...

  C3F_V3F:

  glColor3f(...);

  glVertex3f(...);

  ...

  C4F_V3F:

  glColor4f(...);

  glVertex3f(...);

  ...

  V2F:

  glVertex2f(...);

  ...

  C3F_V2F:

  glColor3f(...);

  glVertex2f(...);

  ...

  C4F_V2F:

  glColor4f(...);

  glVertex2f(...);

  ...

---

**Note –** All vertex data patterns, other than one of the above repeating patterns, take more memory.

---

- Using `glDrawElements` in immediate mode.

## Attributes That Increase Hardware Rasterization Overhead

Attributes that result in slower hardware rasterization are:

- Enabling line antialiasing (`GL_LINE_SMOOTH`)
- Enabling point antialiasing (`GL_POINT_SMOOTH`)

## Environment Variables Affecting Read Performance

- `unsetenv SUN_OGL_ABGR_READPIX_NOCONFORM` (default)

  The alpha value read back from the frame buffer during `glReadPixels` with the `GL_ABGR_EXT` format is always 1.0. This is conformant but slower than the following variable.

- `setenv SUN_OGL_ABGR_READPIX_NOCONFORM`

  The alpha value read back from the frame buffer during `glReadPixels` with the `GL_ABGR_EXT` format is undefined. This is up to 30% faster than the conformant version. For Creator, the alpha value is not stored in the frame buffer anyway. Consequently, if the application does not use the alpha value, then this version is a significantly faster way to read pixels back from the frame buffer.

## Attributes That Force the Use of the Software Rasterizer

Setting the following attributes forces the use of the software rasterizer. This is the slowest data path. If your application requires any of the following attributes for performance critical functionality, you may want to determine whether this performance is acceptible. If not, you can evaluate whether the use of these attributes is advisable.

1. Rasterization attributes
   - In Indexed color mode, enabling line anti-aliasing (`GL_LINE_SMOOTH`) or point anti-aliasing (`GL_POINT_SMOOTH`)
   - Enabling polygon anti-aliasing (`GL_POLYGON_SMOOTH`)
   - Stippled lines (`GL_LINE_STIPPLE`) where the line stipple scale factor is larger than 15
   - Non-antialiased (“jaggy”) points with `glPointSize(3gl)`greater than 1.0

---

**Note –** The only anti-aliased point size supported by Creator3D and Creator is 1.0. `glPointSize` is ignored for anti-aliased points. Although the nominal antialiased point size is 1.0, the actual visible size is approximately 1.5.

---

2. Fragment Attributes

   - Blending (`GL_BLEND`) forces the use of the software rasterizer unless both the source and destination blend functions are in the following set of blend functions supported by the hardware:
     `GL_ZERO`
     `GL_ONE`
     `GL_SRC_ALPHA`
     `GL_ONE_MINUS_SRC_ALPHA`

   - Enabling the stencil test (`GL_STENCIL_TEST`) on Creator3D or Creator3D Series 2. (Enabling the stencil test does not force the use of the software rasterizer on Creator3D Series 3 because it supports hardware stencilling).

     On the UltraSPARC platform, a VIS optimized software rasterizer is used for smooth-shaded non-textured stenciled triangles whenever the `glStencilOp` parameter *fail* is anything other than `GL_INCR` or `GL_DECR` and the depth test does not affect the stencil buffer. (This is the case when depth test is disabled or the `glStencilOp` parameters *zfail* and *zpass* are identical).

   - Enabling any type of fog in Indexed color mode

FIGURE 3-1 shows the data path for hardware rasterization on the Creator3D system. FIGURE 3-5 on page 34 illustrates the data path that the application uses when it sets an attribute that forces the use of the software rasterizer.

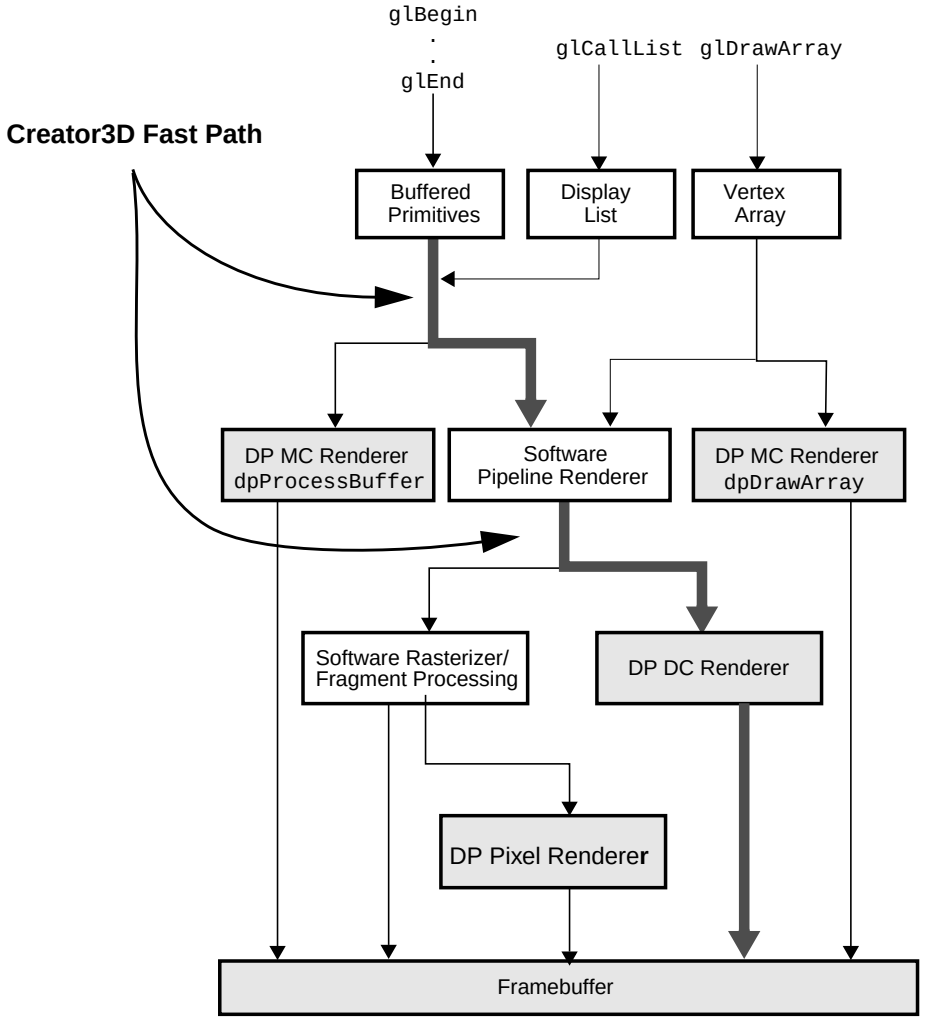


**FIGURE 3-1** Hardware Rasterizer Path for Creator3D

3. Texturing Attributes

- Color Table—When the `GL_TEXTURE_COLOR_TABLE_SGI` extension is used, the only `glTexEnv` texture base internal formats that are accelerated are `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA` and `GL_INTENSITY`.
- The texture environment mode `glTexEnv GL_TEXTURE_ENV_MODE` of `GL_BLEND` is not accelerated when it is used with the `GL_TEXTURE_COLOR_TABLE_SGI` extension.

- Fog—On Creator3D, only linear fog is accelerated. On Creator3D Series 2, all types of RGBA fog are accelerated.

## Attributes That Vary Optimized Texturing Speed

Texturing  makes extensive use of VIS on UltraSparc platforms and allows for large textures. Texturing speed naturally increases with faster CPUs (a 300 Mhz UltraSparc CPU is 1.6 times faster than a 167 Mhz CPU). Though texturing fill rates are slower on a host CPU than on dedicated hardware, the system costs are lower.

The extensions supported for texturing include 3D Texture Mapping, SGI Color Table, and SGI Texture Color Table.

Stencil and some fragment blending cases are slow. The rest are fast (done by Creator 3D hardware).

Some texturing attributes are handled by generic code and result in the slowest texturing speed when the `GL_TEXTURE_COLOR_TABLE_SGI` extension is used with texture environment color blending or base internal formats of `GL_ALPHA`, `GL_RGB`, or `GL_RGBA`.

Texturing attributes with the most impact on speed are:

- Minification filter
- Texture Coordinate Interior/Exterior Classification (per triangle)
- All wrap modes set to `GL_REPEAT`
- Texture Color Lookup Table

The VIS optimized software rasterizer will vary in texturing speed based on the texturing attributes specified. The factors affecting texturing speed are listed below. Note that this is *variance within the optimized path*, not the difference between the optimized and generic paths.

- Projection Type—The type of projection matrix. Orthographic is faster than perspective.
- Wrap Mode—Best speed is when all dimensions (`GL_TEXTURE_WRAP_`*x*) are set to `GL_REPEAT`. If all the texture wrap modes are `GL_REPEAT`, this case is specially optimized. If any of the texture wrap modes are `GL_CLAMP`, then the standard texture wrap routine is used, but it is slower than the special case.
- Dimension—In general, 2D texturing is faster than 3D texturing, since there is one less texture coordinate to deal with. However, this does not mean it is better to use many 2D textures to approximate 3D texturing since the texture load time (see next section) may significantly increase the overhead.

- Minfilter—The fastest GL_TEXTURE_MIN_FILTER parameter is GL_NEAREST, which is approximately 4x the speed of GL_LINEAR. See FIGURE 3-3 on page 31 and FIGURE 3-4 on page 32. The approximate relative speed in decreasing order is: GL_NEAREST, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR, GL_LINEAR_MIPMAP_NEAREST, and GL_LINEAR_MIPMAP_LINEAR.
- Magfilter—For GL_TEXTURE_MAG_FILTER, the same speed ratio of 4x applies to GL_NEAREST vs. GL_LINEAR. Note, however, that GL_TEXTURE_MAG_FILTER is ignored when GL_TEXTURE_MIN_FILTER is set to GL_NEAREST or GL_LINEAR. This can be overridden with a shell environment variable but this will slow down texturing speed for GL_NEAREST and GL_LINEAR, since they now have to perform level-of-detail calculations to determine when to use GL_TEXTURE_MAG_FILTER. The shell environment variable that forces this slower behavior is:
- `setenv SUN_OGL_MAGFILTER "conformant"`
- Texture Coordinate Classification—If all texture coordinates of a triangle/quad/polygon are at LEAST 1/2 texel inside away from the texture map edge, then the primitive is considered interior and are render faster than those whose texture coordinates touch or cross the texture map's edges. If any vertex touches or crosses the texture map edge, then the primitive is considered exterior. If a primitive is interior, then the texture edge related attributes such as wrap modes and texture border no longer affect the texturing speed.
- Env Mode—The fastest glTexEnv() GL_TEXTURE_ENV_MODE is GL_REPLACE, followed closely by GL_MODULATE. GL_DECAL is the same speed as GL_REPLACE.
- Color Table—The use of the extension GL_TEXTURE_COLOR_TABLE_SGI will reduce texturing speed.
- Texture Color Lookup Table—Using this table causes significant slowdown of texturing speed. Only cases of one or two channel lookups are optimized - GL_LUMINANCE, GL_INTENSITY, GL_LUMINANCE_ALPHA. Three or four channel lookups (GL_RGB, GL_RGBA) go to a generic code routine that is slower than the special case.

