SGI® OpenGL Multipipe™ User's Guide

Version 2.5.4

007-4318-018

CONTRIBUTORS

Written by Ken Jones and Jenn Byrnes

Illustrated by Chrystie Danzer

Production by Karen Jacobson and Ken Jones

Engineering contributions by Ye Cong, Craig Dunwoody, Bill Feth, Alpana Kaulgud, Claude Knaus, Ravid Na'ali, Jeffrey Ungar, Christophe Winkler, Guy Zadicario, and Hansong Zhang

COPYRIGHT

LIMITED RIGHTS LEGEND

TRADEMARKS AND ATTRIBUTIONS

New Features in This Release

This revision of the guide documents features from both OpenGL Multipipe 2.5.3 and OpenGL Multipipe 2.5.4.

OpenGL Multipipe 2.5.4 features, released with SGI ProPack 4 Service Pack 3:

- Two performance monitoring tools: `ompmon` and a Performance Co-Pilot data collection agent
- Support for context sharing between threads
- Time-Based decomposition using hardware compositors
- Memory placement and use of the `tmpfs` filesystem
- Support for the OpenGL Shading Language and the ARB_texture_rectangle extension
- Slave synchronization on **glFlush()** for single-buffer windows

OpenGL Multipipe 2.5.3 features, released with SGI ProPack 4 Service Pack 2:

- Improved immediate mode performance
- Improved performance for downloading pixel data
- Preservation of triangle strips when spatially splitting a display list
- Better control over the number of splits done by `dlSplit`

Record of Revision

Version	Description
001	August 2000 Beta release.
002	November 2000 Updated for release 1.0 of the OpenGL Multipipe product.
003	February 2001 Updated for release 1.1 of the OpenGL Multipipe product. New features: - Increased overall performance - Support for overlapping screens, as in SGI Reality Center facilities
004	May 2001 Updated for release 1.2 of the OpenGL Multipipe product. New features: - Transparent OpenGL Pipe Management - Subset of multipipe applications made aware of Xinerama
005	August 2001 Updated for release 1.3 of the OpenGL Multipipe product. New features: - Enhanced Support for Multithreaded Applications - Enhanced tgl Script
006	November 2001 Updated for release 1.4 of the OpenGL Multipipe product.

Bugfixes:

- Enhanced GLX conformance for context manipulation

- Support for pixmaps, pbuffers, and GLXWindows

Beta features:

- Curved Screen Support

This allows you to run applications on a non-planar Reality Center in immersive mode by adapting the 3D projections to the display layout.

- Window Manager Support for Aware Windows

All applications started in aware mode can now be under window manager control by using the customized window manager included with this release.

007 February 2002

Updated for release 1.4.1 of the OpenGL Multipipe product.

Broader application support

Bugfixes:

- Enhanced OpenGL conformance for applications using **glCallList()** within another display list

- Stability improvements to (beta) aware window manager

008 April 2002

Updated for release 1.4.2 of the OpenGL Multipipe product.

- Broader application support

- Stability improvements to (beta) aware window manager

009 October 2002

Updated for release 2.1 of the OpenGL Multipipe product.

Features:

Replacement of the Transparent OpenGL (TGL) layer with a proxy render library and render servers

Support for additional servers. The list of supported servers now includes the following:

- Silicon Graphics Onyx

- Silicon Graphics Onyx2

- SGI Onyx 3000, InfiniteReality

- SGI Onyx 3000, InfinitePerformance
- Silicon Graphics Octane2

010 May 2003

Updated for release 2.1.2 of the OpenGL Multipipe product.

Features:

- Performance enhancements over the OpenGL Multipipe 2.1.1 and 2.1 releases
- Increased application compatibility for vertex array applications
- Option of running in master render mode or slave-only mode
- Support for compositor-based systems
- Seamless cursor movement across overlapped or composited screen regions (IRIX 6.5.20 or later required)
- Support for SGI-SCREEN-CAPTURE and ReadDisplay X extensions in SGI Xinerama mode (IRIX 6.5.20 or later required)
- An API introduced for integration of multipipe applications with SGI Xinerama
- Hardware swap-synchronization option (Swap Ready, Genlock)
- Support for additional platforms. The list of supported visualization systems now includes the following:
  - o SGI Onyx 3000 series with InfinitePerformance graphics
  - o SGI Onyx 3000 series with InfiniteReality graphics
  - o SGI Onyx 350
  - o Silicon Graphics Octane2
  - o Silicon Graphics Onyx
  - o Silicon Graphics Onyx2

011 July 2003

Updated for release 2.2 of the OpenGL Multipipe product.

Features:

- Performance enhancements over the 2.1.2 release including the following:
  - o Pixel drawing enhancements
  - o Geometry culling, which can improve application performance for some applications beyond what can be achieved on a single pipe
- Support for the DMX (Distributed Multihead X) meta display server

- Support for DMX and SGI Xinerama meta display servers by the MPC API for multipipe-aware applications
- Support for additional platforms. The list of supported visualization systems now includes the following:
  - o SGI Onyx 3000 series with InfinitePerformance graphics
  - o SGI Onyx 3000 series with InfiniteReality graphics
  - o SGI Onyx 350
  - o Silicon Graphics Octane2
  - o Silicon Graphics Onyx
  - o Silicon Graphics Onyx2
  - o Silicon Graphics Onyx4 UltimateVision

012 December 2003

Updated for release 2.3 of the OpenGL Multipipe product.

Features:

- Performance enhancement options including (disabled by default):
  - o Geometry culling to optional, additional OpenGL clip planes
  - o OpenGL viewport clipping and geometry culling to improve fill and geometry performance when using a compositor
  - o Spatial partitioning of large display lists for efficient geometry culling
- Baseline performance enhancements over OpenGL Multipipe 2.2
  - o Swap synchronization and frame latency control improvements between slave rendering processes
  - o Improved performance when using geometry culling
- Support for Onyx4 OpenGL extensions
- Support for additional platforms. The list of supported visualization systems now includes the following:
  - o SGI Onyx 3000 series with InfinitePerformance graphics
  - o SGI Onyx 3000 series with InfiniteReality graphics
  - o SGI Onyx 350
  - o Silicon Graphics Octane2
  - o Silicon Graphics Onyx
  - o Silicon Graphics Onyx2
  - o Silicon Graphics Onyx4 UltimateVision
  - o Silicon Graphics Tezro

013 April 2004

Updated for release 2.3.1 of the OpenGL Multipipe product.

Enhancements over OpenGL Multipipe 2.3:

- Capability to expand the OpenGL viewport size beyond the single-pipe limit
- Support for overlay visuals in the DMX proxy X server if supported by underlying X servers
- Support for applications that use GLX pbuffers or GLX pixmaps
- Improved application compatibility

014 October 2004

Updated for release 2.4 of the OpenGL Multipipe product.

Enhancements over OpenGL Multipipe 2.3.1:

- Support for most OpenGL Multipipe features on Silicon Graphics Vizualization System for Linux 64-bit platforms
- Small object culling (small relative to screen space)
- Drawpixels culling
- Improved application compatibility

015 November 2004

Updated for release 2.4.1 of the OpenGL Multipipe product.

Enhancements over OpenGL Multipipe 2.4:

- Improved daemon processing
- Improved support for static tiling modes of hardware compositors
- Enhanced application compatibility

016 January 2005

Updated for release 2.5 of the OpenGL Multipipe product.

Enhancements over OpenGL Multipipe 2.4.1:

- Support for vertex buffer objects
- The optional use of the omprun command on selected platforms
- The use of X11 resources to control performance features as an alternative to command-line options on omprun
- Improved application stability

017 June 2005

Updated for release 2.5.2 of the OpenGL Multipipe product.

Enhancements over OpenGL Multipipe 2.5.1:

- Support for compositor pixel averaging to provide transparent antialiasing
- Support for multiple compositors
- **glDrawPixels()** clipping now enabled by default (not tied to geometry culling)
- Improved application stability and bugfixes

018 January 2006

Updated for releases 2.5.3 and 2.5.4 of the OpenGL Multipipe product.

2.5.3 features:

- Improved immediate mode performance
- Improved performance for downloading pixel data
- Preservation of triangle strips when spatially splitting a display list
- Better control over the number of splits done by `dlSplit`

2.5.4 features:

- Two performance monitoring tools: `ompmon` and a Performance Co-Pilot data collection agent
- Support for context sharing between threads
- Time-Based decomposition using hardware compositors
- Memory placement and use of the `tmpfs` filesystem
- Support for the OpenGL Shading Language and the ARB_texture_rectangle extension
- Slave synchronization on **glFlush()** for single-buffer windows

Contents

Figures

Tables

About This Guide

This guide describes the OpenGL Multipipe product, which allows you to run single-pipe applications in a multipipe environment without modification. You can seamlessly move single-pipe application windows across the single logical display that OpenGL Multipipe creates from multiple pipes. Both multipipe applications and single-pipe applications run concurrently.

Related Publications

The following SGI documents contain additional information that may be helpful:

- *SGI Scalable Graphics Compositor User's Guide*
- *SGI ProPack for Linux Start Here*
- *Performance Co-Pilot for IA-64 Linux User's and Administrator's Guide*

These books might also be helpful:

- Dave Shreiner, OpenGL Architecture Review Board, Mason Woo, Jackie Neider and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*. Reading, MA: Addison Wesley Longman Inc., 2003. ISBN 0-321-17348-1.
- Nye, Adrian, *Volume One: Xlib Programming Manual*. Sebastopol, California: O'Reilly & Associates, Inc., 1992.

Obtaining Publications

You can obtain SGI documentation in the following way:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- You can also view man pages by typing man< *title*> on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
interface	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists. Functions are also denoted in bold with following parentheses.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
Right angle brackets (>)	These brackets indicate a path through menus to a menu option. For example, "**File > Open**" means "Under the **File** menu, choose the **Open** option."

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, CA 94043-1351

SGI values your comments and will respond to them promptly.

OpenGL Multipipe Overview

This overview of OpenGL Multipipe consists of the following sections:

- "What OpenGL Multipipe Provides"
- "Architecture of OpenGL Multipipe"
- "Components of OpenGL Multipipe"
- "Supported Platforms"

What OpenGL Multipipe Provides

SGI has always been focused on high-end graphics solutions. The Silicon Graphics Prism family of scalable visualization systems allows you to have multiple graphics pipes on one single-system-image machine in order to reach new visualization performances. These multipipe systems are commonly used to drive expanded visualization systems such as SGI Reality Center facilities. OpenGL Multipipe extends the use of these powerful supercomputers to a broad spectrum of graphics applications without the requirement of modifying the applications.

Many existing graphics applications—such as Netscape or applications based on Open Inventor, for example—are constrained to run on a single pipe. On these single-pipe applications, you can choose the pipe on which to open the application's windows, but the windows cannot be dragged from one pipe to another. The main reason is that the graphics pipes are separate logical units and are handled by an X server as different, unconnected screens. This means that the X server does not provide any functionality to group multiple screens into a single logical display. A second reason is that OpenGL applications connect directly to a specified graphics pipe and bypass the X protocol layer.

In the past, displaying an application on multiple screens required you to explicitly write the application for that purpose. You had to use tools like the OpenGL Performer or OpenGL Multipipe SDK libraries to help you create these multipipe applications. These tools allow you to explicitly open windows on different screens and to draw into them

using OpenGL. However, this solution lacks consistency. In fact, all of the windows on the different pipes are independent; hence, moving or iconifying one window on one screen will not handle the other windows accordingly.

OpenGL Multipipe has been designed to overcome these difficulties. The goal is to group pipes managed by the X server in order to create a consistent, single virtual screen as shown in Figure 1-1. This means that the applications are unaware of the underlying hardware configuration. Rather, they only know about a single display and behave accordingly.

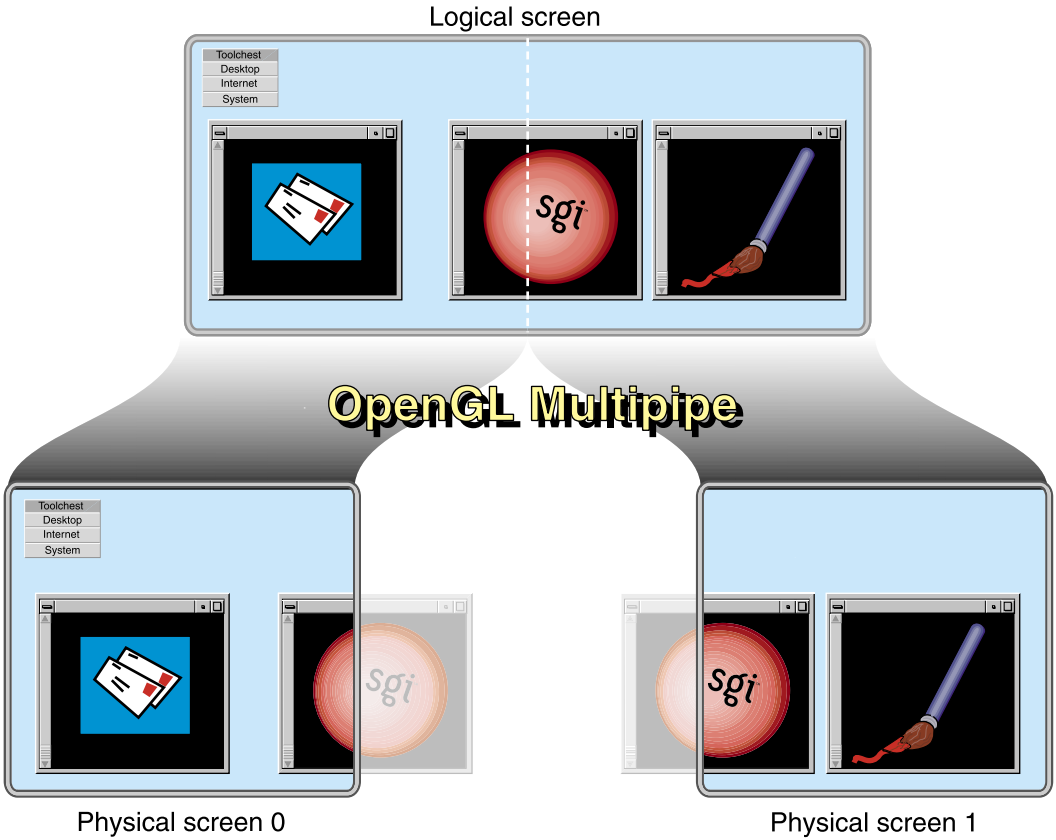

Figure 1-1 OpenGL Multipipe with Non-Overlapping Screens

In contrast to Figure 1-1, if you have screens that overlap each other (such as in an SGI Reality Center wall display with edge blending), OpenGL Multipipe allows you to

specify the amount of this overlap. Figure 1-2 shows the image blended on overlapping screens.

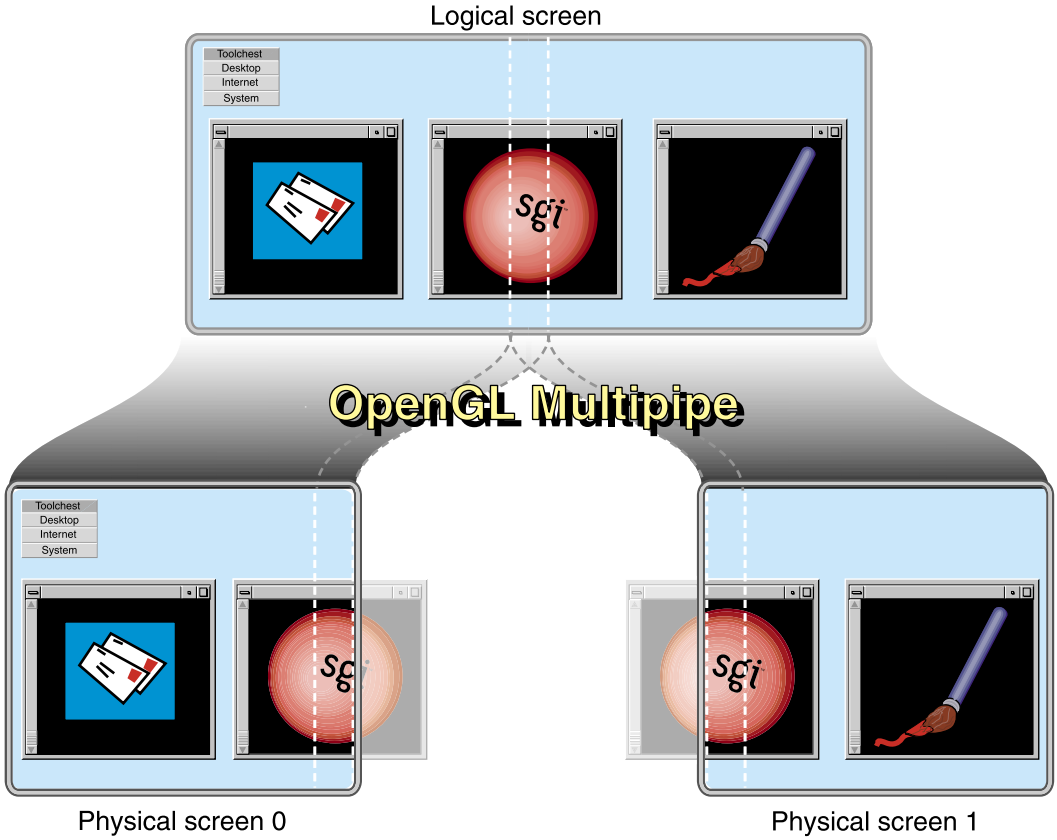

Figure 1-2 OpenGL Multipipe with Overlapping Screens

Note: OpenGL Multipipe does not require you to modify or recompile your application.