## Attributes That Vary Texture Load Time

The time to load the texture image into a texture object or a display list will vary depending on the pixel store and pixel transfer attributes specified when the texture is specified.

FIGURE 3-2 shows the texture load processing flow.

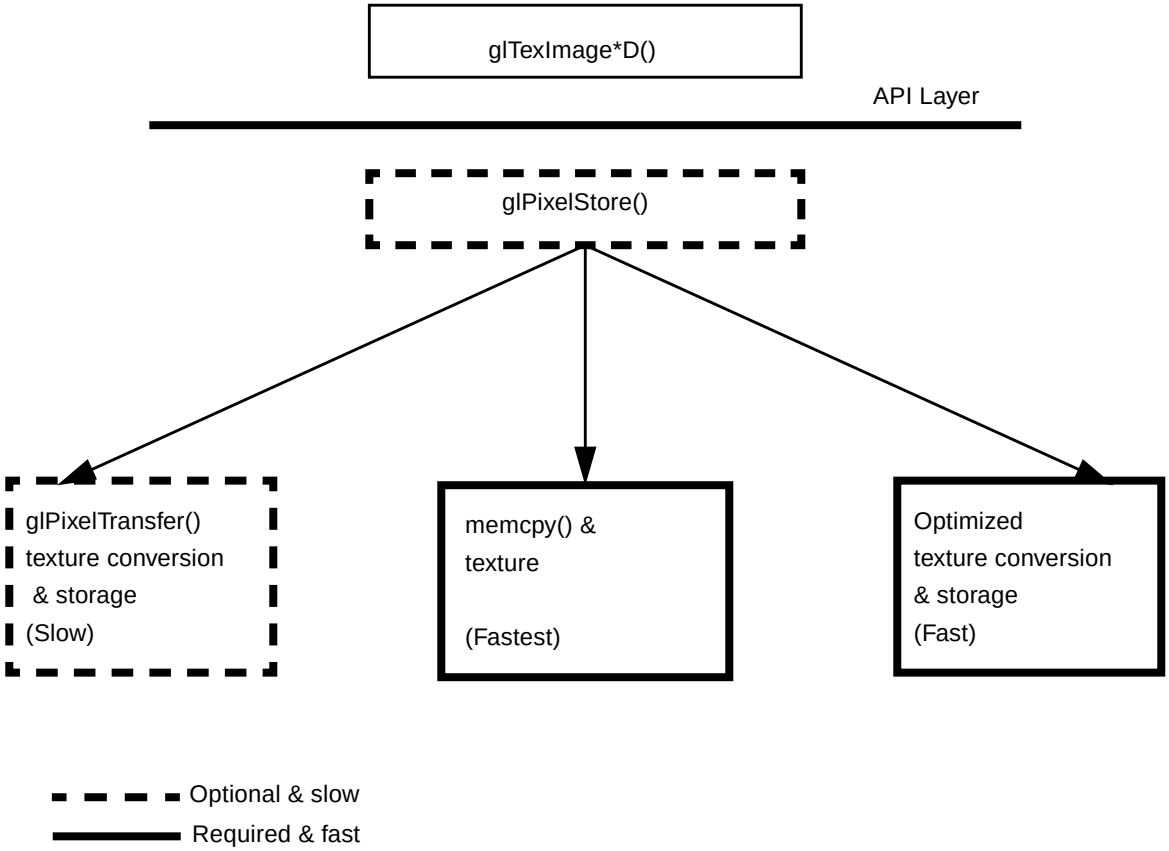


**FIGURE 3-2** Text Load Processing Flow

The following recommendations should be followed where possible to reduce texture load time:

- Use texture objects where possible.
- If multiple textures are being used, put the textures in texture objects and use `glBindTexture` to switch among the textures. This ensures that the internal copy of texture is evaluated only once.
- For faster load time of 1D and 2 D textures, use `GL_ABGR_EXT` format of data type `GL_UNSIGNED_BYTE` and texture internal format of `GL_RGBA`.
- 3D textures use packed representation to minimize memory usage.

- For 3D textures using data type GL_UNSIGNED_BYTE, the following format/base internal format combinations give the best loading performance:

**TABLE 3-1** 3D optimized cases

| Format | Base Internal Format |
|---|---|
| GL_LUMINANCE_ALPHA | GL_LUMINANCE_ALPHA |
| GL_RED | GL_INTENSITY |
| GL_RED | GL_LUMINANCE |
| GL_ALPHA | GL_ALPHA |
| GL_LUMINANCE | GL_INTENSITY |
| GL_LUMINANCE | GL_LUMINANCE |
| GL_ABGR_EXT | GL_RGBA |

## *Relative Performance of Attributes*

The following two charts show the relative performance of the attributes. The Y-axis is a ratio of the measured texturing speed against the fastest texturing case speed (which is 2D ortho nearest replace interior). Since all the charts were computed using the one number as a divisor, individual bars can be compared across charts. For example, the relative performance of 2D vs 3D texturing can be seen by comparing the bars between the 2D and 3D charts.

The meanings of the legend annotations in the charts are:

ortho—Orthographic Projection

persp—Perspective Projection

repeat—All wrap modes set to GL_REPEAT

clamp—Some wrap modes set to GL_CLAMP

intr—All texture coordinates are interior

extr—All texture coordinates are exterior

ctab—Texture Color Lookup Table extension (GL_TEXTURE_COLOR_TABLE_SGI) is enabled

nearest—Texture minification filter is GL_NEAREST

nmn—Texture minification filter is GL_NEAREST_MIPMAP_NEAREST

nml—Texture minification filter is GL_NEAREST_MIPMAP_LINEAR

linear—Texture minification filter is GL_LINEAR

lmn—Texture minification filter is GL_LINEAR_MIPMAP_NEAREST

lml—Texture minification filter is GL_LINEAR_MIPMAP_LINEAR

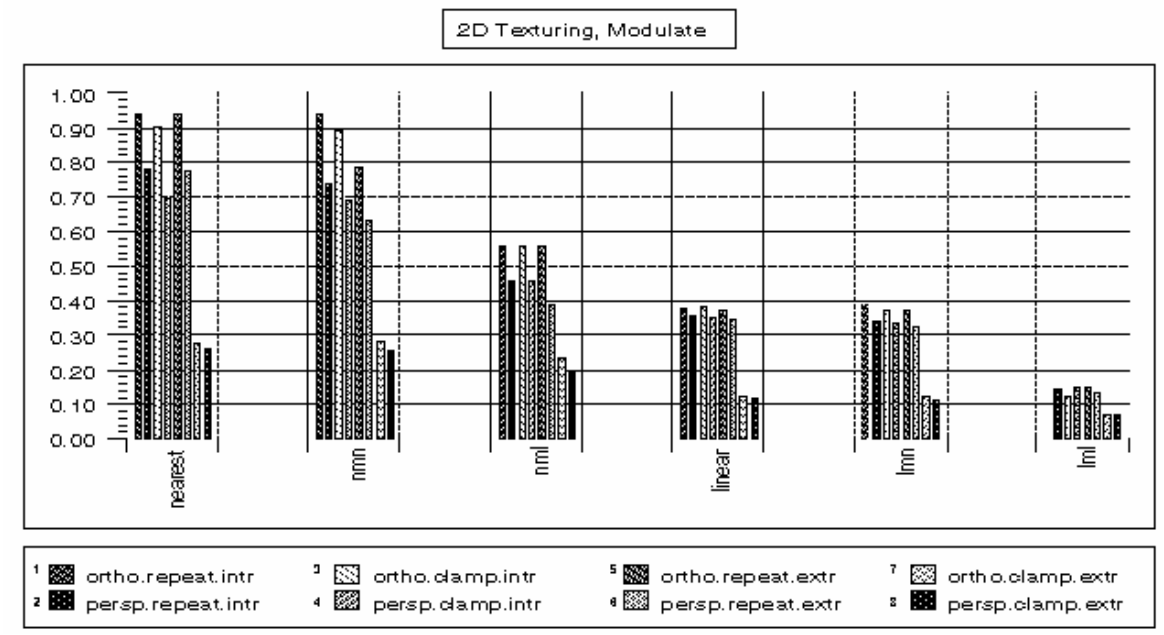


**FIGURE 3-3** 2D Texturing

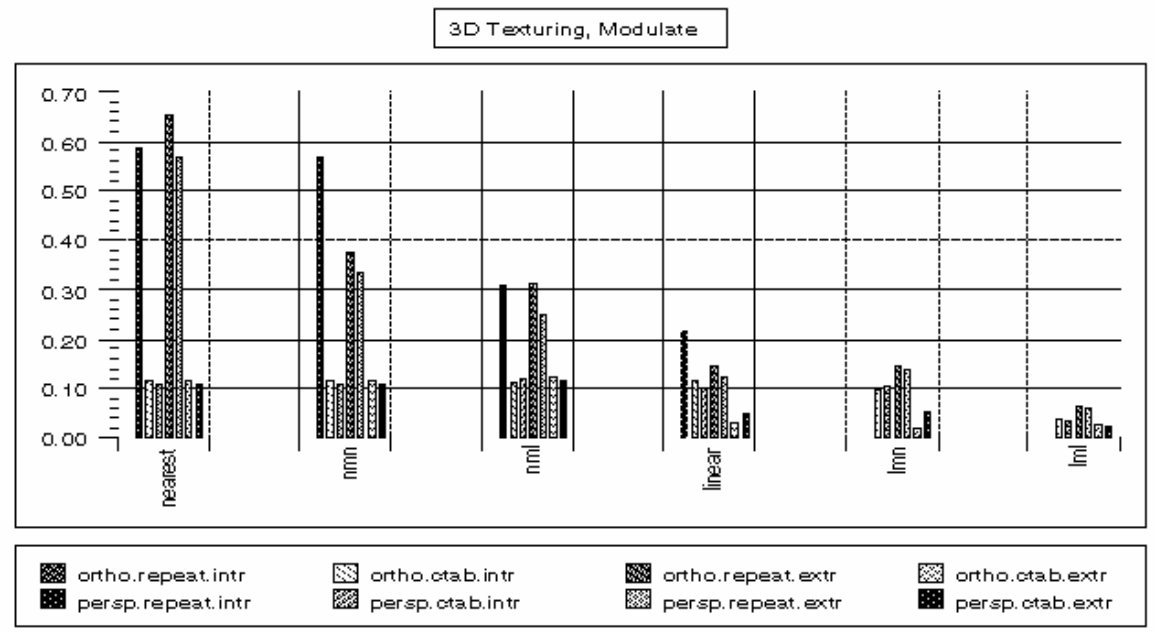


**FIGURE 3-4** 3D Texturing

# Attributes Affecting Creator Performance

This section applies when pure software rendering is being used. This happens on the single-buffered Creator platform when `glDrawBuffer`(3gl) is set to `GL_BACK` or `GL_FRONT_AND_BACK`. The data presented here is also valid for the SX, ZX, GX, GX+, TGX, TGX+, and TCX platforms. Note that for non-Ultra machines, VIS rasterization is replaced by an optimized software rasterizer.

## Attributes That Increase Vertex Processing Overhead

Attributes that result in more vertex processing overhead are:

- Enabling lighting.
- Turning on user specified clip planes (`GL_CLIP_PLANE[i]`).
- Enabling color material (`GL_COLOR_MATERIAL`).

- Enabling non-linear fog (`glFog (GL_FOG_MODE, GL_EXP{2})`). An exception to this is using RGBA mode on Creator3D Series 2.
- Enabling `GL_NORMALIZE`.
- Turning on polygon offset. However, polygon offset is optimized when the factor parameter of the `glPolygonOffset` call is set to 0.0. Users may have to adjust the units parameter accordingly to avoid stitching for this case.

## Attributes That Force the Use of the Generic Software Rasterizer

Setting the following attributes forces the use of the generic software rasterizer. This is the slowest data path. If your application requires any of the following attributes for performance critical functionality, you may want to determine whether this performance is acceptable. If not, you can evaluate whether the use of these attributes is advisable.

1. Texturing Attributes
   - All three-dimensional texturing attributes result in the use of the generic software rasterizer.
   - Two-dimensional texture mapping (`GL_TEXTURE_2D`) in the following cases:

     i. Texture environment mode `glTexEnv GL_TEXTURE_ENV_MODE` is set to `GL_BLEND`.

     ii. `glTexEnv` texture base internal format is `GL_ALPHA`.

     iii. Texturing of points is handled by the generic software.

     iv. Fog is enabled.

     v. Any use of the SGI Texture Color Table (`GL_SGI_texture_color_table`) extension.

2. Fragment Attributes
   - Enabling any type of fog in Indexed color mode.
   - Enabling blending (`glBlendFunc`) (3gl) except when the source blending factor is `GL_SRC_ALPHA` and the destination blending factor is `GL_ONE_MINUS_SRC_ALPHA`. This case is optimized.
   - Enabling logical operations.
   - Enabling depth test `glEnable(GL_DEPTH_TEST)` forces the use of the optimized software rasterizer. If depth test is enabled, then if `glDepthFunc(3gl)` is on, enabling any Z comparison other than `GL_LESS` or `GL_LEQUAL` forces the use of the generic software rasterizer.
   - Enabling alpha test.

- Setting `glDrawBuffer(3gl)` to `GL_BACK` or `GL_FRONT_AND_BACK`, or setting `glReadBuffer(3gl)` to `GL_BACK`.

## Index Mode

When pure software rendering is being used, index mode rendering is handled by the generic software rasterizer. This includes any logic operation, blending, fog, stencil, alpha test, and the above-mentioned cases for Z comparison.

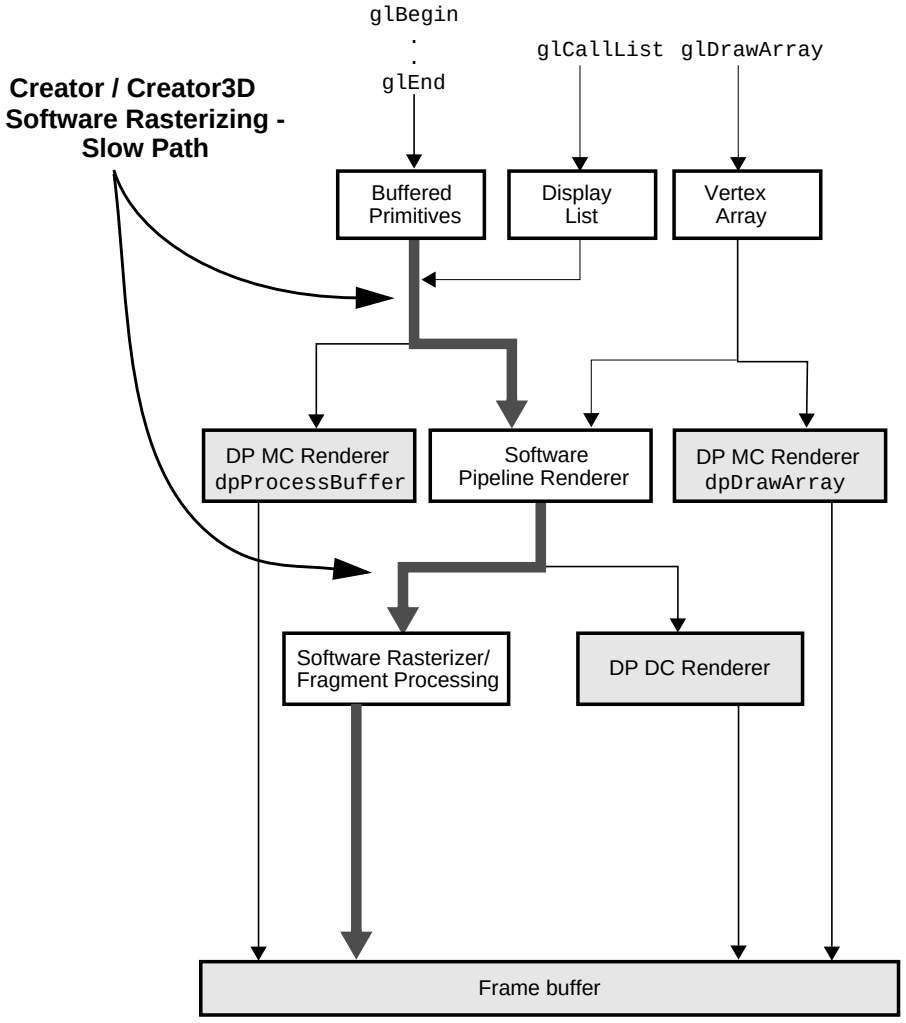


**FIGURE 3-5** Software Rasterizer Data Path for Creator3d and Creator

# Pixel Operations

Under optimal conditions, the commands `glDrawPixels(3gl)`, `glReadPixels`(3gl), and `glCopyPixels`(3gl) are optimized on the Creator and Creator3D systems using the VIS instruction set on the UltraSPARC CPU. Bitmap operations using the command `glBitmap`(3gl) are accelerated in the Creator3D font registers. However, some attribute settings result in the use of the software rasterizer for pixel operations.

FIGURE 3-6 shows the rasterization and fragment processing architecture for `glDrawPixels`(3gl). The figure shows the optimized and unoptimized paths for pixel rendering. Your application will experience performance degradation for each functional box that it needs. In addition, performance degradation will occur if the data type is not unsigned byte; in this case, the data must be reformatted internally.

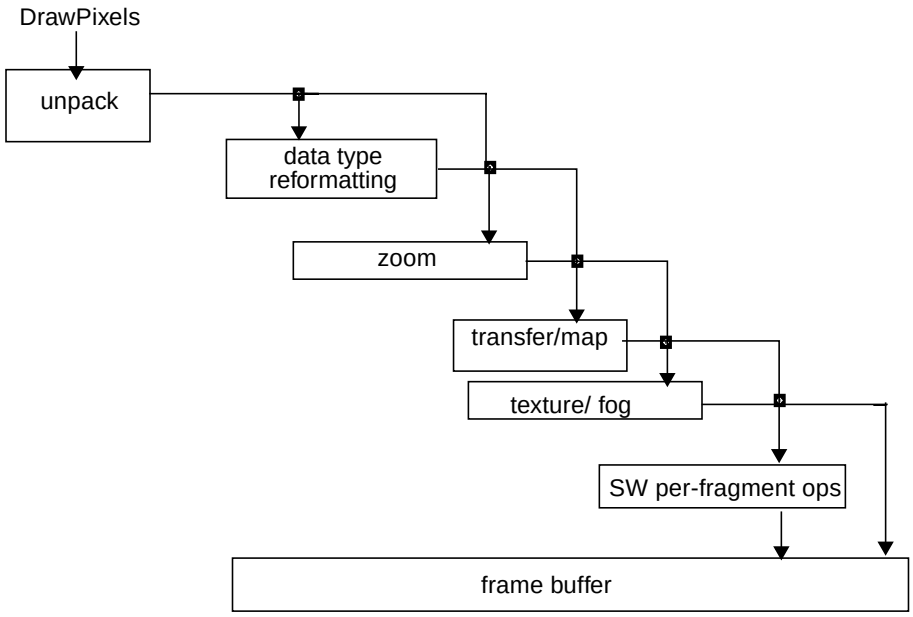


**FIGURE 3-6** OpenGL for Solaris Architecture for Drawing Pixels

## Conditions That Result in VIS Optimization on Creator3D Systems

In general, for DrawPixels, CopyPixels, and Bitmap, the use of texture mapping or nonlinear fog (except in RGBA mode on Creator3D Series 2) will force the use of the generic software rasterizer, resulting in slow performance. In addition, if the

hardware does not support the per-fragment operations that the application has enabled, the generic software rasterizer is used. See the OpenGL documentation or the “OpenGL Machine” diagram for a list of per-fragment operations.

For the Creator3D system, if the following conditions are true, pixel operations are optimized. If these conditions are not true, the generic software rasterizer is used.

### `glDrawPixels` *Command*

- Pixel format is `GL_RGBA`, `GL_RGB`, `GL_ABGR_EXT`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_LUMINANCE`, and `GL_LUMINANCE_ALPHA`.
- Data type is `GL_UNSIGNED_BYTE`. (For `GL_LUMINANCE` the data type can also be `GL_SHORT`).
- For the format of `GL_DEPTH_COMPONENT`, the types `GL_INT`, `GL_UNSIGNED_INT`, and `GL_FLOAT` are optimized for the case with no pixel transfer.
- Texturing is disabled.
- Pixel unpacking is unnecessary.
- For the formats listed in the first line, the pixel transfer operations for scale/bias, pixel map, SGI color table, convolution, SGI post convolution color table, histogram, and minmax may be enabled.
- Pixel Zoom may be done if it zoom factors are other than the default values.
- Pixel transform may be done if its current matrix is other than the identity matrix.