Architecture of OpenGL Multipipe

OpenGL Multipipe provides the illusion that an application is rendering 2D (X perspective) and 3D (OpenGL perspective) on a single local pipe when it is actually using one or more pipes. This illusion, or logical display, is created by a set of protocols and proxies coupled with clients and servers, all of which are hidden from the application.

OpenGL Multipipe uses an X proxy layer to hide the physical screen layout from the application. This X proxy layer presents a single logical pipe or meta screen to all applications and allows their windows to be freely moved across or to span any set of pipes. OpenGL Multipipe uses the Distributed Multihead X (DMX) proxy server.

OpenGL Multipipe also uses an OpenGL proxy library and render servers to send OpenGL calls to each real pipe. Having 3D render servers separate from the 3D proxy library allows the application processing and the rendering to occur in separate processes. This separation aides application compatibility.

Components of OpenGL Multipipe

OpenGL Multipipe has the following components:

- An X proxy layer (the DMX proxy server)
- A session manager process (`ompmgr`)
- A 3D proxy render library
- An optional culling process (`ompcull`)
- 3D render servers

The X Proxy Layer (the DMX Proxy Server)

For pure X applications—that is, applications that do not use other graphics libraries (such as OpenGL) to draw into their windows—the X proxy layer (DMX) is all that is needed to enable such applications to run transparently over multiple pipes. This means that windows of applications that are based on the X protocol and that use X extensions can be dragged from one pipe to another and even span multiple pipes. The applications behave as if they are running on a single, large virtual pipe. The X proxy layer hides the real screens from the client applications connecting to it. It distributes to all pipes the X

requests from the clients but only sends the clients information about the large virtual display.

The DMX proxy server is a separate entity apart from the X server; it is an X application that behaves like an X server to other X applications. DMX is flexible both in regards to its supported display geometries and in its ability to act as a proxy for many different X servers. DMX also has built-in support for OpenGL applications through its support of the GLX X extension. This means that DMX will enable X and OpenGL applications to run transparently across multiple pipes. However, DMX's GLX extension is limited in performance. Hence, it is best to run graphics-intensive applications under the full OpenGL Multipipe environment.

Administrative privileges are not required to start and stop DMX. For more information about the DMX proxy server, see the `Xdmx`(1) man page, which is installed in `/usr/share/omp/doc/user/Xdmx.1.html`.

The Session Manager Process (`ompmgr`)

The session manager process (`ompmgr`) is used to manage and update special resources that are used by the 3D render library and render servers. It maintains structures in shared memory through which all the other components (the render servers, proxy library, and optional ompcull process) can communicate and share global session information. The followings tasks are handled by `ompmgr`:

- Starting the DMX X proxy server during session initialization
- Configuring hardware compositors during session initialization if they are being used
- Maintaining information about all OpenGL windows in the session and updating the window(s) position/size when needed
- Maintaining information about all currently running OpenGL applications that uses OpenGL Multipipe in the session
- Handling hardware compositor re-configuration when using one of the dynamic composition schemes (dynamic load-balanced tiling or time-based composition)

The 3D (Master) Proxy Render Library

OpenGL applications are X applications that use another graphics library (namely the OpenGL library) to draw into their windows. OpenGL applications open a direct connection to a graphics pipe. This means that the application is bypassing the X protocol (and the X proxy layer, which replicates the the X protocol stream to each pipe) in order to draw in the windows through this direct connection. The X proxy layer, which accounts only for the X protocol, is unable to handle this case.

The *master* proxy library in OpenGL Multipipe handles the OpenGL side of any application. It intercepts OpenGL calls to enable distribution to multiple pipes. To distribute the OpenGL calls, the master library encodes each command using an OpenGL stream protocol and sends the command stream to the *slave* renderer processes, which render to local pipes on behalf of the application.

Figure 1-3 illustrates the functions of the master proxy library.

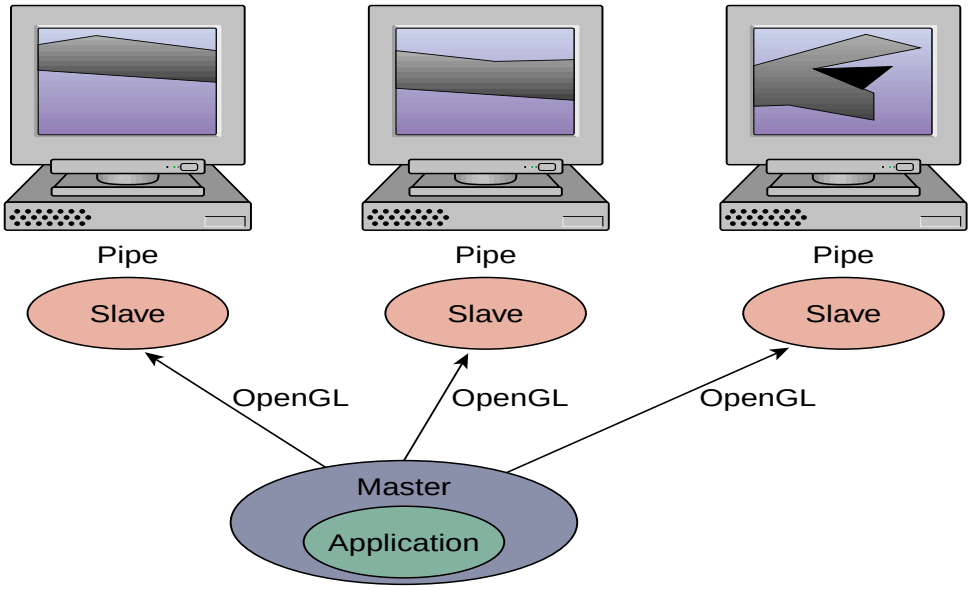

Figure 1-3 Master Proxy Library Functions

In addition to sending an OpenGL stream to each slave, the master also has the capability of rendering directly to a single local pipe in place of a single slave render process (for faster GL state queries), or it may use a local pipe only to track OpenGL state while a slave process renders to that pipe (for improved parallelism). For more information about performance and the master library modes, see Chapter 5, “Optimizing Performance”.

The Culling Process (`ompcull`)

The optional culling process (`ompcull`) will be activated only if the geometry culling feature is turned on. This process reads the OpenGL stream encoded by the master proxy library before it reaches the rendering slaves and modifies each 3D bounding box found in the stream with its projected 2D screen coordinate bounding box, which will then be used by the rendering slave to decide whether to render the followed geometry or not. The purpose of doing that calculation on a separate process is to increase parallelism and throughput by not “stealing” CPU cycles from either the application or rendering slaves on configurations making use of multiple CPUs.

3D (Slave) Render Servers

An application running under the master render library of OpenGL Multipipe communicates its OpenGL commands to slave renderer processes. Each slave process parses the OpenGL command stream and executes the commands on the application's behalf. For each rendering application thread of execution, one slave process exists for each screen of the display.

Supported Platforms

OpenGL Multipipe 2.5.4 runs on Silicon Graphics Prism visualization servers using SGI ProPack 4 Service Pack 3.

Installing OpenGL Multipipe

Using the following two topics, this chapter describes how you install OpenGL Multipipe:

General Installation

The following are the prerequisites for installing OpenGL Multipipe on your system:

Hardware	Silicon Graphics Prism visualization system
Software	SGI ProPack 4 Service Pack 3

OpenGL Multipipe for Silicon Graphics Prism systems is installed by default when installing the SGI ProPack package. OpenGL Multipipe contains the following file subsystem:

`sgi-omp-2.`*`x-yyyyy`*`.ia64.rpm`

When installed with SGI ProPack 4, the `xinetd` service is disabled by default. The service has to be enabled and started in order for OpenGL Multipipe to function. In order to start the service, the following commands need to be executed as a superuser once only after SGI ProPack installation:

```
$ /sbin/chkconfig xinetd on
$ /etc/init.d/xinetd start
```

When trying to run an application under an OpenGL Multipipe session while `xinetd` is disabled, the following error message will be printed:

```
"Could not connect to sgi-ompslave service."
```

Detailed documentation of the `rpm` utility for managing software installation on Linux is available in the `rpm`(8) man page.

For any critical updated information, you can check the release notes, `/usr/share/omp/release_notes/user/relnotes.html`, and the OpenGL Multipipe website, http://www.sgi.com/software/multipipe/ .

Installing the Data Collection Agent for Performance Co-Pilot

As part of OpenGL Multipipe installation, the OpenGL Multipipe Performance Metrics Domain Agent (PMDA) is placed in the directory `/var/lib/pcp/multipipe`. This PMDA needs to be installed after the initial installation of the OpenGL Multipipe RPM or if the hardware configuration of the machine changes. You install the OpenGL Multipipe PMDA by running the following script as a superuser:

```
/var/lib/pcp/pmdas/multipipe/Install
```

Note: This script should be executed only after Performance Co-Pilot is properly installed and configured. For Performance Co-Pilot installation and configuration, refer to the *Performance Co-Pilot for IA-64 Linux User's and Administrator's Guide*.

The following session illustrates the installation of the PDMA:

```
keglevich:/var/lib/pcp/pmdas/multipipe # ./Install

You need to choose an appropriate configuration for installation of
the "multipipe" Performance Metrics Domain Agent (PMDA).

  collector     collect performance statistics on this system
  monitor       allow system to monitor local and/or remote systems
  both          collector and monitor configuration for this system

Please enter c(ollector) or m(onitor) or b(oth) b

Updating the Performance Metrics Name Space (PMNS) ...
Compiled PMNS contains
          167 hash table entries
          723 leaf nodes
          105 non-leaf nodes
         7835 bytes of symbol table
```

```
Installing pmchart view(s) ...
PMCD should communicate with the multipipe daemon via a pipe or a
socket? [pipe]
Terminate PMDA if already installed ...
Installing files ...
gmake: Nothing to be done for `install'.
Updating the PMCD control file, and notifying PMCD ...
Check multipipe metrics have appeared ... 73 warnings, 73 metrics and 0
values
keglevich:/var/lib/pcp/pmdas/multipipe #
```

Once the PMDA is installed, you can verify the installation by running the `pminfo` command. The following session show the typical output:

```
keglevich:/var/lib/pcp/pmdas/multipipe # pminfo | grep multipipe
multipipe.master.nframes
multipipe.master.frame_time
multipipe.master.frame_rate
multipipe.master.draw_time
multipipe.master.draw_rate
multipipe.master.encoded_bytes
multipipe.master.encoding_rate
multipipe.master.wait_write_queue
multipipe.master.wait_write_queue_p
multipipe.master.wait_state_query
multipipe.master.wait_state_query_p
multipipe.master.wait_EOF_sync
multipipe.master.wait_EOF_p
multipipe.slave0.nframes
multipipe.slave0.frame_time
multipipe.slave0.frame_rate
multipipe.slave0.draw_time
multipipe.slave0.draw_rate
multipipe.slave0.wait_read_queue
multipipe.slave0.wait_read_queue_p
multipipe.slave0.wait_sync
multipipe.slave0.wait_sync_p
multipipe.slave0.direct_vertices
multipipe.slave0.direct_vertices_rate
multipipe.slave0.direct_culled_vertices
multipipe.slave0.direct_culled_vertices_rate
multipipe.slave0.direct_rendered_vertices
multipipe.slave0.direct_rendered_vertices_rate
multipipe.slave0.direct_bbox_cmd
multipipe.slave0.retained_vertices
```

```
multipipe.slave0.retained_vertices_rate
multipipe.slave0.retained_culled_vertices
multipipe.slave0.retained_culled_vertices_rate
multipipe.slave0.retained_rendered_vertices
multipipe.slave0.retained_rendered_vertices_rate
multipipe.slave0.retained_bbox_cmd
multipipe.slave0.total_vertices
multipipe.slave0.total_vertices_rate
multipipe.slave0.total_culled_vertices
multipipe.slave0.total_culled_vertices_rate
multipipe.slave0.total_rendered_vertices
multipipe.slave0.total_rendered_vertices_rate
multipipe.slave0.total_bbox_cmd
multipipe.slave1.nframes
multipipe.slave1.frame_time
multipipe.slave1.frame_rate
multipipe.slave1.draw_time
multipipe.slave1.draw_rate
multipipe.slave1.wait_read_queue
multipipe.slave1.wait_read_queue_p
multipipe.slave1.wait_sync
multipipe.slave1.wait_sync_p
multipipe.slave1.direct_vertices
multipipe.slave1.direct_vertices_rate
multipipe.slave1.direct_culled_vertices
multipipe.slave1.direct_culled_vertices_rate
multipipe.slave1.direct_rendered_vertices
multipipe.slave1.direct_rendered_vertices_rate
multipipe.slave1.direct_bbox_cmd
multipipe.slave1.retained_vertices
multipipe.slave1.retained_vertices_rate
multipipe.slave1.retained_culled_vertices
multipipe.slave1.retained_culled_vertices_rate
multipipe.slave1.retained_rendered_vertices
multipipe.slave1.retained_rendered_vertices_rate
multipipe.slave1.retained_bbox_cmd
multipipe.slave1.total_vertices
multipipe.slave1.total_vertices_rate
multipipe.slave1.total_culled_vertices
multipipe.slave1.total_culled_vertices_rate
multipipe.slave1.total_rendered_vertices
multipipe.slave1.total_rendered_vertices_rate
multipipe.slave1.total_bbox_cmd
keglevich:/var/lib/pcp/pmdas/multipipe #
```

Using OpenGL Multipipe

As described in Chapter 1, OpenGL Multipipe consists of three main components: an X proxy layer, a proxy 3D render library, and 3D render servers. This chapter describes how to effectively use these components with your graphics applications. The following sections describe the pertinent tasks:

For information about other features of OpenGL Multipipe specific to this release, see the release notes in the following file:

```
/usr/share/omp/release_notes/user/relnotes.html
```

Setting up the OpenGL Multipipe Environment

To begin using OpenGL Multipipe, you must enable an X proxy layer. This will cause all applications to see a single logical pipe. To deactivate OpenGL Multipipe, just disable the X proxy layer. Some of the steps required to enable or disable OpenGL Multipipe may require `root` access. This section notes this requirement in the applicable steps.

This section describes the following tasks:

Configuring OpenGL Multipipe with DMX as the X Proxy Layer

DMX will group multiple screens into a logical display. This section describes how you initialize DMX and how to create DMX configuration files.

Initializing DMX

To initialize DMX, do the following:

1. Run DMX on top of the existing X server(s).

   You may do this manually after logging into your desktop or you may configure an `.xsession` script to run DMX immediately upon login.

   To manually initialize DMX, enter the following (`root` access not needed) in a command shell:

   ```
   $ ompstartdmx
   ```

   You can use the flag –`help` for more information about the starting options. If you specify no flags, DMX starts on top of the existing X server and will configure a single large screen that overlays the existing *n* screens such that screen 0 will be the leftmost and screen *n* –1 will be the rightmost. To use a different configuration, such as a vertical configuration, you must provide a DMX configuration file. The following section "Creating DMX Configuration Files"describes how to create such a configuration file.

Only the client `xterm` will be started as the session. For more details, see the later subsection "Notes About the Behavior of `ompstartdmx`".

Configuring DMX to Run Automatically on Linux

This following describes how you configure DMX to run automatically upon login using GDM or KDM.

At the login screen, choose one of the following from the **Session** menu:

- **GNOME-OpenGL-Multipipe**
- **KDE-OpenGL-Multipipe**

Choosing one of these will start DMX with GNOME or KDE, respectively. The session will use the default DMX configuration. If you want to customize the `ompstartdmx` options (for example, to use a special DMX configuration file), modify the scripts in the following directory corresponding to your choice of window manager:

- `/usr/share/omp/X11/bin/GNOME-OpenGL-Multipipe`
- `/usr/share/omp/X11/bin/KDE-OpenGL-Multipipe`

If you want to add another new session script, you should add a new desktop session configuration file in the following directory corresponding to your choice of window manager:

- `/opt/kde3/share/apps/kdm/sessions`
- `/opt/gnome/share/xsessions`

You can copy and change one of the `sgiomp_gnome.desktop` or `sgiomp_kde.desktop` files in that directory.

After DMX has initialized, you will see a new session covering all the screens. At this point, you can start using OpenGL Multipipe (see "Running Applications with OpenGL Multipipe" on page 31).

Notes About the Behavior of `ompstartdmx`

This section lists some noteworthy items about the behavior and command-line options of the `ompstartdmx` script:

- The window manager is the default window manager for GNOME or KDE. Use their standard tools to set your preferred window manager.
- The `ompstartdmx` script will return control only when the session ends.