### `glReadPixels` *Command*

- Pixel format is GL_RGBA, GL_RGB, GL_ABGR_EXT, GL_RED, GL_GREEN, GL_BLUE, GL_LUMINANCE, and GL_LUMINANCE_ALPHA.
- Data type is GL_UNSIGNED_BYTE.
- For the format of GL_DEPTH_COMPONENT, the types GL_INT, GL_UNSIGNED_INT, and GL_FLOAT are optimized for the case with no pixel transfer.
- Pixel packing is unnecessary.
- For the formats listed in the first line, the pixel transfer operations for scale/bias, pixel map, SGI color table, convolution, SGI post convolution color table, histogram, and minmax may be enabled.

### `glCopyPixels` Command

- Pixel type is GL_COLOR.
- Texturing is disabled.

- Pixel zooming is in the default state.
- The pixel transfer operations for scale/bias, pixel map, SGI color table, convolution, SGI post convolution color table, histogram, and minmax may be enabled.

### `glBitmap`*(3gl)Command*

- Texturing is not enabled.
- Blending is not enabled.

## Conditions That Result in VIS Optimization on Creator Systems

For the Creator and non-Creator SMCC frame buffers, if the following conditions are true, pixel operations are optimized. If these conditions are not true, the generic software rasterizer is used.

### `glDrawPixels` *Command*

- For GL_LUMINANCE with data types GL_UNSIGNED_BYTE and GL_SHORT, there are special VIS optimized routines for:
  - drawing directly to the framebuffer (or pbuffer).
  - performing pixel transfer (ie. scale/bias, pixel map, SGI color table, convolution, SGI post convolution color table, histogram, and minmax) then displaying directly to the framebuffer (or pbuffer).
  - performing the pixel transform extension, then drawing directly to the framebuffer (or pbuffer).
  - performing pixel transfer followed by the pixel transform extension, then finally drawing directly to the framebuffer (or pbuffer).
- Pixel format is `GL_RGBA`, `GL_RGB` or `GL_ABGR_EXT`.
- Data type is `GL_UNSIGNED_BYTE`.
- Texturing is disabled.
- Pixel unpacking is unnecessary.
- If depth test is enabled, then if `glDepthFunc`(3gl) is on, enabling any Z comparison other than `GL_LESS` or `GL_LEQUAL`.

### `glReadPixels` *Command*

- For `GL_RED` with the data type `GL_UNSIGNED_BYTE`, there is one special VIS optimized routine for extracting the red channel from an `ABGR` framebuffer or pbuffer.

- If `glReadPixels` format is `GL_RGBA`, `GL_RGB`, or `GL_ABGR_EXT`, and the pixel type is `GL_UNSIGNED_BYTE`, then `glReadPixels` is optimized.
- If `glReadPixels` format is `GL_DEPTH_COMPONENT`, then these pixel types are optimized: `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`.
- Pixel packing is unnecessary.

### `glCopyPixels` Command

- Pixel type is `GL_COLOR`.
- Texturing is disabled.
- Enabling any Z comparison other than `GL_LESS` or `GL_LEQUAL`.

### `glBitmap` *Command*

- Texturing is disabled.
- If depth test is enabled, then if `glDepthFunc` is on, enabling any Z comparison other than `GL_LESS` or `GL_LEQUAL`.

# Pixel Transfer Pipeline Imaging Extensions and the Pixel Transform

The Pixel Transfer Pipeline consists of a small set of image processing functions which operate on most rectangular imagery with OpenGL. These operations are performed whenever Pixel Transfer operations can occur within OpenGL (that is,. `glDrawPixels`, `glReadPixels`, `glCopyPixels`, `glTexImage2D`, `glTexImage3DEXT`, and so on).

This pipeline has been fine tuned for maximum performance on GL_LUMINANCE formatted data for the data types GL_UNSIGNED_BYTE and GL_SHORT. Other formats have been accelerated as well; however, GL_LUMINANCE gains the most in performance with this Implementation of the Pipeline.

This pipeline has been accelerated using the Visual Instruction Set, which is only available on those systems with the UltraSPARC processor. The Pixel Transfer Pipeline with VIS acceleration is not supported on Non-UltraSPARC processors; however, the original Pixel Transfer Functionality is still there, minus the new imaging extensions.

# Implementation

The following figure shows the functions and the order of execution (from top to bottom) of these functions in the Pixel Transfer Pipeline:

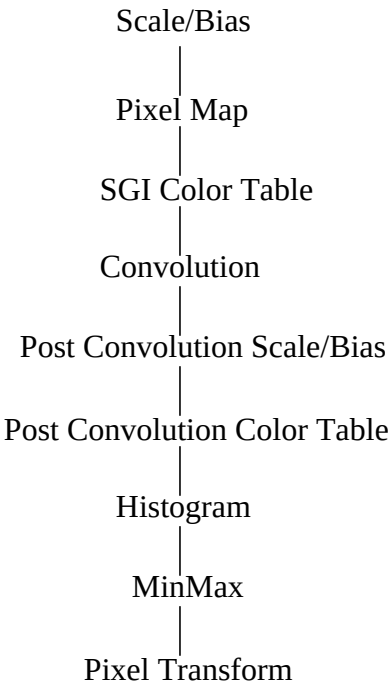


**FIGURE 3-7** Pixel Transfer Pipeline Functions and Order of Execution

All functions in the pipeline have been accelerated using VIS whenever possible. The new imaging extensions within this pipeline are convolution, post convolution scale/bias, post convolution color table, histogram, minmax, and pixel transform. The last one, pixel transform, is not really part of the pixel transfer pipeline, but is instead considered part of the pixel rasterizer. Also, pixel transform is only executed in the `glDrawPixels` interface. The functions for scale/bias, pixel map, and SGI color table are part of the previous release, OpenGL 1.1. The difference here is that they are accelerated using VIS when possible in OpenGL 1.1.1.

Another optimization that is worth noting here is that direct output to the display, via the `glDrawPixels` interface, or into a pbuffer has been optimized for GL_LUMINANCE format with GL_UNSIGNED_BYTE and GL_SHORT data types. For GL_UNSIGNED_BYTE, while the framebuffer is in TrueColor mode (rgb mode), the luminance pixels are expanded to XBGR format and then written directly to the

framebuffer memory using VIS for optimal throughput. For GL_LUMINANCE, GL_SHORT data, the conversion of GL_SHORT data to GL_UNSIGNED_BYTE and then expansion to XBGR for direct display has been optimized for maximum throughput using VIS.

When the input format is GL_LUMINANCE and the input data type is GL_SHORT the Pixel Transfer Pipeline has been made so that it will process the data from the beginning to end of the pipe as GL_SHORT data. This maintains the accuracy and integrity of the data from one stage of the pipeline to the next. Only just before rendering into the frame buffer or pbuffer does the data get scaled down and clamped to [0, 255].

In this pipeline none or all of these processing blocks can be enabled. Any time the Pixel Transfer Pipeline is used, there is only one pass through the pipe, and the order of execution does not change from that represented in the figure above.

## How To Use the Pixel Transfer Pipeline and Pixel Transform

For the most part, OpenGL operates on RGBA colors. Therefore, to be specification compliant in OpenGL, if a user of OpenGL wants to do pixel transfer operations on GL_LUMINANCE data, then that data should first be expanded to GL_RGBA format, (or GL_ABGR_EXT format) before doing any processing. However, depending on the OpenGL pixel transfer state parameters, it may not be necessary to expand the image data before processing in the pixel transfer pipeline. That is, if we expand the data from GL_LUMINANCE to GL_RGBA first, process the image as 4 banded data in the Pixel Transfer Pipeline, and then display, or if we process the GL_LUMINANCE data as a single banded image in the Pixel Transfer Pipeline, then expand the data at the end of the pipeline, then display the data; if the result would be the same using either of the 2 paths, then it makes sense to use the faster path, which, in this case, would be the latter path.

This takes about 1/4th the time, (or less) to do the correct desired operation. The Pixel Transfer Pipeline evaluates the various states of the pixel transfer functions and determines if it needs to do format expansion, before, during, or after processing, but expansion always occurs, if needed, just before rendering to the framebuffer or pbuffer.

The only case were format expansion can occur inside the Pixel Transfer Pipeline is within the “pixel map” block. If you want optimal throughput for GL_LUMINANCE data, do not use pixel map, instead use SGI color table if you need to use a color table at this stage in the pipeline.

The following sections explain each stage of the Pixel Transfer Pipeline. The example code provided shows you how to set the state parameters for the given stage so that GL_LUMINANCE data is not expanded until the very end of the pipeline, just before rendering to the frame buffer's window or the pbuffer.

## Scale / Bias

This operation multiplies all pixels by a given scale value, then adds a bias value. Scale and Bias values can be set differently for each color component of a pixel. These values are set as follows:

```
glPixelTransferf (GL_RED_SCALE,   red_scale_value);
glPixelTransferf (GL_GREEN_SCALE, green_scale_value);
glPixelTransferf (GL_BLUE_SCALE,  blue_scale_value);
glPixelTransferf (GL_ALPHA_SCALE, alpha_scale_value);
glPixelTransferf (GL_RED_BIAS,    red_bias_value);
glPixelTransferf (GL_GREEN_BIAS,  green_bias_value);
glPixelTransferf (GL_BLUE_BIAS,   blue_bias_value);
glPixelTransferf (GL_ALPHA_BIAS,  alpha_bias_value);
```