- The valid options for the –`session` flag are as follows:

  `[ -session {gnome | kde | noaware | `*scriptname*`} ]`

Option	Behavior
–`session` unspecified	Starts a single xterm as the session. The session ends when this xterm dies.
`gnome`	Starts a GNOME session with aware support if GNOME is installed; otherwise, xterm will be used.
`kde`	Starts a KDE session with aware support if KDE is installed; othersise, xterm will be used.
`noaware`	Launches **without** aware window management the default session that would run with the `startx` command.
scriptname	Runs the script as the session. When the script returns control, the session ends. **Note:**The `ompstartdmx` script does not launch any window manager; the session script must do so.

- Use `logout` or `Ctrl-Alt-q` to end the session.

DMX Limitations on Linux

When using a GNOME session with an aware window manager, the session will not be saved; the default session will be used at each login. This happens because the `gnome-wm` script does not pass the session ID parameter to any aware window manager, only to those which are supported by the GNOME environment (`sawfish`, `metacity`, `twm`, and their like). When using a KDE or GNOME session without aware window management, the session will be saved and restord at the next login.

Creating DMX Configuration Files

A DMX configuration file is simply a text file that describes the configuration of a virtual display, the real displays it manages, and the geometry of the virtual screen. This section provides some short examples of configuration files. These and other example configuration files may be found in the directory `/usr/share/omp/examples/dmx` .

To start DMX with one of these configurations, do the following:

1. Save the configuration to a text file with any name—for example, `updown.dmx`.
2. Invoke `ompstartdmx` with the option –`cfgfile`, as shown in the following entry:

   ```
   $ ompstartdmx -cfgfile updown.dmx
   ```

   It is also possible to place many configurations in a single file. In this case, you can choose one configuration from the file by specifying both the –`cfgfile` and –`cfgname` options, as shown in the following:

   ```
   $ ompstartdmx -cfgfile allmyconfigs.dmx -cfgname updown
   ```

The following example configuration file specifies a vertical layout:

```
virtual updown 1280x2048 {
    display :0.0 1280x1024;
    display :0.1 1280x1024 @0x1024;
}
```

This configuration file defines a virtual screen configuration named `updown` of size 1280x2048. The virtual screen includes the following two real back-end displays:

Display `:0.0` It has a size of `1280x1024` and is located at location `0x0` in the virtual screen space.

Display `:0.1` It has a size of `1280x1024` and is located at `0x1024` in the virtual screen space.

You may also define some overlap between each of the screens, as in the following horizontal layout:

```
virtual overlap 2460x1024 {
    display :0.0 1280x1024;
    display :0.1 1280x1024 @1180x0;
}
```

This configuration file defines two screens of `1280x1024`, each with 100 pixels of overlap, resulting in a virtual screen size of `2460x1024`.

The display value specified can be any valid display value, including a display value that specifies a remote machine, as in the following example:

```
virtual remote 2560x1024 {
    display localhost:0.0 1280x1024;
    display remotehost:0.1 1280x1024 @1280x0;
}
```

There is also a graphical tool to create and edit configuration files. You can find documentation for this tool in `/usr/share/omp/doc/user/xdmxconfig.1.html`. The tool is installed in `/usr/share/omp/X11/bin/xdmxconfig`.

More information about the configuration file format can be found in the file `/usr/share/omp/doc/user/Xdmx.1.html`.

Setting Other Configuration Options

Starting with version 2.5, OpenGL Multipipe uses X11 resources to specify configuration options. This approach allows you to define values per application, per user, and system-wide. This section describes setting these options in the following subsections:

- "Specifying Resource Names" on page 18
- "The Resource Search Path" on page 19
- "Resource Types" on page 19
- "Resources and Their Default Values" on page 20
- "Resource Descriptions" on page 22

Specifying Resource Names

The full name of each OpenGL Multipipe resource has the following format:

`OMP.`*app_name*`.`*resource_name*

The *app_name* field is the application name, defined to be the last component in the running executable path name. The *resource_name* is one of the available OpenGL Multipipe resources.

You can specify a resource with *loose binding*—that is, matching the resource for all applications—by using an asterisk in the following manner:

`OMP*`*resource_name*

For more information on the X11 resources and their scope, see the `X`(1) and `XrmInitialize`(3) man pages.

The Resource Search Path

The following ordered list of resource files indicates the search path OpenGL Multipipe uses to assign values to the resources:

1. Resources set with `omprun`
2. Resources set with `xrdb`
3. `$HOME/.Xdefaults`
4. `/usr/share/omp/app-defaults/app_name`
5. `/usr/share/omp/config`

The first match found will be used. However, resources set with the full-name format will always precede resources set with loose binding even if the full-name format is found later in the search path. Resources set with the `omprun` command always have precedence. For information of the use of the `omprun` command, see section "Setting Run-Time Options" on page 32.

If a resource is not specified in the preceding search path, then the defined default value for that resource will be used.

When you set the environment variable `SGIOMP_PRINT_CONFIG` to `1`, OpenGL Multipipe prints the configuration values it uses to `stdout` when an application runs through OpenGL Multipipe. This is useful to ensure that any overrides you have specified are being honored.

Resource Types

Each resource value can have one of the following types:

Type	Description
`Bool`	A boolean value. Valid values are `0`, `1`, `off`, or `on`.
`Enum`	Enumeration: one of a specified list of tokens.
`Float`	A floating point value.
`FloatList`	A comma-separated list of floating point values.
`Int`	An integer number.
`IntList`	A comma-separated list of integer values.

`Enum` An enumeration: one of a specified list of tokens.

Resources and Their Default Values

Table 3-1 lists all resources with their default values.

Notes:

- A resource name is not identical to the name of the associated command-line option of `omprun`.
- The default values might be changed in future releases.

Table 3-1 X11 Resources and Defaults

Resource[a]	Value Type	Default Value
`aa2Jitter.X`	`FloatList`	`0.24649, -0.24649`
`aa2Jitter.Y`	`FloatList`	`0.249999, -0.249999`
`aa4Jitter.X`	`FloatList`	`-0.208147, 0.203849, -0.292626, 0.296924`
`aa4Jitter.Y`	`FloatList`	`0.353730, -0.353780, -0.149945, 0.149994`
`activeScreens`	`IntList`	The list of all back-end screens of the meta X server
`culling` (`cull`)	`Bool`	`off`
`culling.cullUserClipPlanes`	`Bool`	`off`
`culling.minPixels` (`minpixels`)	`Int`	`1`
`culling.showStat` (`cullshow`)	`Enum:` `off` `bbox` `stats` `all`	`off`
`culling.texCulling`	`Bool`	`off`

Table 3-1 X11 Resources and Defaults **(continued)**

Resource[a]	Value Type	Default Value
dlSplit (dlsplit)	Bool	off
dlSplit.maxBoundAspectRatio	Int	8
dlSplit.maxDepth	Int	8
dlSplit.maxStripLen	Int	-1
dlSplit.maxTotalTris	Int	400000
dlSplit.maxTris (dlsplitmaxtris)	Int	800
dlSplit.showRandomColors (dlsplitshow)	Bool	off
drawPixelsClipping	Bool	on
masterMode (mstrmode)	Enum: render track off	track when omprun is used off, otherwise
masterScreen	Int	0
maxFramesLatency (latency)	Int	4
pbuffers.disable	Bool	off
pbuffers.layout	Enum: duplicate horizSplit vertSplit rectSplit	duplicate
shmQueueSize	Int	DisplayWidth * DisplayHeight * 4
slaveCPUs	IntList	Empty

Table 3-1 X11 Resources and Defaults **(continued)**

Resource[a]	Value Type	Default Value

`swapSyncMode`
`(nosync,swapready)`
	`Enum:`
	`none`
	`soft`
	`barrier`
		`soft`

`syncOnFlushMode`	`Enum:`
	`never`
	`always`
	`frontBuffer`
		`frontBuffer`

`texShm`	`Bool`	`off` (unless `culling.texCulling` is on)
`useTmpfs`	`Bool`	`on`

`useTmpfs.shmPlacement`	`Enum:`
	`master`
	`slaves`
	`all_omp`
	`global`
	`os`
		`os`

`viewportClippingMode`
`(novpclip)`
	`Enum:`
	`none`
	`scissor`
	`viewport`
	`subwin`
		`subwin` for `Xdmx` displays

a. The associated `omprun` command-line options are shown in parentheses, where applicable.

Resource Descriptions

This section briefly describes each of the OpenGL Multipipe tunable resources:

`aa2Jitter.X` and `aa2Jitter.Y`
Controls the jittering offset for each input pipe when the hardware compositor configured for pixel averaging with two pipes. The value must include two values for both X and Y to specify the frustum offset in pixels on the x and y axes to be applied for the first and second pipe participating in the pixel averaging.

`aa4Jitter.X` and `aa4Jitter.Y`
Controls the jittering offset for each input pipe when the hardware compositor configured for pixel averaging with four pipes. The value must include four values for both X and Y to specify the frustum offset in pixels on the x and y axes to be applied for the first, second, third, and fourth pipes participating in the pixel averaging.

`activeScreens`
Specifies the back-end screens to be active. OpenGL Multipipe will render OpenGL primitives only on those pipes. For example, when `Xdmx` manages three physical screens (pipes), setting the `activeScreens` resource to `0,1` will make OpenGL rendering to be visible on screens 0 and 1 only. On screens 2 and 3, the OpenGL part of the application windows will be either black or garbage. By default, all managed screens will be active with respect to OpenGL.

`culling`
Specifies the enable switch for the geometry culling feature of OpenGL Multipipe. It can be either `off` or `on`. When this resource is enabled OpenGL Multipipe will draw to each pipe only the geometry primitives that are visible on that pipe. The default value for this switch is `off`. For more information, see section "Geometry Culling" on page 64.

`culling.cullUserClipPlanes`
Enables culling against user-defined clip planes when one of `GL_CLIP_PLANE`*i* is enabled. The default value is `off`, in which case OpenGL Multipipe will cull only against the viewing frustum.

`culling.minPixels`
Specifies approximate culling—that is, the minimum object size in pixels that should be drawn. All objects smaller than this size will be culled. For more information, see section "Small Object Culling" on page 65.

`culling.showStat`
Enables the drawing of culling statistics. The following are the possible values:

`off`	No statistics are drawn (the default).
`bbox`	The bounding box for each geometry object is drawn in blue.
`stats`	Cull percentage statistics are drawn for each screen (only for doubly buffered windows).
`all`	Both bounding boxes and culling percentage statistics are drawn.

`culling.texCulling`
Enables or disables texture culling. When texture culling is enabled, texture loads will be postponed until the texture is really viewable on each specified pipe. In that case only, the relevent part from the texture will be downloaded to the pipe. The resource `OMP*culling` should be turned on for this feature. For more information and current limitations of this feature, see the release notes.

`dlSplit`
Allows OpenGL Multipipe to split a display list into smaller display lists based on a geometry spatial sort to acheive better culling. The smaller display lists can be distributed among the rendering pipes. This feature is usually used when culling is turned on. The valid values for `dlSplit` are `on` and `off`; the default is `off`. For more information, see section "Display List Partitioning" on page 65.

`dlSplit.maxBoundAspectRatio`
Controls the initial cube cell size and, hence, the initial number of grid divisions produced by the `dlSplit` algorithm. As part of the `dlSplit` algorithm, the display list geometry is spatially sorted into an oct tree data structure by iteratively dividing the geometry bound to the grid of cube cells. Normally, the cell size is taken to be equal to the smallest bounding box dimension. However, for models where the aspect ratio of the bounding box is large (for example, 2D surfaces), the algorithm would produce a large number of initial cells, which would lead to a high-memory footprint. The default value is `8`.

Note that setting `maxBoundAspectRatio` to a value less than or equal to `1.0` suppresses the splitting operation. Negative values are not allowed.

`dlSplit.maxDepth`
Limits the number of recursive subdivisions by the `dlSplit` algorithm, which is an adaptive subdivision algorithm. After the specified number of recursive subdivisions are made, the process will end even if it does not meet the `dlSplit.maxTris` criterion. The default value is `8`.

`dlSplit.maxStripLen`
Controls the amount of triangulation. By default with `dlSplit` enabled, geometry is triangulated into separate triangles before applying the splitting algorithm. This resource can be used to preserve the application-supplied strips so that fewer vertices are generated at the cost of less efficient splitting. When the value of this resource is set to `-1` (the default), all geometry will be triangulated before applying the `dlSplit` algorithm. When set to `0`, each strip given by the application will be considered atomic for splitting purposes and will not be triangulated. When set to a positive value, each

application-supplied strip will not be triangulated but will be split into multiple strips of `dlSplit.maxStripLen` vertices, of which each will then be considered atomic for splitting purposes.

`dlSplit.maxTotalTris`
Limits the number of triangles that will be considered by the `dlSplit` algorithm at each iteration. If a display list contains more triangles than the specified number, then the splitting algorithm will be applied multiple times, once for each group that contains the specified number of triangles. The default value is `400000`.

`dlSplit.maxTris`
Limits the number of triangles to the specified maximum for each bounding box of subordinate display lists. Display lists that have fewer triangles (or other primitives) than the specified number will not be split by the `dlSplit` algorithm. The default value is `800`.

`dlSplit.showRandomColors`
Allows OpenGL Multipipe to randomly assign a different color for each divided geometry partition. For testing and debugging purposes, this coloring allows you to better see the divisions made by the `dlSplit` algorithm. This may not work for all applications as it simply uses **glColor()** to set the color for each partition. The default is `off`.

`drawPixelsClipping`
Controls whether the **glDrawPixels()** operation will be clipped such that only the viewable rectangle of pixels will be sent to each pipe.

`masterMode`
Specifies if and how the master process needs to use one of the graphics pipes. The following are the valid values:

`render`	The master process will render to the master local pipe and no slave draw process will be forked for this pipe.
`track`	The master process will use its local pipe for state tracking only so that each **glGet()** call will be executed on the local pipe.
`off`	The master will not use any physical pipe for either rendering or state tracking. Some OpenGL states are being tracked by the master in software while other states will be queried from one of the slaves when a **glGet()** call is being executed.

For more detailed description of the different master modes, see section "Master Rendering Modes" on page 65.

When the `omprun` command is not being used on platforms where the `omprun` command is optional, the master mode is forced to be `off`, regardless of this resource value. Therefore, this resource is usually set only by the `omprun` command using its `–mstrmode` option.

`masterScreen`
Specifies which managed screen (pipe) will be used by the master application for either rendering or state tracking when running OpenGL Multipipe in master mode render or track (available only when using the `omprun` command). By default, it will be the first managed screen of the `Xdmx` server.

`maxFramesLatency`
Specifies the maximum latency (in frames) allowed between the application and rendering slaves. The application process might pack a few frames ahead in the framebuffer while the slave processes are still drawing the previous frames. This introduces some latency between the application and the real drawing by the slave but helps overall throughput of master and slaves. The latency is always 0 when using `masterMode` render and is undefined if `swapSyncMode` is set to `none`. The default value for this resource is 4. For more information, see section "Frame Latency Control" on page 69.

`pbuffers.disable`
Disables pbuffers support. When this resource is enabled, all pbuffer-capable framebuffer configurations will be filtered out and will not be exposed to the application. The default value is `off`.

`pbuffers.layout`
Controls how the rendering into a pbuffer is split among the rendering slaves. Each rendering slave can be configured to draw only to a sub-region of each pbuffer. The following are the valid values:

`duplicate`
All slaves renders to all regions of the pbuffer, That makes the content of the pbuffer to be duplicated on all rendering pipes. Use this configuration when your application uses **glCopy*()** to copy the pixels from the pbuffer to some other pixel buffer on the pipe. The other options might be better if you are using **glReadPixels()** to read back the pbuffer content into your process memory. This is the default value.

`horizSplit`
The area of the pbuffer is split into *N* horizontal strips, where *N* is the number of pipes. Each slave then limits the rendering to only one strip.

`vertSplit`
The area of the pbuffer is split into *N* vertical strips, where *N* is the number of pipes. Each slave then limits the rendering to only one strip.

`rectSplit`
The area of the pbuffer is split into *N* even rectangular areas as much as possible. For example, for a four-pipe configuration, it will be split into 2x2 tiles, where each tile is of size ($w/2$)x($h/2$) and w and h are the pbuffer width and height, respectively. Each pipe then renders to only one tile.

`shmQueueSize`
Sets the size in bytes for the shared memory block that will be used for master-to-slaves communication. When this resource is not specified, the default value depends on the overall resolution of the meta display as determined by the following:

`(DisplayWidth*DisplayHeight*4)` + 32

This allows a single full-screen **glDrawPixels()** command to fit in the shared memory since one single OpenGL command cannot be split into multiple packets. Depending on your application, you may want to set this value to be smaller or larger. When a shared memory size larger than the default or the specified size is needed, the application will exit with an error message specifiying the minimum shared memory size needed for that application.

`slaveCPUs`
Specifies the CPUs to assign to the draw slave processes for each of the active screens. If there are more active screens than the size of the specified CPU list, then the remaining draw processes will not be assigned to any particular CPU. By default, the draw processes will not be assigned to any particular CPU.

`swapSyncMode`
Specifies what method OpenGL Multipipe will use to synchronize swap buffers on all pipes. The following are the valid values:

`none`	No swap synchronization is being performed. Each rendering slave runs freely.