If any of these deviate from their default values, (1.0 for scale and 0.0 for bias) then the Scale / Bias block in the Pixel Transfer Pipeline is enabled. If any of the red, green, blue, or alpha components differ from each other for either scale or bias, and if the input format can be expanded to GL_RGBA or GL_ABGR_EXT format, then the expansion will occur before processing starts in the pixel transfer pipeline. If the red, green, blue and alpha scale values are all the same or alpha scale is 1.0, and the red, green, blue and alpha bias values are the same or the alpha bias is 0.0, but the red, green, and blue components are different from their default values, then expansion does not need to occur. Hence, if you do a `glDrawPixels` operation and pass in GL_LUMINANCE data, the red component will be used to do the scale and bias, and the output will be a GL_LUMINANCE format image. Hence, the following OpenGL calls will setup Scale / Bias to process GL_LUMINANCE without format expansion:

```
glPixelTransferf (GL_RED_SCALE,   scale_value);
glPixelTransferf (GL_GREEN_SCALE, scale_value);
glPixelTransferf (GL_BLUE_SCALE,  scale_value);
glPixelTransferf (GL_ALPHA_SCALE, scale_value);
glPixelTransferf (GL_RED_BIAS,    bias_value);
glPixelTransferf (GL_GREEN_BIAS,  bias_value);
glPixelTransferf (GL_BLUE_BIAS,   bias_value);
glPixelTransferf (GL_ALPHA_BIAS,  bias_value);
```

or

```
glPixelTransferf (GL_RED_SCALE,   scale_value);
glPixelTransferf (GL_GREEN_SCALE, scale_value);
glPixelTransferf (GL_BLUE_SCALE,  scale_value);
glPixelTransferf (GL_ALPHA_SCALE, 1.0);
glPixelTransferf (GL_RED_BIAS,    bias_value);
glPixelTransferf (GL_GREEN_BIAS,  bias_value);
glPixelTransferf (GL_BLUE_BIAS,   bias_value);
glPixelTransferf (GL_ALPHA_BIAS,  0.0);
```

To disable scale/bias, just reset the scale/bias values back to their default values as shown below:

```
glPixelTransferf (GL_RED_SCALE,   1.0);
glPixelTransferf (GL_GREEN_SCALE, 1.0);
glPixelTransferf (GL_BLUE_SCALE,  1.0);
glPixelTransferf (GL_ALPHA_SCALE, 1.0);
glPixelTransferf (GL_RED_BIAS,    0.0);
glPixelTransferf (GL_GREEN_BIAS,  0.0);
glPixelTransferf (GL_BLUE_BIAS,   0.0);
glPixelTransferf (GL_ALPHA_BIAS,  0.0);
```

## Pixel Map

When in true color mode (RGB mode), if the input image data format is not GL_RGBA or GL_ABGR_EXT, then expansion is always forced if pixel map is enabled using `glPixelTransfer` (GL_MAP_COLOR, GL_TRUE). If the input image format is GL_COLOR_INDEX and the current display mode is RGB, then Pixel Map is called automatically whether it was enabled or not to do the conversion from color index to RGBA. In terms of performance for GL_LUMINANCE, this case is not optimal and you should use SGI color table instead.

To learn how to use Pixel Map consult the "*OpenGL Reference Manual*," by the OpenGL Architecture Review Board, known as the blue book. Read the sections on `glPixelTransfer`, and `glPixelMap`.

## SGI Color Table

This extension is very useful for accelerating color lookup for GL_LUMINANCE data. Other formats are accelerated as well; however, GL_LUMINANCE benefits the most. The following code fragment shows how to correctly setup SGI color table to perform a color lookup for GL_LUMINANCE data:

```
int unpack_row_length;
int unpack_skip_pixels;
int unpack_skip_rows;
int unpack_alignment;
int lut_size;
void *lut;
/* Turns on SGI color table. */
glEnable (GL_COLOR_TABLE_SGI);
/* The current pixel storage modes also affect color table  */
/* definition at the time the color table is created. We    */
/* need to grab the current values, set the row length,     */
/* skip pixels and skip rows to the defaults and            */
/* set unpack alignment to 1. When finished defining the    */
/* color table, restore the original values.                */
glGetIntegerv (GL_UNPACK_ROW_LENGTH,  (long *) &unpack_row_length);
glGetIntegerv (GL_UNPACK_SKIP_PIXELS, (long *) &unpack_skip_pixels);
glGetIntegerv (GL_UNPACK_SKIP_ROWS,   (long *) &unpack_skip_rows);
glGetIntegerv (GL_UNPACK_ALIGNMENT,   (long *) &unpack_alignment);
glPixelStorei (GL_UNPACK_ROW_LENGTH,  0);
glPixelStorei (GL_UNPACK_SKIP_PIXELS, 0);
glPixelStorei (GL_UNPACK_SKIP_ROWS,   0);
glPixelStorei (GL_UNPACK_ALIGNMENT,   1);
/* Define the color table for GL_LUMINANCE.                      */
/* If data type is GL_UNSIGNED_BYTE create a lookup table with   */
/* 256 entries.  Each entry is of type GL_UNSIGNED_BYTE.         */
/* Range of values for any entry is [0, 255].                    */
/* For a GL_SHORT lookup table, generate a table of 65536 entries */
/* ranging from -32768 to 32767.                                 */
if (data_type == GL_UNSIGNED_BYTE) {
       lut_size = 256;
       lut = generate_unsigned_byte_lut();
```

```
}
else if (data_type == GL_SHORT) {
       lut_size = 65536;
       lut = generate_short_lut();
}
glColorTableSGI (GL_COLOR_TABLE_SGI,
       GL_LUMINANCE, /* Need to specify internal format.   */
       lut_size,
       GL_LUMINANCE,      /* Format of lut passed in.      */
       data_type,         /* Data type of lut passed in.   */
       lut);              /* Actual pointer to lut arrayl. */
/* Restore original Pixel Storage values in case something else */
/* needed these values.                                          */
glPixelStorei (GL_UNPACK_ROW_LENGTH,  unpack_row_length);
glPixelStorei (GL_UNPACK_SKIP_PIXELS, unpack_skip_pixels);
glPixelStorei (GL_UNPACK_SKIP_ROWS,   unpack_skip_rows);
glPixelStorei (GL_UNPACK_ALIGNMENT,   unpack_alignment);
```

## Convolution, Post Convolution Scale/Bias and Post Convolution Color Table

---

Convolution comes in 3 flavors: 1D convolution (applies to 1D textures only), 2D general convolution, and 2D separable convolution. Special effort has been made to maximize throughput for 2D general and separable convolutions for GL_LUMINANCE format for GL_UNSIGNED_BYTE and GL_SHORT data types via the `glDrawPixels` interface.

Convolution allows you to set scale and bias values that are applied to the convolution filter kernel before it is used for convolving the image. This is different from post convolution scale/bias (below) in that the bias is applied to the filter itself before processing, where as with post convolution scale/bias, the bias is added to the final convolution result before clamping for the given data type (GL_UNSIGNED_BYTE or GL_SHORT).

Convolution and post convolution scale/bias have been combined into one operation. The kernel values for convolution are multiplied by the scale value of the post convolution scale/bias, then after each pixel is convolved the bias is added. Since this is all done in VIS, there is no loss in performance when compared with an ordinary convolve implemented in VIS.

The OpenGL 1.1.1 implementation of convolution only supports 1x3, 1x5, and 1x7 convolves for 1D convolves, and 3x3, 5x5, and 7x7 for 2D convolves. Also, the source image must be 3 times larger than the size of the convolve kernel to be used.

OpenGL 1.1.1 convolution also supports the following border modes:

GL_REDUCE_EXT, GL_IGNORE_BORDER_HP, GL_CONSTANT_BORDER_HP,

GL_WRAP_BORDER_SUN, GL_REPLICATE_BORDER_HP.

SGI post convolution color table is set up exactly the same way as SGI color table. The only difference being the target value when defining the table.