`soft`	Software synchronization is performed using a software barrier to force all slaves to issue the **glXSwapBuffers()** request at the same time.
`barrier`	OpenGL Multipipe uses the hardware ImageSync or Swap Ready line whenever possible (when no other application is using the hardware barrier). Otherwise, software synchronization will be used. For more information on the synchronization of swap buffers, see section "Buffer Swap Synchronization" on page 70.

`syncOnFlushMode`
Specifies if the rendering slaves need to be synchronized with each other after the execution of the **glFlush()** command. This is commonly required by single-buffer applications that never calls swap; this leaves the slaves asynchronous with each other. The following are the valid values for this resource:

never	Never synchronize the slaves after **glFlush()**.
always	Always synchronize the slaves after **glFlush()**.
frontBuffer	Synchronize the slaves after **glFlush()** only when it is operated on a single-buffer drawable or when drawing to the front buffer. (Default)

`texShm`
Controls the caching of textures in shared memory. When this switch is turned on, OpenGL Multipipe will cache a copy of each texture in shared memory that remains valid until the texture is used by all pipes. Once the texture has been downloaded to driver memory of all pipes, the shared memory cache is purged. This feature works together with texture culling (see the `culling.texCulling` resource). When `culling.texCulling` is turned on, the `texShm` resource is implicitly turned on. The default for the `texShm` resource is `off`.

`useTmpfs`
Allocates shared memory under the `tmpfs` filesystem (mounted under `/dev/shm`) rather than using `mmap` to map a file under the `/tmp` directory. Note that the `/dev/shm` filesystem needs to have enough space to allocate the shared memory. The default value is `on`.

`useTmpfs.shmPlacement`
Governs how OpenGL Multipipe allocates shared memory across a non-uniform memory access (NUMA) architecture. In a NUMA architecture, some memory areas have different latencies and bandwidths. This memory placement control requires that the `OMP*useTmpfs` resource will be set to `on`.

The following are the valid values for this resource:

`master`	Shared memory will be allocated on the application memory node.
`slaves`	Shared memory will be allocated on the slaves memory nodes in a round-robin fashion.
`all_omp`	Shared memory will be allocated on memory nodes used by OpenGL Multipipe processes (master and slaves) in a round-robin fashion.
`global`	Shared memory will be allocated across all the system memory nodes in a round-robin fashion.
`os`	Shared memory will be allocated according to the system default.

Note: If resource is set to `master`, `slaves`, or `all_omp`, you should place OpenGL Multipipe processes on specific CPUs using the `OMP*slaveCpus` resource. Otherwise, the behavior of this resource reverts to the default, that of `os`.

`viewportClippingMode`

Specifies viewport clipping mode. Each rendering slave may limit the rendering region to only the portion of the window that is visible on its rendering pipe. This is required to support large viewport areas, which are allowed by a single pipe, and it also helps to scale fill rate performance. The `viewportClippingMode` resource selects the method OpenGL Multipipe uses to limit this rendering region. The following are the valid values:

`none`	No viewport clipping is performed. Each hardware pipe clips the window to the pipe's boundary.
`scissor`	OpenGL Multipipe uses **glScissor()** to limit the rendering to the viewing area only.
`viewport`	OpenGL Multipipe defines a different viewport for each rendering slave.
`subwin`	OpenGL Multipipe creates a separate child window for each rendering slave. The window size is the viewable region of the application window on the slave's pipe. This option is supported on DMX configurations only. When this resource is not specified, the default depends on the meta X server that is being used. For `Xdmx` the default is `subwin`. For more information, see section "Viewport Clipping" on page 63.

Verifying That the OpenGL Multipipe Environment is Enabled

The OpenGL Multipipe environment is enabled if DMX is enabled.

To verify that DMX is enabled, ensure that your `DISPLAY` environment variable is pointing to the correct display (usually `:1.0`) and enter the following commands in a command shell:

```
$ xdpyinfo | grep DMX
```

If `DMX` appears as the result of the prior commands, DMX is enabled.

Note: If `DMX-DRI` appears as the result of the prior commands, not only is DMX enabled but the use of the `omprun` command will be optional as well. See section "Running Applications with OpenGL Multipipe" on page 31.

Disabling the OpenGL Multipipe Environment

To disable the OpenGL Multipipe environment, end a DMX session by simply logging out. You may also force the DMX server to exit by pressing `Ctrl+Alt+q`. If you ran `ompstartdmx` from the command line, you will be returned to your regular X session after the session ends.

If you chose DMX as a login option or modified your `.xsession` file to start your DMX session, you will return to the login screen after the DMX session ends.

To permanently disable DMX from starting upon login, reverse the DMX-related changes you made to your `.xession` file or delete (or rename) your `$HOME/.xsession` file. On Linux systems, simply select another default login option.

Running Applications with OpenGL Multipipe

Plain X applications will generally run under an X proxy layer without assistance. OpenGL (3D) applications need to use the OpenGL Multipipe 3D proxy library to handle 3D rendering correctly and efficiently on all screens.

To use the OpenGL Multipipe 3D proxy library with OpenGL applications, just run the program using the `omprun` script:

```
$ omprun app_name app_args
```

This will preload the OpenGL Multipipe proxy libraries to intercept OpenGL calls.

The following is an example:

```
$ omprun ivview /usr/share/data/models/Banana.iv
```

Note: The use of the `omprun` script is optional on Silicon Graphics Prism systems running the latest recommended operating system versions (SGI ProPack 4 Service Pack 3). On these platforms, you can invoke the application in the normal fashion and the application will automatically utilize the full OpenGL Multipipe environment (master-slave 3D proxy layer) if the DMX proxy layer is detected. Later references to the use of the `omprun` script in this guide should be considered optional on these platforms, unless otherwise noted.

OpenGL Multipipe causes an OpenGL application to use the intermediate 3D proxy libraries instead of the normal OpenGL library. This enables the OpenGL application to behave correctly when its windows are moved across parts of the logical screen. Such an application is considered to be started in multipipe-unaware mode (or simply, *unaware* mode), since it is not aware of the underlying graphics hardware structure.

Technically, the `omprun` command sets `LD_PRELOAD`,the link editor environment variable, to use the `libOMPmaster.so` library of matching format prior to using the `libGL.so` library. When the `omprun` command is not used on systems where it is is optional, OpenGL Multipipe utilizes the X server's DRI extension, a mechanism in the `libGL.so` library, and a special `SGIOMPdmx_dri.so` module to intercept and route an application's OpenGL calls to the OpenGL Multipipe library `libOMPmaster.so`.

For more information on using `omprun`, see the `omprun`(1) man page or use the `-help` command-line option of `omprun` as follows:

```
$ omprun -help
```

The following sections describe run-time options and how to best run different types of graphics applications:

- "Setting Run-Time Options"
- "Running OpenGL Single-Pipe Applications"
- "Running Pure X Applications"
- "Running Multipipe Applications in Multipipe-Aware Mode"

For more information on running applications with OpenGL Multipipe, see the release notes, `/usr/share/omp/release_notes/user/relnotes.html`.

Setting Run-Time Options

Even though the use of the omprun script is optional on Silicon Graphics Prism platforms, you may want to use the command to invoke the application to set run-time options, some of which can override configuration options set otherwise. The configuration options are described in section "Setting up the OpenGL Multipipe Environment" on page 14.

Table 3-2 shows the run-time options for `omprun` along with the related configuration resources. For more information about the various options, see the `omprun`(1) man page or use the `-help` command-line option of `omprun`.

Table 3-2 `omprun` Command-Line Options

omprun Option	Configuration Resource	Description
`-cull`	`culling: on`	Enable geometry culling.

`-cullshow`
`bbox`
`stats`
`all`
	`culling.showStat`
	`bbox`
	`stats`
	`all`
		Show culling information.

`-dlsplit`	`dlSplit: on`	Enable display list spatialization.
`-dlsplitmaxtris`	`dlsplit.maxTris`	Set the display list spatialization threshold.

Table 3-2 `omprun` Command-Line Options **(continued)**

omprun Option	Configuration Resource	Description
`-dlsplitshow`	`dlsplit.showRandomColors: on`	Show random colors.
`-dmxglx`	N/A	Use GLX instead of master and slaves.
`-latency`	`maxFramesLatency`	Set latency (how far ahead of the slaves the master works).
`-minpixels`	`culling.minPixels`	Set the small object culling threshold.

`-mstrmode`
`render`
`track`
`off`
	`masterMode`
	`render`
	`track`
	`off`
		Set the master rendering mode.

`-nodlopen`	N/A	Disable `dlopen/dlsym` overriding.
`-nosync`	`swapSyncMode: none`	Disable all slave synchronization.
`-novpclip`	`viewportClippingMode: off`	Disable viewport clipping.
`-pthread`	N/A	Force the master to use the pthread API.
`-swapready`	`swapSyncMode: barrier`	Use hardware swap barrier.

Running OpenGL Single-Pipe Applications

OpenGL single-pipe applications are the targeted applications for OpenGL Multipipe. To run such applications, simply enable the OpenGL Multipipe environment and invoke the application using the `omprun` script.

Under DMX, OpenGL applications started without the `omprun` script will display correctly on all screens, using the GLX indirect rendering support in DMX. However, using the `omprun` script will provide better performance for OpenGL applications.

Hint: For an easy way to run a number of single-pipe OpenGL applications under OpenGL Multipipe without the need to always explicitly invoke `omprun`, start a command shell under `omprun`, as shown in the following :

```
$ omprun xterm
<omprun xterm>$ app_name app_args
```

Any application started from this new command shell will be started automatically in transparent multipipe mode.

Running Pure X Applications

As noted in Chapter 1, “OpenGL Multipipe Overview”, the X proxy layer enables pure X applications to run transparently over multiple pipes. To run pure X applications, simply enable DMX before invoking them and they will run correctly. You do not need to use the `omprun` script for these applications.

Running Multipipe Applications in Multipipe-Aware Mode

Multipipe applications are intentionally written to take advantage of systems that have multiple graphics pipes. If they know about the underlying graphics hardware, they can explicitly address and take advantage of the individual graphics pipes. Typically, multipipe applications are written using OpenGL Performer or OpenGL Multipipe SDK.

It is best not to run such applications under OpenGL Multipipe, which hides the hardware configuration of the system from the applications. To run at full potential, these applications should be aware of the different graphics pipes in the system. To allow such applications to see the real graphics hardware does not require you to disable OpenGL Multipipe.

The OpenGL Multipipe layer may be bypassed on a per-application basis. This allows a multipipe application to be fully aware of the multipipe environment while other applications, aware of only a single large pipe, continue to run under OpenGL Multipipe. Applications that bypass the OpenGL Multipipe layer are said to run in multipipe-aware mode (or simply, *aware mode*), because they are aware of the different graphics pipes handled by the X server.

To run a multipipe application in aware mode while other single-pipe applications run concurrently in unaware mode, set the `DISPLAY` environment variable to point to the desired backend display that is managed by the X proxy layer (for example, `:0.1`), and then start the multipipe application with the `-aware` command-line option of the `omprun` script, as in the following example:

```
$ setenv DISPLAY :0.0
$ omprun -aware perfly
```

Under DMX, it is especially important to set the `DISPLAY` environment variable because the DMX meta display has a completely different display name than its component backend displays. By default, the DMX display is `:1` and the backend aware display is `:0`.

Hint: For an easy way to run a number of commands in aware mode, start a command shell in aware mode.

```
$ setenv DISPLAY :0.0
$ omprun -aware xterm
<aware xterm>$ app_name app_args
```

Any application started from this new command shell will be started automatically in aware mode.

Using SGI Scalable Graphics Hardware with OpenGL Multipipe

You may configure SGI Scalable Graphics Compositor hardware for use with OpenGL Multipipe to improve geometry and fill performance for an application. This requires no changes to the application. Using the following topics, this section describes how to configure DMX for this purpose as well as settings for OpenGL Multipipe to improve performance and usability in composited logical screen mode:

Configuring Composited Screens with DMX

To configure completely overlapped screens under DMX, simply create a DMX configuration file to manage the screens of the backend X server in the desired order. Do

not specify an offset or a 0x0 offset after the screen specifications. The following is an example configuration file:

```
virtual totaloverlap 1280x1024 {
    display :0.0 1280x1024;
    display :0.1 1280x1024;
    display :0.2 1280x1024 @0x0; # “@0x0” is optional
    display :0.3 1280x1024;
}
```

For more information on DMX configuration files, see section “Creating DMX Configuration Files” on page 16.

Specifying Static Composited Regions with `ompstartdmx`

You can specify a static tiling mode for a hardware compositor during DMX startup. The `ompstartdmx` script will start both the hardware compositor and DMX with the specified tiling mode.

The –`compositor` option on the `ompstartdmx` script allows you to specify the desired static tiling mode, as shown in the following example:

```
$ ompstartdmx -compositor mode
```

The following values are available for *mode*:

Mode	Description
`quad`	A quad tiling
`vert2`	Two vertical stripes
`vert3`	Three vertical stripes
`vert4`	Four vertical stripes
`horiz2`	Two horizontal stripes
`horiz3`	Three horizontal stripes
`horiz4`	Four horizontal stripes

You can also set up the video format using the –`compositor` option, as shown in the following entry:

```
$ ompstartdmx -compositor quad,1280x1024_75
```

The video format must be supported by both the hardware compositor and the pipes, and all pipes must have the same video format. Otherwise, `ompstartdmx` exits with an error. The default video format is the current one for the pipes.

Using Pixel Averaging Composition Mode for Full-Scene Antialiasing

You can set up the hardware compositor for pixel averaging during DMX startup. In pixel averaging mode, the following occurs:

1. OpenGL Multipipe applies a different sub-pixel jittering offset for each pipe's frustum.
2. The hardware compositor averages the pixel values.
3. The resulting image on the compositor output is antialiased.

The hardware compositor can be configured for pixel averaging mode with either two or four input pipes. For proper operation, configure all pipes connected to the compositor to be fully overlapped.

Note: When using two pipes, connect them to compositor channels 0 and 2, not 0 and 1.

If you are starting a session with only one compositor attached, then the simplest way to use pixel averaging is to use the `-compositor` option on the `ompstartdmx` script, as follows:

```
$ ompstartdmx -compositor {aa2 | aa4}
```

This command entry sets up a DMX configuration file with full overlap for two or four pipes and configures the compositor and OpenGL Multipipe for pixel averaging.

If you are using more than one compositor or you are use your own DMX configuration file, you must use a compositor configuration file, described in the "Using Multiple Compositors" section in the release notes.

You can configure the jittering offset values per application using the following X resources:

- `OMP*aa2Jitter.X` and `OMP*aa2Jitter.Y` (for two-pipe pixel averaging)
- `OMP*aa4jitter.X` and `OMP*aa4Jitter.Y` (for four-pipe pixel averaging)

For more information on the resources, see section “Resource Descriptions” on page 22.

Using Time-Based Compositing

You can set up an OpenGL Multipipe session in time-based composition mode. In that mode, either two or four pipes can be connected to each compositor. This allows two or four different rendering slaves to render two or four different frames in parallel. The hardware compositor will be configured to display one pipe at a time in a round-robin fashion. All frames will be seen on the compositor output while each rendering slave renders only every fourth or second frame depending on whether two or four pipes are attached to each compositor. In theory, this approach might scale the rendering performance by a factor of 4 (for the 4-pipe case). However, the following conditions should be met in order to get that performance improvement:

- Trivially, the application must be graphics-bounded (rather than host-bounded). Otherwise, scaling the rendering performance would not get any better overall performance. Usually, immediate mode applications tend to be host-bounded. Applications that draw static geometry in retained mode (for example, using display lists or vertex buffer objects) are more likely to scale.
- The application must use doubly buffered windows. Switching between one pipe to the next is done at the end of each frame when the application is calling **glXSwapBuffers()**. Singly buffered applications may run inside an OpenGL Multipipe session with time-based compositing but would not gain any performance improvement.
- The application should allow at least four (or two in a two-pipe configuration) frames of latency between the application and the visible compositor output. This minimum is required in order to render more than one frame in parallel. The maximum allowed frame latency can be controlled with the `OMP*maxFrameLatency` resource. Its default value is `4`, which is acceptable for time-based decomposition. However, setting this value to be lower will affect the rendering parallelism and, hence, performance.

Some operations causes OpenGL Multipipe to synchronize the application with the rendering slaves, which zero out any possible latency between the application and rendering slaves. Consequently, the application needs to be free of operations where synchronization will occur under the following conditions:

- When the application is calling **glFinish()**
- When the application is calling **glXWaitGL()**

- When the application is calling **glXQueryDrawable()**
- When the application is doing a **glGet*()** operation while the OpenGL Multipipe master mode is off (the default mode when not using the `omprun` command) and the specific queried OpenGL state is not being tracked by the OpenGL Multipipe internal state tracker
- When the application is calling **glReadPixels()**
- When switching between two different GLX contexts

If you are starting a session with only one compositor attached, then the simplest way to use time-based decomposition is to use the `-compositor` option on the `ompstartdmx` command as follows:

```
$ ompstartdmx -compositor {time2 | time4}
```

This command sets up a suitable DMX configuration file and starts an OpenGL Multipipe time-based composition session using the first 2 or 4 pipes connected to the compositor attached to hyperpipe network 0 in the system.