The code fragment below shows how to setup 2D convolution for both the general and separable cases for a 3x3 convolve on GL_LUMINANCE format image data. The setup is the same for either GL_UNSIGNED_BYTE or GL_SHORT data. It also prepares for using the GL_CONSTANT_BORDER_HP mode, uses the GL_CONVOLUTION_FILTER_SCALE_EXT and the GL_CONVOLUTION_FILTER_BIAS_EXT, sets up for post convolution scale/bias, then finally sets up the SGI post convolution color table.

```
int unpack_row_length;
int unpack_skip_pixels;
int unpack_skip_rows;
int unpack_alignment;
int lut_size;
void *lut;
float kernel3x3[9]  = { 0.111111111, 0.111111111, 0.111111111,
       0.111111111, 0.111111111, 0.111111111,
       0.111111111, 0.111111111, 0.111111111};
float sepkernel3[3] = { 0.333333333, 0.333333333, 0.333333333};
float const_color[4] = { 0.5, 0.5, 0.5, 0.5 };
float kernel_scales[4] = { 0.8, 0.8, 0.8, 0.8 };
float kernel_biases[4] = { 0.2, 0.2, 0.2, 0.2 };
float post_conv_scales[4] = { 0.75, 0.75, 0.75, 0.75 };
float post_conv_biases[4] = { 0.25, 0.25, 0.25, 0.25 };
/* The current pixel storage modes affect convolve kernel   */
/* destination at the time the kernels are created.         */
/* We need to grab the current values, set the row length,  */
/* skip pixels and skip rows to the defaults and set unpack */
/* alignment to 1.                                          */
/* When finished defining the color table, restore the      */
```

```
/* original values.                                          */
glGetIntegerv (GL_UNPACK_ROW_LENGTH,  (long *) &unpack_row_length);
glGetIntegerv (GL_UNPACK_SKIP_PIXELS, (long *) &unpack_skip_pixels);
glGetIntegerv (GL_UNPACK_SKIP_ROWS,   (long *) &unpack_skip_rows);
glGetIntegerv (GL_UNPACK_ALIGNMENT,   (long *) &unpack_alignment);
glPixelStorei (GL_UNPACK_ROW_LENGTH,  0);
glPixelStorei (GL_UNPACK_SKIP_PIXELS, 0);
glPixelStorei (GL_UNPACK_SKIP_ROWS,   0);
glPixelStorei (GL_UNPACK_ALIGNMENT,   1);
/* Now, setup convolution with constant color border mode. */
if (convolve_type == GL_CONVOLUTION_2D_EXT) {
       glEnable (GL_CONVOLUTION_2D_EXT);
       glConvolutionFilter2DEXT (GL_CONVOLUTION_2D_EXT,
                GL_LUMINANCE,   /* Internal format.         */
                3, 3,           /* Kernal dimensions.       */
                GL_LUMINANCE,   /* Input kernel data format */
                GL_FLOAT,       /* Data type for kernel     */
                                /* entries.               */
                (void *) kernel3x3); /* Pointer to kernel  .*/
       glConvolutionParameteriEXT(GL_CONVOLUTION_2D_EXT,
                GL_CONVOLUTION_BORDER_MODE_EXT,
                GL_CONSTANT_BORDER_HP);
       glConvolutionParameterfvEXT(GL_CONVOLUTION_2D_EXT,
                GL_CONVOLUTION_BORDER_COLOR_HP,
                const_color);
       glConvolutionParameterfvEXT(GL_CONVOLUTION_2D_EXT,
                GL_CONVOLUTION_FILTER_SCALE_EXT,
                kernel_scales);
       glConvolutionParameterfvEXT(GL_CONVOLUTION_2D_EXT,
                GL_CONVOLUTION_FILTER_BIAS_EXT,
                kernel_biases);
}
else if (convolve_type == GL_SEPARABLE_2D_EXT) {
       glEnable (GL_SEPARABLE_2D_EXT);
       glSeparableFilter2DEXT (GL_SEPARABLE_2D_EXT,
                GL_LUMINANCE,
                3, 3,
                GL_LUMINANCE,
```

```
                GL_FLOAT,
                sepkernel3,  /* Horizontal Kernal Values. */
                sepkernel3); /* Vertical Kernal Values.   */
        glConvolutionParameteriEXT(GL_SEPARABLE_2D_EXT,
                GL_CONVOLUTION_BORDER_MODE_EXT,
                GL_CONSTANT_BORDER_HP);
        glConvolutionParameterfvEXT(GL_SEPARABLE_2D_EXT,
                GL_CONVOLUTION_BORDER_COLOR_HP,
                const_color);
        glConvolutionParameterfvEXT(GL_SEPARABLE_2D_EXT,
                GL_CONVOLUTION_FILTER_SCALE_EXT,
                kernel_scales);
        glConvolutionParameterfvEXT(GL_SEPARABLE_2D_EXT,
                GL_CONVOLUTION_FILTER_BIAS_EXT,
                kernel_biases);
}
glPixelTransferf(GL_POST_CONVOLUTION_RED_SCALE_EXT,
        post_conv_scales[0]);
glPixelTransferf(GL_POST_CONVOLUTION_GREEN_SCALE_EXT,
        post_conv_scales[1]);
glPixelTransferf(GL_POST_CONVOLUTION_BLUE_SCALE_EXT,
        post_conv_scales[2]);
glPixelTransferf(GL_POST_CONVOLUTION_ALPHA_SCALE_EXT,
        post_conv_scales[3]);
glPixelTransferf(GL_POST_CONVOLUTION_RED_BIAS_EXT,
        post_conv_biases[0]);
glPixelTransferf(GL_POST_CONVOLUTION_GREEN_BIAS_EXT,
        post_conv_biases[1]);
glPixelTransferf(GL_POST_CONVOLUTION_BLUE_BIAS_EXT,
        post_conv_biases[2]);
glPixelTransferf(GL_POST_CONVOLUTION_ALPHA_BIAS_EXT,
        post_conv_biases[3]);
/* Turns on SGI post convolution color table. */
glEnable (GL_POST_CONVOLUTION_COLOR_TABLE_SGI);
/* Define the color table for GL_LUMINANCE.                        */
/* If data type is GL_UNSIGNED_BYTE create a lookup table with     */
```

```
/* 256 entries.  Each entry is of type GL_UNSIGNED_BYTE.          */
/* Range of values for any entry is [0, 255].                     */
/* For a GL_SHORT lookup table, generate a table of 65536 entries */
/* ranging from -32768 to 32767.*/

if (data_type == GL_UNSIGNED_BYTE) {
       lut_size = 256;
       lut = generate_unsigned_byte_lut();
}
else if (data_type == GL_SHORT) {
       lut_size = 65536;
       lut = generate_short_lut();
}
glColorTableSGI (GL_POST_CONVOLUTION_COLOR_TABLE_SGI,
           GL_LUMINANCE,      /* Need to specify internal format. */
           lut_size,
           GL_LUMINANCE,      /* Format of lut passed in. */
           data_type,         /* Data type of lut passed in. */
           lut);              /* Actual pointer to lut arrayl. */
/* Restore original Pixel Storage values in case something else */
/* needed these values.                                          */
glPixelStorei (GL_UNPACK_ROW_LENGTH,  unpack_row_length);
glPixelStorei (GL_UNPACK_SKIP_PIXELS, unpack_skip_pixels);
glPixelStorei (GL_UNPACK_SKIP_ROWS,   unpack_skip_rows);
glPixelStorei (GL_UNPACK_ALIGNMENT,   unpack_alignment);
```

## Histogram and Minmax

The Histogram and Minmax operations come at the end of the Pixel Transfer Pipeline. When used, both can have their own “sink” values. If sink is enabled (GL_TRUE), processing of image data stops here, and does not continue down the pipeline and no output is generated. If the histogram's sink value is true, then minmax is not executed. (See the man pages for more information about the sink behavior of these operations).

The code below gives an example of getting a histogram for GL_LUMINANCE and data for both GL_UNSIGNED_BYTE and GL_SHORT. Notice below that the requested width of the histogram definition for GL_SHORT has been specified to be 32768 instead of 65536. The reason is that, for GL_SHORT data, the data is effectively

clamped in the range [0, 32767]. That is, if any of the GL_SHORT values are negative, they will contribute to the very first histogram bin counter value for 0. Specifying a larger width is pointless since only every other histogram bin would have a value in it. Histogram widths, in general, may be any value which is a power of 2 in the range [0, 65536]. However, for those cases where you want to actually display the computed histogram, you can specify a smaller width for GL_SHORT data type, say 256, 512, or 1024. This saves you the time because you do not have to do the code. By requesting a smaller histogram width, histogram bins are added together. For example, for GL_SHORT, if you requested a width of 256, each returned bin value in the histogram image would have 128 bins added together. Hence, all values in the range [0, 127] would be in bin 0. All values in the range [128, 255] would be in bin 1, and so on.

Minmax uses the histogram to compute its values. It gets the minmax values using the histogram for the full width of the positive values for GL_UNSIGNED_BYTE and GL_SHORT. Therefore, if the histogram is taken of GL_UNSIGNED_BYTE, the possible range of minmax values is [0, 255]. For GL_SHORT, the possible range of minmax values is [0, 32767].

```
int             minmax[2];
int             histogram[32768];
unsigned char   *uc_buff;
short           *s_buff;
glEnable(GL_HISTOGRAM_EXT);
glEnable(GL_MINMAX_EXT);
/* Allocate enough space for 64 x 64 GL_LUMINANCE images. */
uc_buff = (unsigned char *) malloc (4096*sizeof(unsigned char));
s_buff  = (short *)         malloc (4096*sizeof(short));
/* First, do it for GL_UNSIGNED_BYTE with GL_LUMINANCE format. */
glHistogramEXT(GL_HISTOGRAM_EXT, 256, GL_LUMINANCE, GL_FALSE);
glMinmaxEXT(GL_MINMAX_EXT, GL_LUMINANCE, GL_FALSE);
glDrawPixels(64, 64, GL_LUMINANCE, GL_UNSIGNED_BYTE, uc_buff);
/* Since the call to glHistogramEXT defined a width of 256, */
/* 256 entries of the histogram array will be filled in.    */
/* The remaining entries in the array are untouched.        */
glGetHistogramEXT(GL_HISTOGRAM_EXT, GL_TRUE, GL_LUMINANCE, GL_INT,
           histogram);
glGetMinmaxEXT(GL_MINMAX_EXT, GL_TRUE, GL_LUMINANCE, GL_INT,
           minmax);
/* Do something with the histogram and minmax. */
/* Now, do GL_SHORT data. */
```

```
glHistogramEXT(GL_HISTOGRAM_EXT, 32768, GL_LUMINANCE, GL_FALSE);
glMinmaxEXT(GL_MINMAX_EXT, GL_LUMINANCE, GL_FALSE);
glDrawPixels(64, 64, GL_LUMINANCE, GL_SHORT, s_buff);
/* Since the call to glHistogramEXT defined a width of 32768, */
/* 32768 entries of the histogram array will be filled in.    */
glGetHistogramEXT(GL_HISTOGRAM_EXT, GL_TRUE, GL_LUMINANCE, GL_INT,
       histogram);
glGetMinmaxEXT(GL_MINMAX_EXT, GL_TRUE, GL_LUMINANCE, GL_INT,
       minmax);
```

## Pixel Transform

Pixel Transform, while shown at the end of the Pixel Transfer Pipeline, is not part of it. Pixel Transform is in the Pixel Rasterizer, and it only works through the `glDrawPixels` interface.

Pixel Transform has been especially optimized for applying affine transformation warping to an input image on its way to the frame buffer or pbuffer. It has been specially tuned for handling GL_LUMINANCE format and the GL_UNSIGNED_BYTE and GL_SHORT data types. For GL_SHORT, the data is scaled and clamped to [0, 255] and then warped into the frame buffer or pbuffer. On the way to the frame buffer, the data is also expanded from GL_LUMINANCE data to XBGR format, which is the native format of the frame buffer while in rgb mode.

Pixel Transform has its own matrix mode with its own matrix stack 32 deep.

```
glMatrixMode(GL_PIXEL_TRANSFORM_2D_EXT);
```

Pixel Transform is always enabled; however, if its current matrix is the identity matrix, then the pixel transform is not performed. Only when the current matrix is not the identity matrix will pixel transform be performed.

You can use all of the existing API calls available for matrix operations in OpenGL. These will operate on the current matrix of the GL_PIXEL_TRANSFORM_2D_EXT matrix mode (that is, `glLoadMatrix`, `glTranslate`, `glRotate`, `glScale`, `glLoadIdentity`, `glPushMatrix`, `glPopMatrix`, `glMultMatrix`, and so on). When using these matrix operators on the current matrix, after the operation is performed, only the affine components are kept. Entries in the matrix which apply to the z and w components are left like they were initialized with the identity matrix.

The pixel transform extension operates as if the current raster position is the origin of the coordinate system. To simplify, set the current raster position to be located in the lower left corner of the display window, then figure out your operations. If you want to translate the image, you can use `glTranslate`, or move the current raster

position. The difference is that `glTranslate` will be integrated into the total transformation for pixel transform, while moving the raster position will translate the image regardless of the current matrix contents of the pixel transform matrix.

glPixelZoom also affects the pixel transform current matrix; however, only if the current matrix mode is set to GL_PIXEL_TRANSFORM_2D_EXT. Also, if glPixelZoom is called, it replaces the contexts of the current matrix as shown below:

```
+--                          --+
| x_zoom   0      0        0  |
|   0    y_zoom   0        0  |
|   0      0      1        0  |
|   0      0      0        1  |
+--                          --+
```

If the current matrix mode is not GL_PIXEL_TRANSFORM_2D_EXT, then the current matrix of GL_PIXEL_TRANSFORM_2D_EXT is not replaced. However, pixel zoom will still be set.

If the current matrix of GL_PIXEL_TRANSFORM_2D_EXT has been set to something different than identity, and glPixelZoom has been set, then the pixel transform will override the glPixelZoom operation.

If you want to do any image warping, use the pixel transform extension. Do not use the glPixelZoom interface. Instead, use glScale to set up a zoom matrix. If you are using multiple matrix operations on the pixel transform's current matrix, do not use glPixelZoom in the middle or end of the list of operations since it will reset the matrix (shown above) and remove the affect of any previous operations. Instead, use glScale.

Pixel Transform supports 4 types of resampling for minification and 3 types for magnification. GL_NEAREST, GL_LINEAR, and GL_CUBIC_EXT are shared by minification and magnification. GL_AVERAGE_EXT is only supported for minification.

The code fragment below demonstrates how to prepare a pixel transform matrix to do an arbitrary rotation of “angle” degrees about the center of the input image in the center of the frame buffer display window. It assumes the image is GL_LUMINANCE data and GL_UNSIGNED_BYTE. It also sets up the resampling method to be GL_LINEAR for minification and GL_CUBIC_EXT for magnification and sets the GL_CUBIC_WEIGHT_EXT to have the value -0.5.

```
double rotation_angle;
int    window_width, window_height;
int    image_width, image_height;
unsigned char *image_data;
/* Grab needed values for placing image in center. */
window_width   = get_window_width();
window_height  = get_window_height();
```

```
image_width    = get_image_width();
image_height   = get_image_height();
image_data     = get_image_data();
rotation_angle = get_rotation_angle_between_0_and_360_degrees();
/* Prepare current pixel transform matrix. */
glMatrixMode(GL_PIXEL_TRANSFORM_2D_EXT);
glLoadIdentity();
glTranslated(window_width/2.0, window_height/2.0, 0.0);
glRotated(rotation_angle, 0.0, 0.0, 1.0);
glTranslated (-image_width/2.0, -image_height/2.0, 0.0);
/* Set up resampling methods. */
glPixelTransformParameteriEXT(GL_PIXEL_TRANSFORM_2D_EXT,
               GL_PIXEL_MIN_FILTER_EXT,
               GL_LINEAR);
glPixelTransformParameteriEXT(GL_PIXEL_TRANSFORM_2D_EXT,
               GL_PIXEL_MAG_FILTER_EXT,
               GL_CUBIC_EXT);
glPixelTransformParameterfEXT(GL_PIXEL_TRANSFORM_2D_EXT,
               GL_PIXEL_CUBIC_WEIGHT_EXT,
               -0.5);
/* Finally, render the image to the screen. */
glDrawPixels (image_width, image_height, GL_LUMINANCE,
               GL_UNSINGED_BYTE,
               image_data);
```

# GX Performance

GX performance is affected by attributes that force the use of the generic software rasterizer:

1. Texturing Attributes

   a. Only triangles are optimized. Texturing of points and lines is handled by the generic software.

   b. Texture environment mode `glTexEnv`(3gl) `GL_TEXTURE_ENV_MODE` is `GL_BLEND`.

2. Fragment Attributes
    a. Stencil operations
    b. Logic operations
    c. Any blending operation
    d. Linear or nonlinear fog
    e. Enabling any Z comparison other than GL_LESS or GL_LEQUAL

CHAPTER 4

# X Visuals for the OpenGL for Solaris Software

## Programming With X Visuals for the OpenGL for Solaris Software

OpenGL rendering is supported on a subset of the visuals exported by the Solaris X window server on the Creator and Creator3D workstations. Because GLX overloads the core X visual classes with a set of attributes that indicate frame buffer capabilities, such as double buffer mode or stereo capabilities, the number of visuals supported by an OpenGL-capable X server is potentially large. For example, for the 24-bit TrueColor visual, the Solaris X window server on the Creator and Creator3D workstations exports the following types of GLX visuals: double buffer, single buffer, monoscopic, and stereoscopic.

This approach of exporting multiple GLX visuals for each X protocol core visual is colloquially referred to as the GLX *expansion* (or *visual explosion)*. For each different type of GLX visual that is exported, there is a corresponding X protocol core visual. Thus, there are multiple GLX visuals whose core X visual attributes are all identical.

---

**Note –** OpenGL for Solaris does not support windows with backing store. Enabling backing store on a window will penalize the user's Creator3D rendering performance.

---

Various OpenGL-capable visuals are supported in various releases of the Solaris operating environment. These are the visuals that an OpenGL program can use. This information applies to both Creator3D and Creator systems.

- In Solaris 2.5.1-based systems, expanded visuals are *disabled* by default. The user will have the option of enabling or disabling expanded visuals by using the command `ffbconfig -expvis` <*enable*|*disable*>.

See Table 4-1 and Table 4-2 for detailed information on using OpenGL with or without expanded visuals.

---

**Note –** In Solaris 2.5.1-based systems, an OpenGL-capable overlay visual is present only if you run `/usr/sbin/ffbconfig -sov` *enable* before the Window system is started. You must run this command as `root`.

---

The advantage to the overloading of X visuals is that the X server can be specific about the frame buffer configurations that the graphics hardware provides. This approach also enables the OpenGL implementation to better manage resources. Instead of allocating the maximal amount of resources for each window, the OpenGL implementation only needs to allocate the resources necessary for the GLX visual the application has selected. Thus, the application has more direct control over resource allocation.

Using the `glXGetConfig`(3gl) and `glXChooseVisual`(3gl) routines, applications can get information on the supported visuals and choose the appropriate visual. For helpful information on GLX programming, refer to *OpenGL Programming for X Windows Systems* by Mark Kilgard and *OpenGL Programming Guide*.

Table 4-1 lists OpenGL-capable visuals with expanded visuals.

**Table 4-1** OpenGl-capable Visuals With Expanded Visuals

| Double Buffer Capable? | GLX BufferSize | X Visual Class | GL_RGBA | Gamma Corrected? | GLX Level |
|---|---|---|---|---|---|
| Yes | 24 | TrueColor | True | No | 0 |
| Yes | 24 | TrueColor | True | Yes | 0 |
| Yes | 24 | DirectColor | True | No | 0 |
| Yes | 8 | PseudoColor | False | No | 0 |
| No | 24 | TrueColor | True | No | 0 |
| No | 24 | TrueColor | True | Yes | 0 |
| No | 24 | DirectColor | True | No | 0 |
| No | 8 | PseudoColor | False | No | 0 |
| No | 8 | PseudoColor | False | No | 1 |

When the frame buffer video mode is monoscopic, only GL_MONO versions of these visuals are supported. In a stereoscopic video mode, both GL_MONO and GL_STEREO versions of these visuals are supported.

Table 4-2 lists OpenGL-capable visuals without expanded visuals.

**Table 4-2** OpenGL-capable Visuals Without Expanded Visuals

| Double Buffer Capable? | GLX BufferSize | X Visual Class | GL_RGBA | Gamma Corrected? | GLX Level |
|---|---|---|---|---|---|
| Yes | 24 | TrueColor | True | No | 0 |
| Yes | 24 | TrueColor | True | Yes | 0 |
| Yes | 24 | DirectColor | True | No | 0 |
| Yes | 8 | PseudoColor | False | No | 0 |
| No | 8 | PseudoColor | False | No | 1 |

# Colormap Flashing for OpenGL Indexed Applications

With the visuals exploded, there is greater potential for colormap flashing to occur for OpenGL indexed applications. This is because applications are forced to create private colormaps in order to create windows on the GLX visual they choose. In the

Solaris 2.6 release, the colormap flashing problem is eased by the colormap equivalence feature. This feature allows OpenGL color indexed applications to be written in a way that creates less flashing.

Colormap equivalence allows a program to assign a colormap of one visual to a window that was created with a different visual, as long as the two visuals are colormap equivalent. This means, in general, that they share the same plane group and have the same number of colormap entries. The standard X11 protocol does not let programs mix visuals of colormaps and windows in this way. For more information on colormap equivalence, see the `XSolarisCheckColormapEquivalence`(3) man page.

Colormap equivalence is useful for OpenGL programs because the GLX visual expansion creates up to four different variants of each base `GL_CAPABLE` visual. So, for example, instead of one 8-bit default PseudoColor colormap, there may be a double-buffered variant, a stereo variant, and so on. Without colormap equivalence, an application cannot assign the default colormap to windows of these variant visuals, and this will result in more colormap flashing. With colormap equivalence, windows of all variants can share a colormap that was created using the base visual, and less colormap flashing will occur.

# GL Rendering Model and X Visual Class

OpenGL RGBA rendering is supported on the 24-bit TrueColor and DirectColor visuals. OpenGL indexed rendering is supported on the 8-bit PseudoColor visuals and on the indexed or 224-color overlay visuals.

# Depth Buffer

All GL-capable visuals, except for overlay visuals, have a 28-bit Z buffer (`GLX_DEPTH_SIZE == 28`).

## Accumulation Buffer

All GL RGBA visuals have a (16, 16, 16, 16) accumulation buffer (`GLX_ACCUM_RED_SIZE == GLX_ACCUM_GREEN_SIZE== GLX_ACCUM_BLUE_SIZE == GLX_ACCUM_ALPHA_SIZE = 16`).

## Stencil Buffer

All GL capable visuals, except for the overlay and stereo visuals, have a 4-bit stencil buffer (`GLX_STENCIL_SIZE == 4`).

## Auxiliary Buffers

Auxiliary buffers are not supported by the OpenGL for Solaris product (`GLX_AUX_BUFFERS == 0`).

## Stereo

---

**Note –** This section is specific to Creator and Creator3D systems.

---

To run a stereo application in stereo mode, the frame buffer must be configured for stereo operation.

### ▼ To Set Up the Frame Buffer for Stereo Operation:

1. **Exit the window system.**

2. **Type this command**:

   For Solaris 2.5.1 HW297 `/usr/sbin/ffbconfig -res stereo -expvis` *`enable`*

   For Solaris 2.6 `/usr/sbin/ffbconfig -res stereo`

---

**Note –** In the Solaris 2.6 release, this command must be run under superuser permissions or sys admin permissions.

---

3. **Restart the window system.**

   Application can now use the stereo hardware buffers.

# Rendering to DirectColor Visuals

The OpenGL API has no support for color mapping. The only way to get a DirectColor visual is to implement visual selection in the application using `XGetVisualInfo`(3gl) and `glXGetConfig`. If you request a visual with `glXChooseVisual`, you will get a 24-bit TrueColor visual for RGBA rendering and an 8-bit PseudoColor visual for index rendering.

When rendering to DirectColor visuals, the GL system calculates pixel values in the same way as it does for TrueColor visuals. The application is responsible for loading the window colormap with cells that make sense to the application.

# Overlays

The Creator and Creator3D systems have one 8-bit overlay visual in monoscopic mode and two 8-bit overlays in stereo mode. The overlay visual GLX level is greater than zero (`GLX_LEVEL > 0`). Visuals with a GLX level less than or equal to zero are underlay visuals.