If you are using more than on compositor or need to use your own DMX configuration file, you must use a compositor configuration file, described in the "Using Multiple Compositors" section in the release notes.

After the OpenGL Multipipe session has been initialized, every doubly buffered OpenGL application running in the session will be joined to the time-based composition group. All such applications will share the same swap group. This means that all such applications will need to swap at the same time. When one application reaches the **glXSwapBuffers()** point, it will wait for all the other applications to reach their swap point before proceeding with the swap request. The result is that one application might block the another.

It is possible to run a doubly buffered OpenGL application under this session without being attached to the time-based composition group by running the application with `omprun -novpclip` *app_name* or by setting the `OMP*viewportClippingMode` resource value for that application to `none`. In this case, the application will not join the swap group and will not block other applications but will not gain any better performance either. The same applies for any singly buffered applications.

Applications attached to the time-based composition group will be forced to synchronize on vertical sync. Therefore, the maximum expected frame rate cannot be higher than the monitor refresh rate.

Using Multiple Compositors in an OpenGL Multipipe Session

OpenGL Multipipe allows you to configure multiple compositors by specifing a special compositor configuration file with the `-compositor` option for the `ompstartdmx` command, as shown in the following:

```
$ ompstartdmx -compositor config-file
```

The compositor configuration file *config-file* describes the desired mode of operation for each compositor in the session. The format and use of the configuration file is described in the "Using Multiple Compositors" section in the release notes.

Enabling Duplicate Cursor Images in Overlap Regions

When an X proxy layer is used to overlap screen regions on an edge-blended display or a compositor-based system, the cursor will seem to disappear when it enters the overlapped or uncomposited regions of the display. In X, a mouse belongs to one screen of the X server at a time. Therefore, it is normally not possible to draw the mouse multiple times (on different screens) in the overlap region.

To prevent the cursor from disappearing in these cases, you need to create additional or *duplicate cursor* images (not real cursors) where two or more screens overlap. There is still just one real cursor position on the display.

There are two ways to enable duplicate cursors under DMX:

- Managing an appropriate subarea of each composited input pipe instead of the whole screen area
- Managing multiple backend X servers

Managing Screen Subregions with DMX

You can use a DMX configuration file to cause DMX to manage less than a whole backend screen. When the subregions of each screen are visually assembled using a congruent static compositor tiling, the resulting logical screen will look and behave identically to the DMX displays with fully overlapping screens described previously. The following configuration file entry illustrates this feature:

```
virtual quadtilesubregions 1280x1024 {
    display :0.0 640x512+0+0     @0x0;
    display :0.1 640x512+640+0   @640x0;
```

```
    display :0.2 640x512+0+512   @0x512;
    display :0.3 640x512+640+512 @640x512;
}
```

Managing Multiple Backend X Servers with DMX

The X mouse cursor can be made visible in the partially or fully overlapped screen regions needed for edge-blended or composited displays. As long as the logically overlapping areas of the DMX display are composed by screens of different backend X servers, DMX can utilize the mouse cursor that is available on each backend X server.

Instead of having DMX manage multiple screens of a single backend X server, you can start multiple backend X servers and specify a DMX configuration that logically overlaps areas from different backend X servers, as shown in the following:

```
virtual totaloverlapmulti 1280x1024 {
   display :0.0 1280x1024;
   display :1.0 1280x1024;
   display :2.0 1280x1024;
   display :3.0 1280x1024;
}
```

Note: The multiserver technique can be combined with the technique of managing screen subregions to create a logical display on which mouse cursors are visible in all overlapped and composited regions.

Managing Windows for Aware Applications

To allow window manager support for applications started in aware mode, OpenGL Multipipe provides aware window management. A window manager wrapper is provided that is much like `omprun` for GL applications.

When the window manager wrapper is not used, applications started in aware mode (using `omprun –aware` *app_name*) will bypass the window manager. This means that their windows cannot be moved, resized, iconified, or otherwise managed. This includes the regular decoration provided by the window manager. The windows will not have this decoration. This occurs because an unaware window manager does not know about all of the real screens that the X proxy layer is managing. It cannot manage aware windows that were not created through X proxy layer.

If you invoke `ompstartdmx` with the standard arguments (or if DMX is configured to start automatically when you log in), a window manager of your choice will be automatically started with the window manager wrapper so that it is able to manage aware windows. You can change the DMX default window manger by using the `ompstartdmx –wm` option. You may follow the steps in the following subsections if you want to run a different window manager to manage aware windows.

This section describes the following topics:

- "Starting an Aware Window Manager"
- "Exiting an Aware Window Manager"
- "Setting an Aware Window Manager as the Default"

Note: The multipipe-aware window manager is currently not supported for compositor-based systems.

Starting an Aware Window Manager

To start an aware window manager, perform the following steps:

1. Exit or kill any unaware window manager that is currently managing the display.

   If possible, exit your window manager without logging out. One way to do this is to find the process number for your window manager and kill it manually, as the following illustrates:

   ```
   $ ps -e | grep my_window_manager
      23878 ? 0:42 my_window_manager
   $ kill 23878
   ```

   Some window managers may not allow you to exit the window manager and remain logged in. If this is the case, you will need to start the aware window manager from a `.xsession` file. See the section "Setting an Aware Window Manager as the Default" on page 43 for more information.

2. Start the specialized window manager.

   Enter the following:

   ```
   $ ompwrapwm twm
   ```

The `ompwrapwm` script starts the window manager `twm` in aware window management mode under DMX. If the display server is determined to be compatible, the script starts the window manager with aware window management support enabled. If the display server is not compatible, the script will exit. The script can be made to start the window manager in unaware mode (with aware window management disabled) as a contingency.

For more information on using the `ompwrapwm` script, see its man page or use the `-help` command-line option of the script as follows:

```
$ ompwrapwm -help
```

Notes:

- You can use any window manager with the `ompwrapwm` script. KDE is the recommended window manager.
- Starting an application in aware mode and then starting the window manager will result in the application's windows being unmanaged by the aware window manager. An aware window manager must be started prior to running an application in aware mode in order for its windows to be managed.

Exiting an Aware Window Manager

To exit an aware window manager, simply log out and log back in. The default window manager will again manage your display.

Setting an Aware Window Manager as the Default

An alternate way to run an aware window manager is to invoke the `ompwrapwm` *my_window_mgr* script in your `$HOME/.xsession` file. You must set up a default aware window manager under DMX. See section "Initializing DMX" on page 14.

Monitoring Performance

The main purpose of monitoring OpenGL Multipipe performance is to find bottlenecks. OpenGL Multipipe provides the following two tools for this purpose:

- "ompmon - The OpenGL Multipipe Monitoring Tool" on page 46
- "The OpenGL Multipipe PMDA for Performance Co-Pilot" on page 59

The `ompmon` tool is specific to OpenGL Multipipe and the applications running in its domain. The OpenGL Multipipe Performance Metric Domain Agent (PMDA) collects data that can be viewed in the Performance Co-Pilot environment along with other system-wide data (for example, CPU utilization, memory allocation, I/O operations, etc.).

This chapter describes these two tools. Once you identify performance bottlenecks, see Chapter 5, "Optimizing Performance".

Limitations:

With both monitoring tools, the two following limitations apply:

- Performance monitoring only applies to double-buffer applications since most performance metrics are averaged on a per-frame basis and a frame is defined to be the end of a **glXSwapBuffers()** call.
- Vertices drawn using vertex buffer objects (VBOs) or vertex array objects (VAOs) are not counted either in retained mode or immediate mode.

ompmon - The OpenGL Multipipe Monitoring Tool

The ompmon monitoring tool allows you to monitor OpenGL Multipipe performance and various other aspects of the applications running under OpenGL Multipipe. This section describes this tool using the following topics:

Starting ompmon

You invoke the graphical viewer using the ompmon command. By default, the viewer connects to and monitors the server specified by the DISPLAY environment variable. If you want the viewer displayed on a server different from the one where OpenGL Multipipe is running, you can use the -s option to specify the OpenGL Multipipe server to monitor. For example, the following entry specifies amstel:0 to be the ompmon display and keg:1 to be the OpenGL Multipipe display server:

```
keg % env DISPLAY=amstel:0 ompmon -s :1
```

The ompmon Screen

As shown in Figure 4-1, the ompmon screen has the following two main sections:

- **Active Applications** selection box (right side of the window)
- Application information (left side of the window)

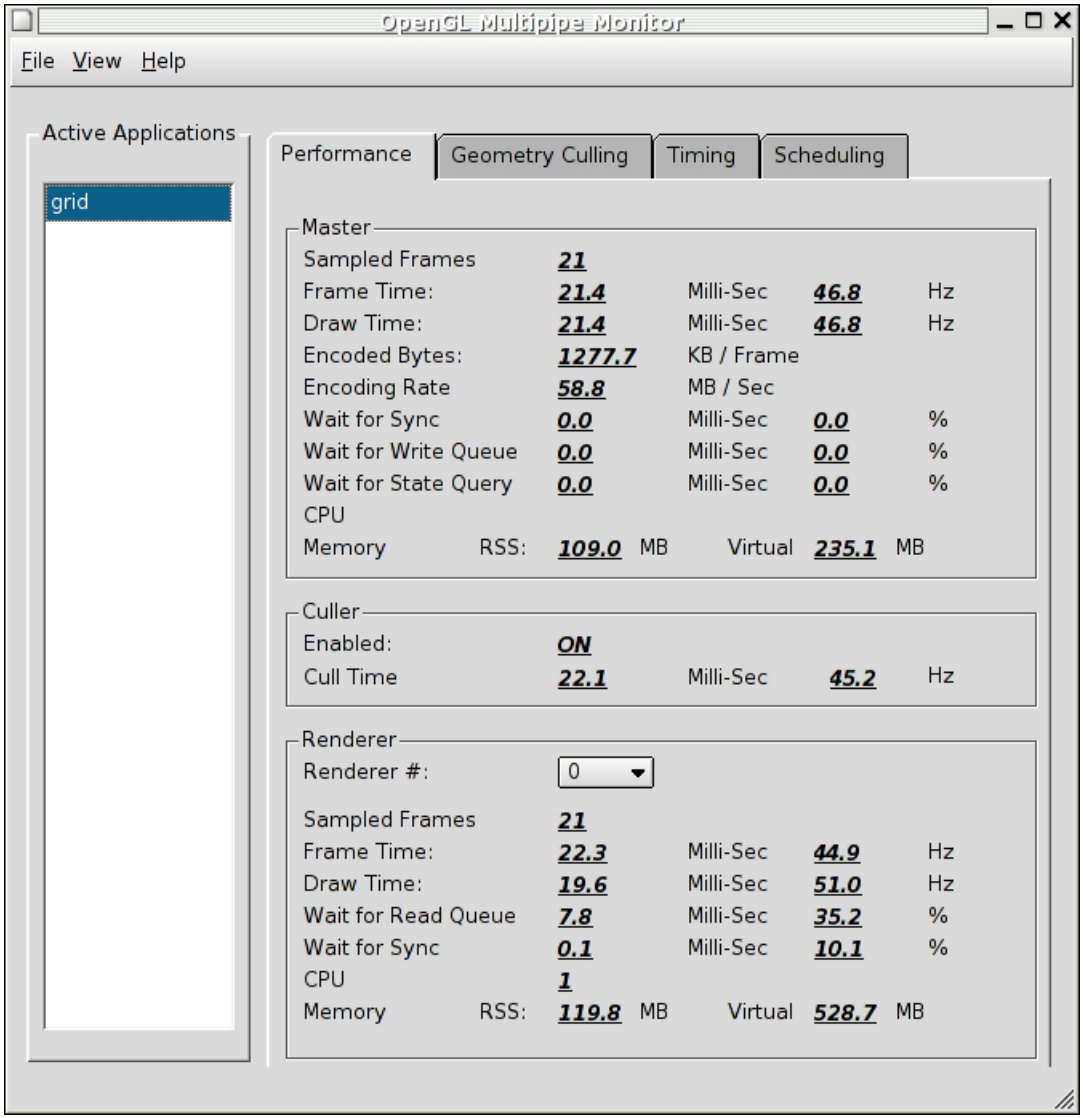

Figure 4-1 The ompmon Monitoring Tool

Active Applications

The **Active Applications** selection box lists the applications currently running under OpenGL Multipipe. Use this box to select the application you want to monitor.

Note: `ompmon` monitors each thread that makes OpenGL calls. Therefore, an application that has multiple rendering threads will have multiple entries in the list so that you can select the appropriate thread.

Application Information

The application information portion of the screen is actually a tabbed form that contains the following tabs:

Performance	Application performance behavior
Geometry Culling	Culling statitics
Timing	Inter-process execution timing
Scheduling	The process-to-CPU mapping

Performance Information

As shown in Figure 4-1, the **Performance** tab provides an overview of the application's performance under OpenGL Multipipe. The performance data is divided into the following three groups:

- Master data
- Culler data
- Renderer data

The various metrics are averaged across the number of frames as displayed in the **Sampled Frames** field. In a case where the application has not generated a new frame during the last sampling period, the data displayed by `ompmon` will not be updated and will be displayed in red.

Master Data

The OpenGL Multipipe master is the layer in OpenGL Multipipe that runs as part of the application. Its purpose is to intercept the application's OpenGL calls, encode them, and send them via shared memory to the renderers (slaves) for display. Table 4-1 describes the data collected for the master.

Table 4-1 Metrics for the OpenGL Multipipe Master

Metric	Description
Sampled Frames	Number of frames sampled. The number of sampled frames depends on the application frame rate and the data collection time defined in `ompmon` **Preferences** (see the **File –> Preferences** dialog). In order to get meaningful statistics, the data collection period should be set about 10 times the frame rate of the application.
Frame Time	Frame time, expressed in milliseconds as well as in frames per second, is defined to be the period between successive calls to **glXSwapBuffers()**.
Draw Time	Draw time, expressed in milliseconds as well as in frames per second, is defined to be the period between a call to **glClear()** and a call to **glXSwapBuffers()**.
Encoded Bytes	Amount of bytes per frame encoded by the master. This metric directly depends on the amount of data the application is sending to the graphics pipe per frame.
Encoding Rate	Encoding rate of the master. The value of this metric is dominated by the **Encoded Bytes** per frame value and the encoding performance of OpenGL Multipipe. A low value for this metric and a high value for **Encoded Bytes** might indicate that the application is CPU-bounded. A CPU pinning for the application should be considered. See the man page `dplace`(1) for details.
Wait for Sync	Time spent by the master waiting for slaves to catch up. The frame latency threshold is set by the `OMP*maxFramesLatency` resource.
Wait for Write Queue	Amount of time (and percentage of total draw time) spent by the master waiting for the queue to be available for writing.
Wait for State Query	Amount of time (and percentage of the total draw time) spent by the master waiting for state queries from the graphics pipes.

Table 4-1 Metrics for the OpenGL Multipipe Master **(continued)**

Metric	Description
CPU	CPU where the master last executed.
Memory	Resident and virtual memory footprint of the application.

Culler Process Data

The culler process scans the encoded stream generated by the master and attaches culling information to it for future culling by the renderers (slaves). Table 4-2 describes the data collected for the culler.

Table 4-2 Metrics for the Culler Process

Metric	Description
Enabled	`ON` if cull is enabled for this application.
Cull Time	Time between successive **glXSwapBuffers()** calls spent by the culler. Note that since the master and the culler process are inherently synchronized, the value for this metric should be close to the application's frame time. High values for the culler draw time might indicate a culling bottleneck; in this case, you should consider turning culling off.

Renderer Data

The renderers (slaves) are processes that receive the data encoded by the master and generates OpenGL calls to the graphics pipes. The number of the renderers running (as part of the an application) is determined by the number of pipes in the system and the `OMP*activeScreens` resource.

Table 4-3 describes the data collected for the renderers. You can select the renderer of interest from the **Renderer #** drop-down menu.

Table 4-3 Metrics for the Renderers

Metric	Description
Sampled Frames	Number of frames sampled during the last sampling period. See Table 4-1 for more details.
Frame Time	Frame time and period executed by the renderer. Frame time is defined to be the period between successive calls to **glXSwapBuffers()**.
Draw Time	Draw time, expressed in milliseconds as well as in frames per second, is defined to be the period between a call to **glClear()** and a call to **glXSwapBuffers()**.
Wait for Read Queue	Time (and percentage of the total frame time) spent by the renderer waiting for encoded data to become available for rendering. A high value for this metric indicates that the system bottleneck is on the encoding side. In such a case, optimization of the application (master) should be considered. Such optimization could be achieved either by allocating more resources to the application or optimizing the application directly. See Chapter 5, “Optimizing Performance” for details.
Wait for Sync	Time (and percentage of the total frame time) spent by the renderer waiting for synchronization. The time includes the following: - Time waiting for the other renderers unless `OMP*swapSyncMode` is set to `none`. - In time-based decomposition sessions, time waiting its turn to show the rendered frame. In this mode, several frames may be rendered in parallel by different slaves and one slave may completely render its frame before its turn and must wait for all previous frames to be displayed before it displays its frame. - In time-based decomposition mode, time waiting for other applications as well to reach their swap points as all double-buffer applications on such sessions share the same swap group. A high value for this metric might indicate either that one of the other slaves is slower or a wait for other applications if using the time-based decomposition mode.

Table 4-3 Metrics for the Renderers **(continued)**

Metric	Description
CPU	CPU where the renderer process last exectued.
Memory	Resident and virtual memory footprint of the renderer.

Geometry Culling Information

The **Geometry Culling** tab provides information about the geometry generated by the application and its culling statistics (that is, the amount of geometry eventually sent to each pipe). The **Geometry Culling** tab is only enabled if culling is enabled for the selected application. As shown in Figure 4-2, the statistics about the geometry sent to the renderer you select fall into the following three categories:

Category	Description
Direct Mode	Vertices processed by the pipe in direct mode
Retained Mode	Vertices processed by the pipe as part of an OpenGL display list
Totals	Total number of vertices processed by a pipe

Table 4-4 describes the metrics displayed for each of these categories.

Table 4-4 Geometry Cullling Metrics

Metric	Description
Total Vertices	Total number of vertices processed.
Culled Vertices	Number of vertices that were culled out.
Rendered Vertices	Number of vertices that were eventually rendered.
Bounding Boxes	Number of bounding boxes that were used to spatially sort the vertices. Note that there is a trade-off between the number of vertices (controlled using the culling configuration resources of OpenGL Multipipe) and the total performance. On one hand, a high number of bounding boxes increases the culling overhead. On the other hand, a low number of bounding boxes makes the culling less accurate and unnecessary vertices will be sent down to the pipe for rendering.

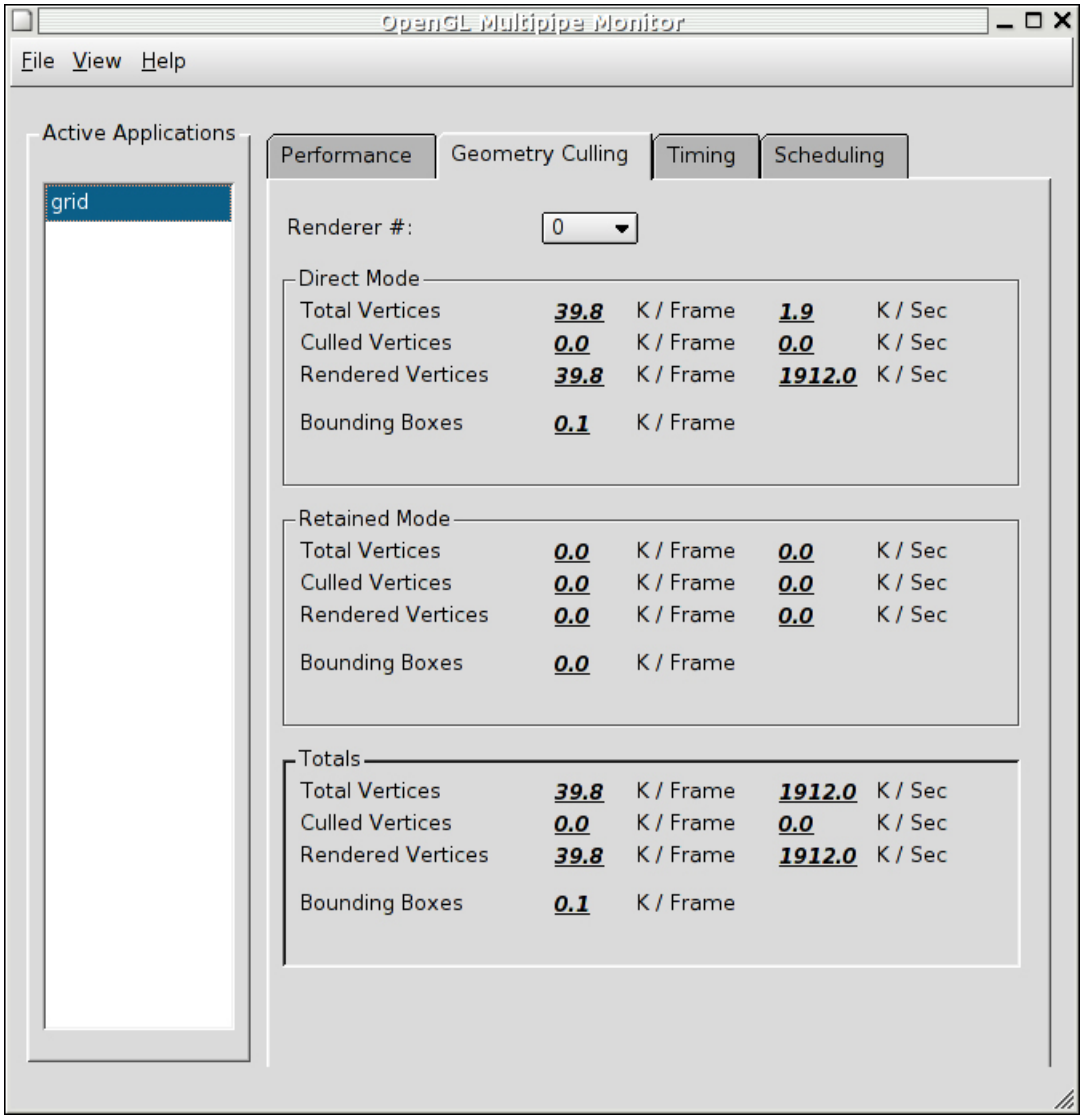

Figure 4-2 Geometry Culling Metrics

Scheduling Information

As shown in Figure 4-3, the **Scheduling** tab contains a chart with color-coded blocks to indicate which CPU is used by each OpenGL Multipipe process and the percentage of CPU time used by this process. The CPU usage is color-coded starting from yellow for low CPU usage through green and blue for medium usage up to magenta and red for high CPU usage.

Note: Processes that consume less the 5% of a CPU's time are not displayed.

For optimum performance, allocate each process with its own CPU. However, this requirement should be optimized according to the application's actual needs and the total amount of resources available in the system. The CPU used by each OpenGL Multipipe process can be controlled using the resource `OMP*slaveCpus` (see Chapter 3, "Using OpenGL Multipipe" for details).

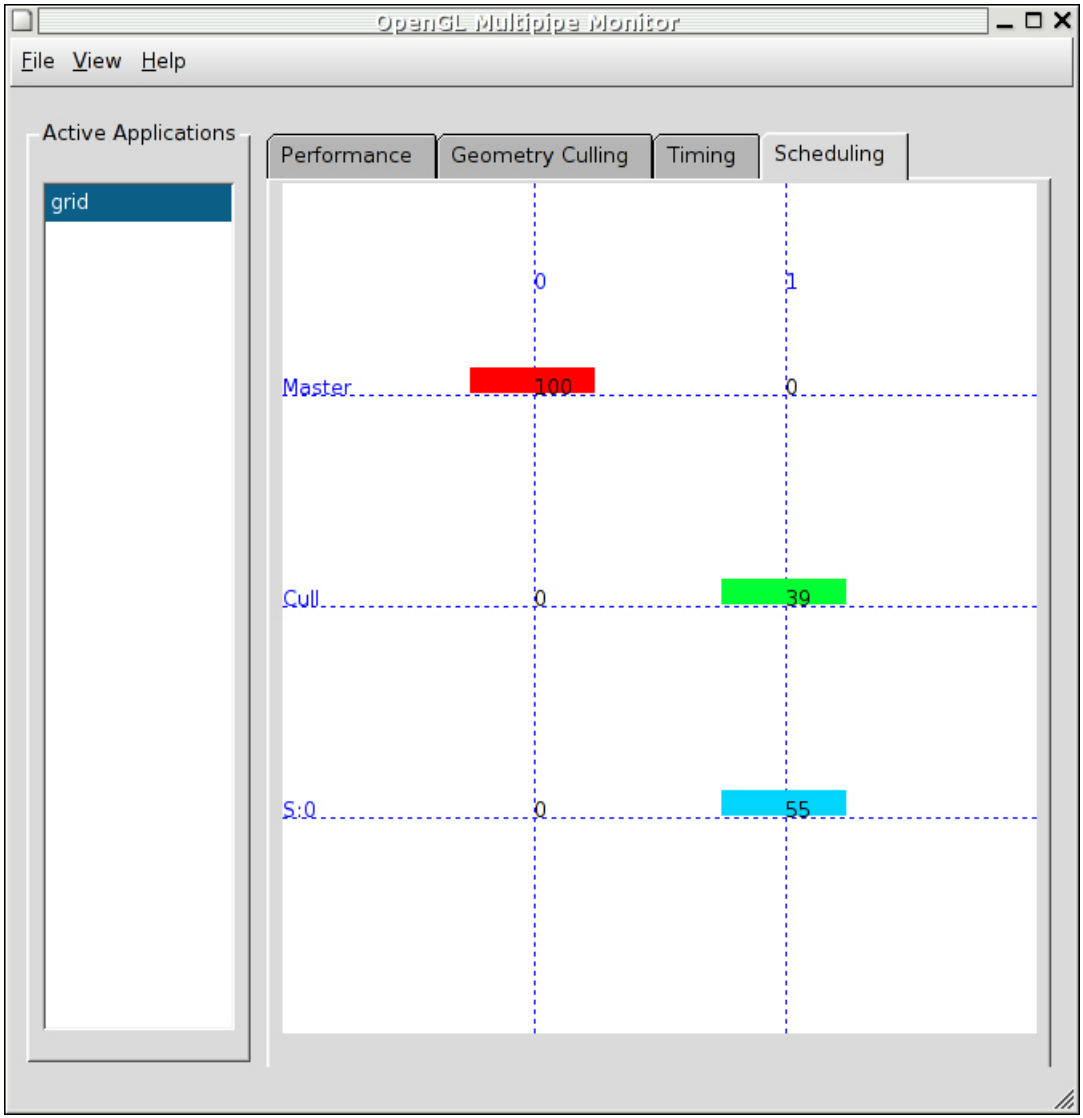

Figure 4-3 Scheduling Metrics

Timing Information

As shown in Figure 4-4, the **Timing** tab provides a timing chart that shows the parallelism of the OpenGL Multipipe pipeline. Each frame processed by OpenGL Multipipe is color-coded and marked with a label T-*n*, where *n* is the number of the frame prior to the last processed frame. In order to avoid display cluttering, the timing labels are not displayed if the processing time becomes small relative to the current display scale. Synchronization time is marked in gray and indicates that a process has completed its current frame and is waiting for synchronization before proceeding to the next frame. The time scale for the timing chart is controlled using the **Scale** drop-down menu. If the timing chart becomes "jumpy" and unreadable, you can pause the data collection of ompmon as described in the next section "Miscellaneous Controls" on page 58.

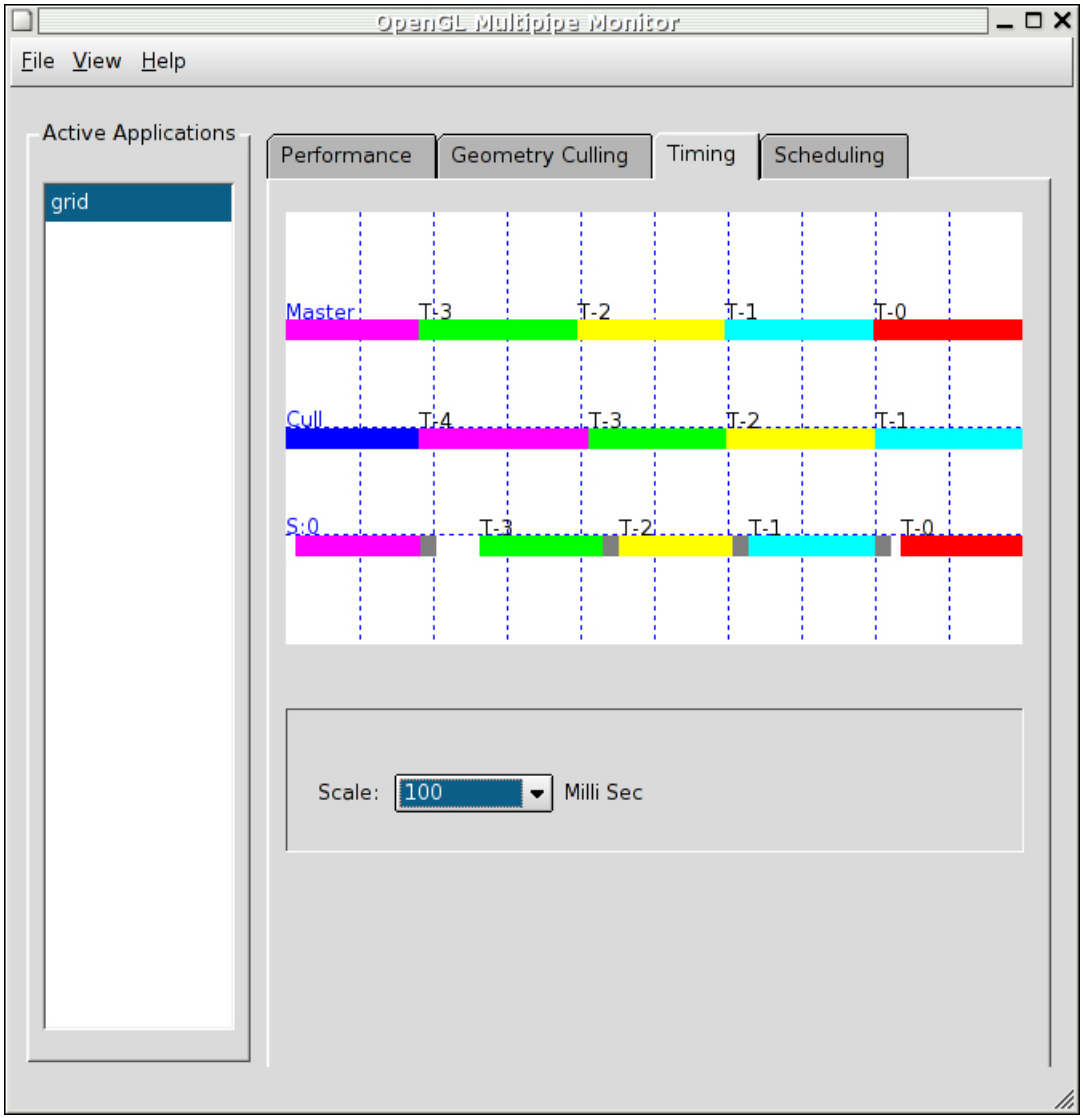

Figure 4-4 Timing Metrics

Miscellaneous Controls

In addition to selecting the application (or application thread) to monitor and the type of performance information to view using the tabs, you can also control other aspects of `ompmon` behavior—for example, pausing the display, controlling the frequency with which `ompmon` refreshes the application list, and controlling the reporting periods. Table 4-5 describes these miscellaneous controls.

Table 4-5 Miscellaneous `ompmon` Controls

Behavior	Description	Control
Data collection frequency	How often should `ompmon` sample the OpenGL Multipipe performance events queue. If the value for this period is too high, you lose performance data and, consequently, get inaccurate performance readings. If this value is too low, you get a heavy load on the `ompmon` process. A reasonable value should be in the range of 3 to 5 times the frame rate of the selected application.	**File -> Preferences** dialog from the menu bar
Data collection pause	`ompmon` collects no new data and displays a `(Paused)` string in its window frame.	**View -> Pause** from the menu bar. Toggle the **Pause** selection to resume data collection.
Reporting period	The period over which `ompmon` is to collect and report statistics. The value of this period should allow enough time to collect statistics and make the display stable and consistent. A reasonable value should be approximately 10 times the frame rate of the selected application.	**File -> Preferences** dialog from the menu bar
Check for new applications	How often to refresh the OpenGL Multipipe application list to account for new and exited applications.	**File -> Preferences** dialog from the menu bar

The OpenGL Multipipe PMDA for Performance Co-Pilot

The OpenGL Multipipe PMDA for Performance Co-Pilot collects the performance data from OpenGL Multipipe and makes it available for the various performance metrics viewers available with Performance Co-Pilot. For example, you can use the Performance Co-Pilot graphical viewer `pmchart` to display OpenGL Multipipe metrics. For the details about using the Performance Co-Pilot tool suite, see *Performance Co-Pilot for IA-64 Linux User's and Administrator's Guide*.

The OpenGL Multipipe PMDA installation is described in Chapter 2, "Installing OpenGL Multipipe". The description of the `OMP` command can be obtained using the following command:

```
% pminfo -t
```

Table 4-6 and Table 4-7 describe the metrics that the OpenGL Multipipe PMDA provides for Performance Co-Pilot.