## Server Overlay Visual (SOV) Convention

Server Overlay Visual (SOV) is an API for rendering transparent pixels in an overlay window. A transparent pixel is a special pixel code that allows the contents of underlay windows underneath to show through. SOV derives its name from the X

property that informs the user of the special transparent pixel value: SERVER_OVERLAY_VISUALS. This value can be used as the input value to glIndex* calls so that the transparent pixel can be rendered into the overlay.

The SOV API, while not an X11 standard, is a convention that is supported by many X11 vendors. It is described at length in the book *OpenGL Programming for the X Window System* by Mark J. Kilgard. This section describes Sun-specific aspects of the SOV implementation.

---

**Note –** In this section, the term underlay is used as a synonym for the normal planes referred to in *OpenGL Programming for the X Window System*.

---

The SERVER_OVERLAY_VISUALS property describes visuals with transparent pixels (TransparentType = TransparentPixel), and also completely opaque visuals (TransparentType = None). If you need an overlay visual with a transparent pixel, make sure that you check the TransparentType field of the entries in this property. The remainder of this section will discuss only the TransparentPixel SOV visuals.

## Enabling SOV Visuals

SOV visuals are present by default in Solaris 2.6. But in Solaris 2.5.1 HW297, they must be explicitly enabled. SOV visuals can be enabled in an OpenWindows environment by becoming root, then typing the following command before starting the OpenWindows™ system: /usr/sbin/ffbconfig -sov *enable*. Then restart the Window system.

Both Creator and Creator3D platforms support SOV visuals. When these devices are configured for a monoscopic video mode, there is one TransparentPixel SOV visual. When in a stereoscopic video mode, there are two TransparentPixel SOV visuals exported: a monoscopic visual and a stereoscopic visual.

---

**Note –** Regardless of the video mode, there is always one overlay visual exported on these devices that is *not* SOV-capable. This visual is provided in order to support OVL, the Sun-specific overlay extension. This visual is not GL_CAPABLE and is never returned by glXChooseVisuals.

---

# OpenGL Restrictions on SOV

---

**Note –** Creator and Creator 3D Series 3 systems support SOV directly when the ftbconfig-extovl option is enabled. Earlier Creator and Creator 3D systems do not directly support SOV, so the OpenGL for Solaris software provides the SOV support using a low-overhead software translation mechanism. If a program follows the restrictions described below, this mechanism provides rendering to SOV windows at full hardware speeds in most cases.

---

On Creator and Creator 3D systems earlier than series 3, SOV is fully supported on SOV-capable visuals except for the following features, which are not supported:

- Uncorrelated window configurations. These window configurations are described below.
- Read back of transparent pixels via `glReadPixels.`
- Interframebuffer copies of transparent pixels via `glCopyPixels`.
- Logic operations other than `GL_COPY`.
- Index masks other than 0xff.
- `glShadeModel` (`GL_SMOOTH`).

If one of these unsupported features is used, rendering will complete without generating an error but the visual results will be undefined.

A correlated window configuration is a combination of an overlay and an underlay window that are the exact same size and shape. Typically, the overlay window is a child of the underlay window, but it may also be a sibling. In either case, there must be no other windows (mapped or unmapped) intervening between them. Once the window configuration is set up, it should not be changed by re-parenting one of the windows. If a window configuration doesn't meet this definition, then it is called an uncorrelated configuration and is not supported by OpenGL.

The application is responsible for maintaining the correlated relationship. The system does not maintain it automatically. The client must check for underlay window shape changes and if any occur, it must perform the equivalent changes on the overlay window.

# Compatibility of SOV with other Overlay Models

Programs that use SOV visuals may coexist on the same screen with programs that use OVL, the Sun-specific overlay extension. But the two may not be used simultaneously with the same window.

Some XGL™ and OpenGL 1.0 programs are written to use the SOV transparent pixel if the SOV property is present, and to use XOR rendering in the default underlay visual if the SOV property is not present. These programs may not behave properly when the SOV property is present. When the SOV property is not present and the underlay is being used, a program may simply attach the default colormap to the default visual underlay window. In the presence of the SOV visual, the program will switch to using the SOV overlay visual but may continue to use the default colormap. Since the SOV overlay visual is usually not the same as the default visual, this will result in an X11 BadMatch error when the program attempts to attach the colormap to the overlay window. Care should be taken to write programs that always attach colormaps of the proper visual to overlay windows. In this case, the program should have created a colormap using the SOV visual instead of trying to use the default colormap.

Programs that use SOV can also coexist with programs using the Sun visual overlay capability `glXGetTransparentIndexSUN`. However, `glXGetTransparentIndexSUN` is deprecated. It is provided only for compatability for existing programs that use it. Newly written transparent overlay programs should use SERVER_OVERLAY_VISUALS instead.

For information on using the Sun visual overlay capability, see the `glXGetTransparentIndexSUN` man page. In addition, look at the overlay example programs included in the `SUNWglut` package. These programs are installed by default into the directory `/usr/openwin/share/src/GL/contrib/examples/sun/overlay`.

# Gamma Correction

On Creator and Creator3D workstations, two 24-bit TrueColor visuals are exported. One is gamma corrected; the other is not. To support imaging and Xlib applications, the nonlinear (not gamma-corrected) visuals are listed before linear visuals. However, to provide linear visuals for graphics applications running under the OpenGL for Solaris software, the `glXChooseVisual()` call was modified to return a linear visual.

If you want to use a nonlinear TrueColor visual, you need to get the visual list from Xlib. Use the Solaris API `XSolarisGetVisualGamma`(3) to query the linearity of the visual. To determine whether a visual supports OpenGL, call `glXGetConfig` with *attrib* set to `GLX_USE_GL`.

If you are using another vendor's OpenGL and displaying your application on a Creator or Creator3D graphics workstation, and you want to use a linear visual, run the command `/usr/sbin/ffbconfig -linearorder first` to change the order

of visuals so that the linear (gamma-corrected) visual is the first visual in the visual list. See *Solaris X Window System Developer's Guide* for more information on gamma correction and `XSolarisGetVisualGamma`.

CHAPTER 5

# Tips and Techniques

This chapter presents miscellaneous topics that you may find useful as you port your application to the OpenGL for Solaris software.

## Avoiding Overlay Colormap Flashing

Colormap flashing may occur when your application uses overlay windows. This problem stems from several characteristics of the Creator3D system: the overlay visual is not the default visual, the Creator3D is a single hardware colormap device, and X11 allocates colormap cells from pixel 0 upward. When the application renders to the overlay window, it must use a non-default visual, and a non-default colormap is loaded. In this case, colormap flashing between the default and non-default colormaps can occur.

The best solution to this problem is to allocate the overlay colors at the high end of the overlay colormap. In other words, if you have *n* colors to allocate, allocate them in the positions *colormap_size -n -1* to *colormap_size -1*.This avoids the colors in the default colormap, which are allocated upward starting at 0. To allocate *n* colors at the top of the overlay colormap, first allocate *colormap_size-n* read / write placeholder cells using `XAllocColorCells`. Then allocate the *n* overlay colors using `XAllocColor`. Finally, free the placeholder cells. This solution is portable; it works on both single- and multiple-hardware colormap devices.

# Changing the Limitation on the Number of Simultaneous GLX Windows

There is a limitation on the number of GLX windows that an application can use simultaneously. Each GLX window that has an attached GLX context uses a file descriptor for DGA (Direct Graphics Access) information. You can find the current number of open file descriptors using the `limit`(1) command:

```
% limit descriptors
descriptors 64
```

The system response tells you that you have up to 64 direct GLX contexts, assuming that you have no other processes concurrently using file descriptors.

You can increase the per-process maximum number of open file descriptors using the `limit` command as follows:

```
% limit descriptors 128
```

This command changes the number of file descriptors available for DGA and other uses to 128. Use the `sysdef`(1M) command to determine the maximum number of file descriptors for your system.

# Hardware Window ID Allocation Failure Message

On Creator3D, when a program calls `glXMakeCurrent`(3gl) to make a window the current OpenGL drawable, the system will attempt to allocate a unique hardware window ID (WID) for the window. This allows double buffering and hardware WID clipping to be used. Because hardware WIDs are a scarce resource and can be used for other purposes, there might not be any WIDs available when `glXMakeCurrent` is called. If this should happen, the following message is displayed:

OpenGL/FFB Warning: unable to allocate hardware window ID

In this situation, double buffering will not be provided for the window, and the window will be treated as a single-buffered window.

# Getting Peak Frame Rate

The frame rate that `ogl_install_check` prints out is synchronized to monitor frequency. It measures the time it takes to render the frame, wait for `vblank`, then swap the buffers. Since `FFB` can render the `ogl_install_check` image very quickly, even on an `FFB1` Electron 167 mhz machine, the bottleneck is waiting for the monitor `vblank`. So, under normal circumstances, `ogl_install_check` is never going to be able to get a frame rate faster than the monitor frequency.

However, there is an environment variable called `OGL_NO_VBLANK` that you can set to see the peak, unsynchronized frame rate. When set, this environment variable swaps buffers immediately, without waiting for `vblank`.

# Identifying Release Version

You can identify the Release Version Number of the OpenGL Library by:

1. Using the `what`(1) or `mcs`(1) command:

   ```
   % what /usr/openwin/lib/libGL.so.1

   % mcs -p /usr/openwin/lib/libGL.so.1
   ```

2. Programatically, by calling `glGetString (GL_VERSION)`

   (see the `glGetString` man page for more details)

3. Running the OpenGL for Solaris `install_check` demo program:

   ```
   % /usr/openwin/demo/GL/ogl_install_check
   ```

# Pixmap Rendering

OpenGL for Solaris does not support GLX pixmaps with direct rendering contexts. Use indirect rendering contexts (see the `glXCreateContext(3gl)` man page for indirect rendering contexts).

# Determining Visuals Supported by a Specific Frame Buffer

To determine what visuals a specific frame buffer supports, use `/usr/openwin/demo/GL/xglinfo`.

# Creator3D Fog

There is a hardware problem with the linear perspective fog on Creator3D that causes the fog color to overtake the scene color faster than it should for a given depth. As a workaround, you can increase the start and end linear fog parameters appropriately. For instance, in a scene where the fog start and end parameters are equal to the start and end of the perspective view frustum, you should increase the start parameter to be as close as possible to the start of the geometry. You can increase the end parameter to attenuate the effect of the scene getting dark too rapidly. Also, it helps to modify the Z begin and Z end values of the view frustum so that they are closer together.

This problem is fixed in Creator3D Series 2.