Table 4-6 PMDA Metrics for the OpenGL Multipipe Master

Metric	Description
`multipipe.master.nframes`	Number of frames processed in the statistics period
`multipipe.master.frame_time`	Time elapsed between two successive **glXSwapBuffers()** calls
`multipipe.master.frame_rate`	Number of **glXSwapBuffers()** calls per second
`multipipe.master.draw_time`	Time elapsed between **glClear()** to **glXSwapBuffers()** calls
`multipipe.master.draw_rate`	1.0/`draw_time`
`multipipe.master.encoded_bytes`	Number of encoded bytes per frame
`multipipe.master.encoding_rate`	Encoding rate of the OpenGL Multipipe master
`multipipe.master.wait_write_queue`	Amount of time spent by the encoder waiting for the write queue to drain

Table 4-6 PMDA Metrics for the OpenGL Multipipe Master **(continued)**

Metric	Description
`multipipe.master.wait_write_queue_p`	Portion of the frame time spent by the encoder waiting for the write queue to drain
`multipipe.master.wait_state_query`	Amount of time spent by the encoder waiting for state queries on the graphics pipes
`multipipe.master.wait_state_query_p`	Portion of the frame time the encoder was waiting for state queries on the graphics pipes
`multipipe.master.wait_EOF_sync`	Time spent by the master waiting for end-of-frame synchronization
`multipipe.master.wait_EOF_p`	Portion of the frame time the encoder was waiting for end-of-frame synchronization

Table 4-7 PMDA Metrics for the OpenGL Multipipe Slaves

Metric (where N denotes the slave number)	Description
`multipipe.slave`N`.nframes`	Number of frames processed in the statistics period
`multipipe.slave`N`.frame_time`	Time elapsed between two successive **glXSwapBuffers()** calls
`multipipe.slaveN.frame_rate`	Number of **glXSwapBuffers()** calls per second
`multipipe.slave`N`.draw_time`	Time elapsed between **glClear**() to **glXSwapBuffers()** calls
`multipipe.slave`N`.draw_rate`	1.0/`draw_time`

Table 4-7 PMDA Metrics for the OpenGL Multipipe Slaves **(continued)**

Metric (where N denotes the slave number)	Description
`multipipe.slave`N`.wait_read_queue`	Amount of time the renderer was waiting for data to render
`multipipe.slave`N`.wait_read_queue_p`	Portion of the frame time the renderer was waiting for new data to render
`multipipe.slave`N`.wait_sync`	Amount of time the renderer to sync with other renderers
`multipipe.slave`N`.wait_sync_p`	Portion of frame that the renderer was waiting to sync with other renderers
`multipipe.slave`N`.direct_vertices`	Number of vertices generated by the application in direct mode
`multipipe.slave`N`.direct_vertices_rate`	Rate the application sends vertices in direct mode
`multipipe.slave`N`.direct_culled_vertices`	Number of vertices sent by the application in direct mode that were culled out
`multipipe.slave`N`.direct_culled_vertices_rate`	Culling rate of vertices sent by the application in direct mode
`multipipe.slave`N`.direct_rendered_vertices`	Vertices rendered in direct mode
`multipipe.slave`N`.direct_rendered_vertices_rate`	Vertex rendering rate in direct mode
`multipipe.slave`N`.direct_bbox_cmd`	Number of bounding boxes tests for vertices in direct mode
`multipipe.slave`N`.retained_vertices`	Number of vertices sent by the appliation in retained mode

Table 4-7 PMDA Metrics for the OpenGL Multipipe Slaves **(continued)**

Metric (where *N* denotes the slave number)	Description
`multipipe.slave`*N*`.retained_vertices_rate`	Rate of vertices sent by the application in retained mode
`modemultipipe.slave`*N*`.retained_culled_vertices`	Number of retained-mode vertices that were culled out
`multipipe.slave`*N*`.retained_culled_vertices_rate`	Culling rate of retained-mode vertices
`multipipe.slave`*N*`.retained_rendered_vertices`	Vertices rendered in retained mode
`multipipe.slave`*N*`.retained_rendered_vertices_rate`	Retained-Mode vertex rendering rate
`multipipe.slave`*N*`.retained_bbox_cmd`	Retained-Mode bounding box commands
`multipipe.slave`*N*`.total_vertices`	Total number of vertices sent by the application
`multipipe.slave`*N*`.total_vertices_rate`	Total rate that the application is generating new vertices
`multipipe.slave`*N*`.total_culled_vertices`	Total number of vertices that were culled out by OpenGL Multipipe
`multipipe.slave`*N*`.total_culled_vertices_rate`	Total culling rate of vertices that were sent by the application
`multipipe.slave`*N*`.total_rendered_vertices`	Total rendered vertices
`multipipe.slave`*N*`.total_rendered_vertices_rate`	Total vertices rendering rate
`multipipe.slave`*N*`.total_bbox_cmd`	Total number of bounding box commands generated by OpenGL Multipipe

Optimizing Performance

OpenGL Multipipe allows single-pipe OpenGL application to be run on multipipe environment in a transparent manner. However, OpenGL Multipipe defines some parameters that control the processing pipeline of the OpenGL stream. In order to achieve the best performance, those parameters will need to be changed depending on the application type and geometry data. The tools described in Chapter 4, "Monitoring Performance" can also help you identify areas for optimization.

This chapter describes these parameters and some guidelines for using them in the following sections:

The release notes provide a more technical discussion some of these features.

Viewport Clipping

Applications that draw large polygons with complex texturing or shading procedures are likely to be fill-limited—that is, the rasterization stage of the graphics pipeline is the bottleneck to improving performance. If slower performance results in proportion to an increase in the OpenGL window size, this is an indicator that fill performance could be the problem.

To eliminate the pixel fill bottleneck, polygon rasterization work can be divided among multiple graphics pipes. Using OpenGL Multipipe, this can be accomplished by simply positioning a window so that it spans multiple graphics pipes and each pipe performs an equal fraction of the rasterization work. On each component screen of the logical display, OpenGL Multipipe automatically clips the OpenGL viewport to the physical screen boundaries.

Viewport clipping is enabled by default. It can be disabled with the `omprun -novpclip` option or with the `viewportClippingMode` resource.

The fill performance benefits of viewport clipping can be more fully realized by using an SGI Scalable Graphics Compositor and specifying additional parameters to OpenGL Multipipe.

Geometry Culling

Applications that render large amounts of geometry in display lists can sometimes reach the limit of the polygon processing capabilities of the graphics hardware. Such an application is said to be transform-limited or geometry-limited—that is, the geometry transformation stage of the graphics pipeline is the bottleneck to improving performance. If performance remains the same when lighting or textures are disabled by the application, these are indicators that geometry performance is the limiting factor.

To eliminate the geometry transformation bottleneck, OpenGL Multipipe can divide geometry among multiple pipes. By default, OpenGL Multipipe renders all geometry on each graphics pipe, even if not all of the geometry is visible on a given pipe. When geometry culling is enabled, each OpenGL Multipipe slave process renders only the geometry from display lists, vertex arrays, and immediate mode commands that is visible on its pipe.

It is also possible for OpenGL Multipipe to cull geometry to user-specified OpenGL clip planes. See the `culling.cullUserClipPlanes` resource in section “Resource Descriptions” on page 22.

Geometry culling is enabled with the `omprun -cull` command-line option or with the `culling` resource.

Small Object Culling

OpenGL Multipipe can cull objects whose screen-space bounding box is less than a threshold number of pixels. You can specify the threshold with the –`minpixels` command-line option for the `omprun` script or with the `culling.minPixels` resource. Geometry culling must be turned on in order for this feature to take effect.

This feature can increase the rendering speed in applications that have many features that are too small to discern and, therefore, may not be of immediate interest—for example, screws in CAD/CAM applications. With a proper `minpixels` culling threshold, OpenGL Multipipe culls such small features and rendering performance is improved as a result.

Display List Partitioning

When rendering display lists with OpenGL Multipipe's geometry culling option enabled, OpenGL Multipipe culls or renders an entire display list as a unit. When the original display list has a large amount of geometry and spans large areas of the scene, performance scalability can suffer.

OpenGL Multipipe can optionally spatially divide user display lists to break them into smaller ones. After the division, the smaller, more spatially coherent pieces are more friendly to load balancing and display list culling.

This feature is enabled with the `omprun -dlsplit` command-line option or with the `dlSplit` resource. Other options related to display list partitioning are described in the section "Resource Descriptions" on page 22.

Master Rendering Modes

As cited earlier, the OpenGL proxy layer of OpenGL Multipipe has two components: a master render library that intercepts OpenGL calls made by the application and render slave processes that each receive a stream of OpenGL calls from the master and perform OpenGL rendering on the application's behalf.

The master render library functions as part of the application process (that is, the "master process") and can have additional responsibilities besides intercepting, packing, and

distributing OpenGL calls to the slave processes. The master process may also render directly to a single local pipe in place of a single slave process, or it may use a local pipe only to track OpenGL state while a slave process renders to that pipe.

The `omprun –mstrmode` option (or the `masterMode` resource) allows you to specify what functions the master component performs on the single local reference pipe. The master's mode may improve or hinder OpenGL performance depending upon the behavior of a particular application. Therefore, it is important to understand the implications of each mode.

The following master modes are available:

- `off`
- `track`
- `render`

Note: To use the `track` or `render` master mode, you must use the `omprun` command to invoke the application.

Master Mode `off`

The `off` master mode is most efficient for applications that do not perform **glGet***xxx***()** calls (GL state queries) in every frame, because in this mode querying GL state requires round-trip communication to a slave. The master process does not render; it only packs and distributes GL calls to all slave processes. A slave process renders to each pipe and one of the slaves is designated to track state for any occasional **glGet***xxx***()** or **glIs***xxx***()** queries. Figure 5-1 illustrates running in `off` mode.

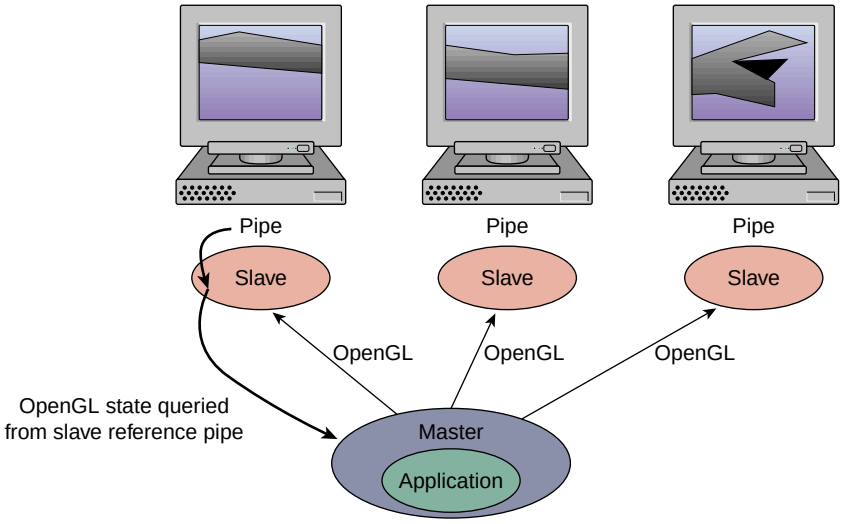

Figure 5-1 Running in Master Mode `off`

Master Mode `track`

The `track` master mode is most efficient for applications that frequently query GL state. The master process does not render but, in addition to packing and sending GL calls to all slave processes, it tracks the GL state on a local reference pipe. Slave processes render on all pipes. Figure 5-2 illustrates running in `track` mode.

Note: To use the `track` master mode, you must use the `omprun` command to invoke the application.

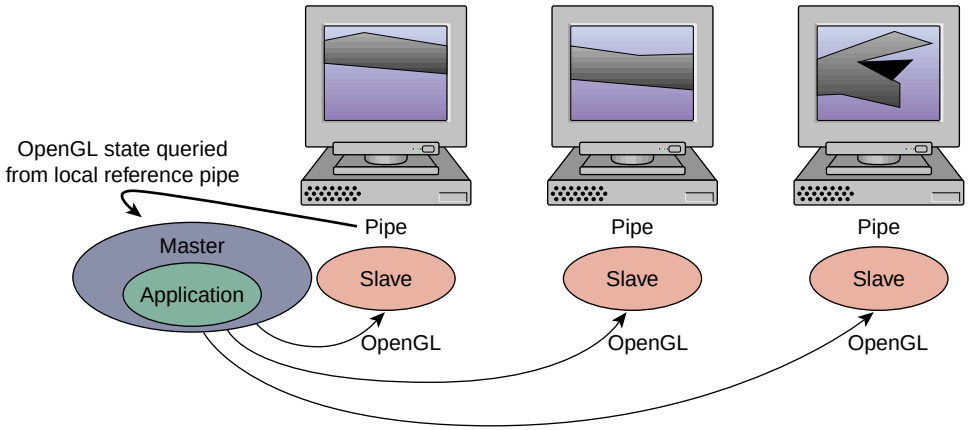

Figure 5-2 Running in Master Mode `track`

Master Mode `render`

The `render` master mode may yield slightly better performance for applications that do not use display lists and that run on systems with only two graphics pipes or with a limited number of processors. The master process renders and tracks GL state on a local reference pipe. One less slave process is needed because the application (master) process renders itself. State queries again are made to the master's local reference pipe. Figure 5-3 illustrates running in `render` mode.

Notes:

- To use the `render` master mode, you must use the `omprun` command to invoke the application.
- Some of the performance features described in this section, including geometry culling, are not available in `–mstrmode render` mode.

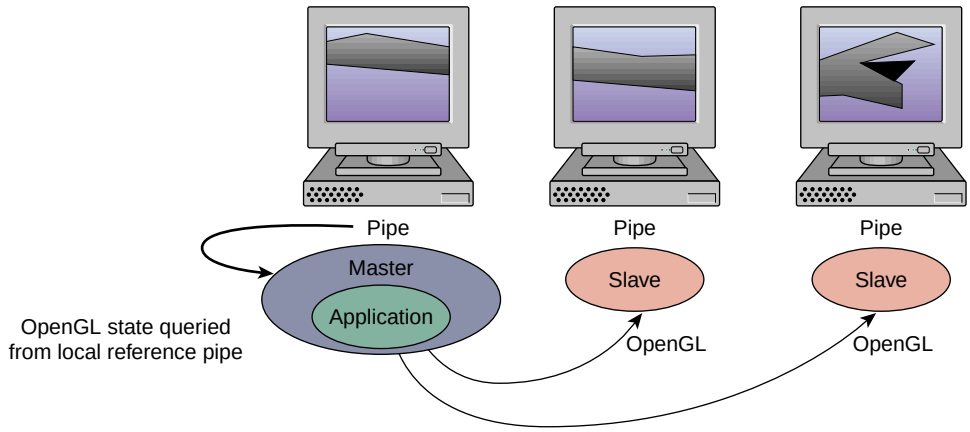

Figure 5-3 Running in Master `Mode` render

Frame Latency Control

OpenGL Multipipe uses a shared memory buffer in between the application and the drawing slaves. This buffer can introduce latency—that is, multiple frames can be buffered to be consumed gradually by the slave. If the application does not call **glFinish()** by itself, then OpenGL Multipipe allows the number of buffered frames to reach a small preset limit.

The latency helps smooth out application and drawing speed differences, and thereby increase throughput. However, if the amount of latency is beyond what you can accept, it can be limited by using the `omprun -latency` command-line argument (or the `maxFramesLatency` resource) to specify the maximum latency in number of frames.

The following command forces the master and slaves to have zero frames of latency between them:

`omprun -latency 0` *app_name app_args*

The master will wait for the slaves to finish executing enough of the previous frames so that the latency is below the threshold before beginning to pack data into the buffer for the next frame.

Buffer Swap Synchronization

Variations in pixel fill, geometry load, and many other factors can lead to an unbalanced load among the graphics pipes. Some pipes will render their parts of the scene faster or more slowly than the rest. Synchronization among the pipes is required to prevent one pipe from rendering faster or slower than another, which in some cases can present visible “tearing” in the output image.

By default, OpenGL Multipipe performs a software synchronization among the slave processes to ensure that they issue their respective swap buffer commands at the same time. The software synchronization approximates a synchronized swap in hardware.

Software swap synchronization is enabled by default. It can be disabled with the `omprun -nosync` command-line option or with the `swapSyncMode` resource. Note that this also disables any meaningful sense of frame latency.

Limitations

OpenGL Mulitpipe allows single-pipe applications to run in a multipipe environment without any modification and without the need to recompile the application. It also allows single-pipe and multipipe applications to run concurrently on the same X server. However, OpenGL Multipipe has limitations and the following sections describe them:

For release-dependent limitations, see the OpenGL Multipipe release notes.

Performance Enhancement

OpenGL Multipipe does not replace performance-focused multipipe APIs—such as OpenGL Performer or OpenGL Multipipe SDK—or any other custom multipipe solution. Using OpenGL Multipipe results in some minimal overhead (performance loss) for traditional single-pipe applications. This is due to the inherent cost of distributing the X and OpenGL commands among the graphics pipes.

It should be noted that process placement—that is, the way in which processes are assigned to processors—can significantly affect performance. For the affect of process placement on performance, see the OpenGL Multipipe release notes.

X Extensions

Some X extensions are not supported by DMX. Applications using these X extensions may not function properly. The behavior of these applications started in unaware mode is undefined, though they will generally behave correctly on screen 0 or in aware mode. To determine whether a particular extension is supported, see the section "X Application Uses Unsupported X Extension" on page 83.

The Multipipe-Aware Window Manager

Due to the nature of the screen overlapping required for composited displays, the aware window manager is currently limited to managing aware windows on noncomposited displays only. Unaware windows will be managed properly on composited displays.

OpenGL Window Size Constraints

The hardware graphics pipes have a hardware-dependent limit on the size of the region into which an OpenGL application renders. The consequence is that an OpenGL application is constrained to draw into a limited area. An OpenGL window may be placed anywhere within the the total area managed by the X server. Only the size of the region into which OpenGL renders is restricted.

Under DMX, however, OpenGL Multipipe allows OpenGL rendering to be unaffected by the window size limit of the graphics hardware. If the OpenGL application is invoked using `omprun`, the application may render into windows of any size.

SGI ProPack and OpenGL Multipipe Versions

Versions of OpenGL Multipipe that ship with SGI Propack for Linux are only supported on the version of SGI Propack with which they ship due to backwards compatibility issues on Linux.

Overlay Windows Support in DMX

DMX supports overlay windows—that is, windows that use *overlay visuals*—only if overlay visuals are available on each of the underlying X servers that DMX manages. This is the case on most SGI systems. On Silicon Graphics Prism platforms, you must explicitly enable support of overlay visuals in the configuration file(s) of the underlying XFree86 server(s).

Troubleshooting

This chapter describes some problems you might encounter and what to do to solve them. For additional considerations, see the OpenGL Multipipe release notes, `/usr/share/omp/release_notes/user/relnotes.html`.

This chapter documents the following problems:

Cannot Connect to the `ompslave` or `ompcull` Daemon

You will see a cannot-connect-to-daemon error if the `ompslave` or `ompcull` daemon is disabled or missing. OpenGL Multipipe installs and enables these daemons by default; therefore, this error could indicate an installation problem. If reinstalling OpenGL Multipipe does not solve the problem, the following instructions demonstrate how to manually correct the configuration problem.

The following commands should produce the output displayed:

```
$ chkconfig --list | grep sgi-omp
sgi-ompcull: on
sgi-ompslave: on
```

If either of the services is labeled `off`, turn them on by running the following command as `root`:

```
# chkconfig sgi-ompcull on
# chkconfig sgi-ompslave on
# killall -HUP xinetd
```

If the services do not exist, then the `sgi-omp` RPM needs to be reinstalled. Remove the installed RPM using the command `rpm -e sgi-omp` and re-install the package from the SGI ProPack CD.

Problems Starting DMX

If there is a problem starting the DMX proxy server, you may see output such as the following after running `ompstartdmx`:

```
ompstartdmx fatal: An error occured when starting Xdmx
Check the Xdmx log file for details: /tmp/Xdmx.log.xxxxx
```

This can result from a number of conditions, some of which have workarounds that are described in the following paragraphs. Inspect the `Xdmx.log.`*xxxxx* file, especially toward the end of the log, for messages that indicate one of the following conditions:

- Incompatible screens, no common visuals

  DMX will not create a logical display from graphics pipes with differing graphics capabilities. If the DMX proxy server detects that there are no common X visuals on the backend screens it tries to manage, DMX will abort with an error to this effect.

- Only one screen on display

  On systems having only one graphics pipe or in the case where the X server is directed to handle only one pipe, `ompstartdmx` will exit with an error such as the following:

  ```
  ompstartdmx fatal: Display :0.0 has only one screen.
  DMX was not started
  ```

  In these cases, it does not make sense to start DMX since there is only one pipe. However, specifying a configuration file with the `ompstartdmx –cfgfile` option will not prevent DMX from running on a single backend screen. Use the `ompstartdmx –help` option for more information.

- `ompstartdmx` fails to start or hangs with a GLX error.

  If a `GLX BadMatch` error occurs when starting DMX, the back-end screens may not have matching sets of GLX visuals. This is not a supported configuration and may be caused by a misconfiguration of the underlying X server(s), whose screens DMX manages.

  The configuration of each screen DMX manages should be identical. To verify that all screens have identical sets of visuals, inspect the output of the `glxinfo` command for each of the back-end screen(s) managed by DMX, as shown in the following:

  ```
  $ env DISPLAY=:0.0 glxinfo
  ```

  ```
  $ env DISPLAY=:0.1 glxinfo
  ```

  Piping the output through `wc` provides a quick comparison.

  Per-screen visual attributes and capabilities can be adjusted by modifying the file `/etc/X11/XF86Config`.

Problems Starting Applications with `omprun`

If an application will not start when using the `omprun` command, one likely scenario is that the DISPLAY environment variable does not point to a meta display. If the DISPLAY environment variable does not point to a meta display, ensure that the following conditions are true (check them in the order listed):

1. The DISPLAY environment variable points to the correct display.
2. An X proxy layer is enabled.

   See the section "Verifying That the OpenGL Multipipe Environment is Enabled" on page 30. An X proxy layer must be enabled or when you invoke an application with `omprun`, the application will exit with the following error:

   ```
   SGIomp fatal: ":0.0" is not a meta display
   ```

3. The application was not run from a shell that was started with the `omprun -aware` command or from a script that used the `omprun -aware` command to start the application.

   The `omprun -aware` command effectively disables the X proxy layer for any programs it invokes.

4. Your application does not use either the OpenGL Multipipe SDK or OpenGL Performer API.

   Recent versions of these APIs may be integrated with DMX and may not run under OpenGL Multipipe. The solution is to run these applications as is or to simply ensure that they are run in aware mode (with `omprun -aware`). Another alternative is to use older versions of these APIs that do not contain the X proxy aware code.

Setting OpenGL Multipipe Resources Has No Effect

If you have made changes to an X resource file but the changes do not appear to be having an effect, set the environment variable `SGIOMP_PRINT_CONFIG` to `1` so that OpenGL Multipipe prints the configuration values it uses to `stdout` when an application runs through OpenGL Multipipe.

Changing resources in an X resource file does not have an effect until the resources are merged into the active database in the X server with the `xrdb` command. For more information, see the `xrdb`(1) man page or use the command `xrdb --help`.

Shared Memory Failure

You might see a shared memory failure if the slave processes cannot open a connection to a back-end display server (for example, `:0.0`). Look in the file `/tmp/ompslave.log` to verify that this is the problem. A message indicating the connection was refused may indicate you need to set the `XAUTHORITY` environment variable or, otherwise, run `xhost+` on the X server.

If you run the application from within the DMX session, then `XAUTHORITY` will be set correctly for you. If you are running a command from a remote shell, you might have to set the `XAUTHORITY` environment variable to point to the correct X authority file before running the application. Otherwise, you will not have permission to open a connection to `:1` or `:0`.

In some X sessions, the `XAUTHORITY` variable will point to a temporary X authority file, but in sessions where `XAUTHORITY` is not set, `xauth` defaults to `$HOME/.Xauthority`. If `XAUTHORITY` is not defined in your session, set it to the following before running your application with `omprun`:

```
$ setenv XAUTHORITY $HOME/.Xauthority
```

For more information, see the `xauth`(1) man page.

Graphics Do Not Display Correctly on All Screens

If a graphics window displays correctly on some screens only, there are several likely scenarios, which are described in the following subsections:

- "Coding Problem in the Application"
- "You Are Using the Aware Window Manager"
- "Set-User-ID ("s-bit") Applications"

Coding Problem in the Application

If you are using the `omprun –cull` feature and you resize or move the application window to different screens, some applications may not draw an image properly on all screens. This can occur if an application does not call **glClear()** at the beginning of each frame (that is, it "builds up" an image, relying on a sort of rendering history from past frames). When culling is enabled, applications that do not call **glClear()** at the beginning of each new frame may have unusual rendering artifacts when they are moved from their initial window position. The culling feature by nature eliminates drawing commands that would otherwise be rendered into an off-screen region. To avoid this behavior, do not use the `–cull` option.

You Are Using the Aware Window Manager

If you started an application in aware mode (that is, by running the script `omprun –aware` *app_name*...), the application running in aware mode only draws to one screen. If you are running an aware window manager, it is possible that the window manager may position the window on a screen other than the one on which the application is drawing. To see the window rendered correctly, move the application's window to the correct screen.

Set-User-ID ("s-bit") Applications

If the `omprun` command is required to utilize the full OpenGL Multipipe environment on your platform, set-user-ID applications may not be able to utilize the full master-slave environment of OpenGL Multipipe. This is because the `omprun` script uses the `LD_PRELOAD` environment variable to force an application to load the OpenGL Multipipe library `libOMPmaster.so` instead of the standard OpenGL library, `libGL.so`.

For security reasons, Linux may ignore this environment variable for set-user-ID programs. Therefore, OpenGL Multipipe is not able to intercept and distribute OpenGL calls to all pipes. As a workaround, you can run the application using the GLX indirect rendering support in DMX. This, of course, has a performance price. To do so, set the `GLFORCEDIRECT` environment variable to `n` before running the application.

Cursor Movement Anomaly When Using a DMX Configuration File

This condition may occur when the DMX screen configuration differs from the screen connectivity of the back-end X server(s) and the mouse cursor is moved quickly from one screen to another.

When a back-end X server is configured so that some of its screens are neighbors, the DMX configuration for these screens should match the back-end neighbor topology for proper mouse cursor movement. Otherwise, the mouse cursor could jump or move to unexpected locations when it crosses a screen boundary where the backend screen layout does not match the DMX screen layout.

To avoid this condition for XFree86 backend X servers, make sure the screen arrangement in the `/etc/X11/XF86Config` file matches the screen arrangement in the DMX configuration file. By default, DMX and XFree86 manage screens left-to-right in a horizontal row. For more information about configuring XFree86, see the `XF86Config`(5x) man page.

Multipipe-Aware Applications Fail to Receive Events on Screen `0`

Windows of applications that are run in aware mode are not handled by ordinary window managers. This can cause some problems on screen `0` for keyboard events.

Moving away all the windows that are overlapping the aware window (even if these windows are displayed behind the aware window) will set the correct focus. The aware window will then receive the keyboard events.

Alternately, running the aware window manager will also fix the focus problem.

Nothing Displays or the Graphic Stalls or Hangs

If you start an OpenGL application with `omprun` and it does not display anything or the graphic stalls or even hangs, the source of the problem might be one of the following:

- "Coding Problem in the Application"
- "Improperly Wired Genlock or Swap Ready Cables"

Coding Problem in the Application

You may see a blank display or experience stalls or hangs for OpenGL applications that do not call **glFlush()**, **glFinish()**, or **glXSwapBuffers()** at the end of each frame. This causes OpenGL Multipipe to draw only when its internal buffer overflows. It can happen that the buffer never fills, in the case of an event-driven application—that is, the application draws one frame and waits for an event before drawing the next frame. Unfortunately, there is no workaround for applications that are not frame-based because OpenGL Multipipe relies on regular calls to the functions just cited.

Improperly Wired Genlock or Swap Ready Cables

If you are experiencing long delays between frames of an OpenGL application (whether or not it was started with `omprun`), the condition may have resulted from using the `omprun -swapready` option with improperly configured or improperly wired Genlock or Swap Ready cables.

For more information about this problem and a workaround, see the OpenGL Multipipe release notes.

X Applications Are Not Behaving Correctly or Fail to Start

If X applications are not behaving correctly or fail to start, consider the cases described in the following subsections:

- "X Application Uses Unsupported X Extension"
- "Application Window Disappears"
- "Application Explicitly Opens a Display Connection to :0.0"

X Application Uses Unsupported X Extension

Verify that the application is not using unsupported X extensions. Unfortunately, there is no way to accurately list the extensions an application uses. However, the following examples using the `nm` command give some hints about the extensions used. If an application is using an X extension, this can usually be detected by searching for occurrences of the string `extension` or for the name of a particular extension. The `xdpyinfo` command lists the names of extensions supported by the X server.

Indicating the use of the `DOUBLE-BUFFER` extension (DBE), the following example shows that command `gmemusage` calls `XdbeQueryExtension`:

```
# nm /usr/sbin/gmemusage | grep -i extension
[116] |2143299120| 436|FUNC |GLOB |DEFAULT  |UNDEF| XdbeQueryExtension
```

For a list of X extensions supported by DMX, see the `Xdmx`(1) man page in the directory `/usr/share/omp/doc/user`. Applications that use unsupported X extensions may be run in aware mode by running them with the `omprun –aware` option so that they bypass the X proxy layer.

X extensions supported on Silicon Graphics Prism systems but not by DMX include the following:

```
DOUBLE-BUFFER            ReadDisplay
FontCache                SGI-SCREEN-CAPTURE
MIT-SCREEN_SAVER         SGI-VIDEO-CONTROL
MIT-SHM                  TOG-CUP
MIT-SUNDRY-NONSTANDARD   X-Resource
RANDR                    XFree86-Misc
READDISPLAY              XFree86-VidModeExtension
RENDER                   XVideo
```

Application Window Disappears

This can occur if you start an application on one of DMX's backend displays without using `omprun –aware`—for example, if your `DISPLAY` environment variable is set to `:0.0` and the DMX proxy is managing display `:1`. The application will run on `:0.0`, a backend ("aware") display, but because it was not started properly in aware mode (using `omprun –aware`), the window will not be managed by the window manager running under DMX. Consequently, it may be "pushed" behind the DMX root window. Exiting the DMX session will reveal the application window. To avoid this problem, open the application in aware mode using `omprun –aware` *app_name*.

Application Explicitly Opens a Display Connection to `:0.0`

This problem can manifest itself in many ways:

- An apparent X error may appear in the output of the application.
- A window may suddenly disappear behind the DMX root window.
- The command `omprun` may output the following message:

  ```
  DISPLAY does not point to a meta display.
  ```

This problem originates from a program explicitly requesting an X display connection to `:0.0` even if the `DISPLAY` environment variable contains a different default display string, which is usually the case when running DMX.

The workaround is for DMX to run as display `:0` and to have the backend X servers use other display numbers so that when the application tries to open `:0.0`, it correctly connects to the DMX meta display on `:0` instead of a backend display by accident. To have other X servers use other display numbers so that `:0` is available for DMX, do the following:

1. Open file `/var/X11/xdm/Xservers`.
2. Replace the instance of `:0` with another display number (for example, `:1` or `:5`).
3. Restart graphics.
4. Remove the `/tmp/.X11-unix/X0` Unix socket file if it exists.

   The existence of this file will cause `ompstartdmx` to start DMX on a display other than :0.

Now when you run `ompstartdmx`, by default, it will use the first available X display number, which should now be `:0`.

Flickering Gray Rubberband During Window Movement

This occurs as a result of the lack of overlay visual support in DMX. See “Overlay Windows Support in DMX” on page 73.

Mouse Disappears on Composited or Edge-Blended Display

When an X proxy layer is used to overlap screen regions on an edge-blended display or a compositor-based system, the cursor will seem to disappear when it enters the overlapped or uncomposited regions of the display. In X, a mouse belongs to one screen of the X server at a time. Therefore, it is normally not possible to draw the mouse multiple times (on different screens) in the overlap region.

To prevent the cursor from disappearing in these cases, you need to create additional or *duplicate cursor* images (not real cursors) where two or more screens overlap. There is still just one real cursor position on the display. See the section "Enabling Duplicate Cursor Images in Overlap Regions" on page 40.

Problems Running Multithreaded Applications

If the application supports the use of POSIX threads (*pthreads*), use the pthread threading model with OpenGL Multipipe.

To force the use of the pthread threading model, use the `-pthread` option when starting the application, as shown in the following:

```
$ ompprun -pthread app_name
```

`ompstartdmx` Does Not Start a Window Manager

With no arguments, `ompstartdmx` starts only an xterm on Linux. To start GNOME or KDE under DMX, use the `ompstartdmx –session` option. OpenGL Multipipe also installs some special GNOME and KDE login options that will appear in the `gdm` login screen. See `ompstartdmx –help` and section "Initializing DMX" on page 14.

Problems with Aware Window Management

The following subsections detail problems with aware window management and workarounds:

- "Windows of Some Aware Applications are Not Managed"
- "Ghost Windows Appear In Overlap Regions on Edge-Blended Displays"

Windows of Some Aware Applications are Not Managed

For windows of aware applications to be managed, first start the aware window manager, then start the desired application in aware mode. If the reverse is done, the windows will not be able to be manipulated.

Ghost Windows Appear In Overlap Regions on Edge-Blended Displays

Aware windows bypass the X proxy layer and are only created on one physical screen, but when an aware window manager manages an aware window, it creates window frames on each screen. The frames are multipipe-transparent—that is, drawn on every screen. However, the application window within the frame is multipipe-aware—that is, drawn only on one screen.

Since an aware application window is not drawn on every screen, the multipipe-transparent frame behind the application window will "show through" in screen-overlap regions on an edge-blended display.

To work around this problem, you may want to run your application in a window of a size and position such that it does not overlap any of the screen-overlap regions of the display. Alternatively, you may want to temporarily quit the aware window manager.

Applications Do Not Behave Correctly in Aware Mode

The `ompstartdmx` script sets the environment variable `SGIOMP_META_DISPLAY`, which is important for running applications in aware mode under DMX. If you are starting an application in aware mode from a remote shell or other login shell, this variable will not be set. If unexpected results occur when you try to run an application in aware mode, ensure that the `SGIOMP_META_DISPLAY` variable is set to the DMX display (often `:1`).

Index

P

R

S

T

U

V

W

X