Silicon Graphics® UltimateVision™ Graphics Porting Guide

007-4297-001

CONTRIBUTORS

Written by Robert Grzeszczuk, Ken Jones

Production by Karen Jacobson

Engineering contributions by Praveen Bhaniramka, Terrence Crane, Alan Commike, Jackie Cox, Brad Grantham, Simon Hayhurst, Eric Kunze, Yair Kurzion, Shrijeet Mukherjee, and Guy Russell

COPYRIGHT

© 2004, Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND

The software described in this document is “commercial computer software” provided with restricted rights (except as to included open/free source) as specified in the FAR 52.227-19 and/or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS

Silicon Graphics, SGI, the SGI logo, Altix, InfiniteReality, IRIX, Onyx, OpenGL, and Origin are registered trademarks and NUMAflex, NUMAlink, OpenGL Multipipe, OpenGL Performer, Onyx4, and UltimateVision are trademarks of Silicon Graphics, Inc., in the United States and/or other countries worldwide.

ATI is a registered trademark of and FireGL, FGL, Radeon, and SmoothVision are trademarks of ATI Technologies, Inc. IBM is a registered trademark of International Business Machines Corporation in the U.S. Intel386 is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in several countries. UNIX is a registered trademark of the Open Group in the United States and other countries. XFree86 is a trademark of the The XFree86 Project, Inc. All other trademarks mentioned herein are the property of their respective owners.

Record of Revision

Version	Description
001	June 2004 Original publication.

Contents

About This Guide

This porting guide describes UltimateVision graphics for Silicon Graphics Oynx4 UltimateVision visualization systems and Silicon Graphics Visualization Systems for Linux. UltimateVision graphics combines the high bandwidth and scalability of the Onyx architecture with superb graphics performance, programmability, and prevalence of off-the-shelf graphics processing units (GPUs).

Audience

This guide is intended for graphics programmers who use SGI systems. It describes the architectural features of UltimateVision graphics and how it supports new and existing extensions to OpenGL.

Related Publications

The following documents contain additional information that may be helpful:

- *OpenGL Reference Manual*
- *OpenGL Programming Guide*

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library:

http://docs.sgi.com

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.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. 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 can be found on the back cover.)

You can contact us 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, California 94043-1351

We value your comments and will respond to them promptly.

Product Overview

Silicon Graphics UltimateVision graphics systems provide a major shift in capabilities and price/performance when compared to previous visualization systems but also maintain compatibility with previous Onyx systems. The best features of Onyx systems—the flexible, scalable, shared memory architecture—have been combined with the best-of-class industry-standard graphics components to create the most powerful visualization solution.

UltimateVision graphics is the next generation in the scalable graphics family, and the first product in the family that provides performance and features that rival InfiniteReality graphics but at a price to compete with clusters. UltimateVision graphics supports compositing of graphics pipes to build channels with massive performance or amazing image quality. Silicon Graphics Onyx4 UltimateVision systems can be configured with up to 32 graphics pipelines and 64 CPUs with up to 128 GB of memory. Each graphics pipe is also capable of driving two independent displays providing up to 64 channels of video.

System Configurations

Onyx4 systems can be ordered in three different configurations: Power, Team, and Extreme. They are all fundamentally built from the same components, but each of these configurations fit a different-sized solution:

- Power configuration
  - 2-8 CPUs
  - 2 or 4 graphics pipes
  - No routers

- Team configuration
  - 8-16 CPUs
  - 4, 6, or 8 graphics pipes
  - 2 routers
- Extreme configuration
  - 16-64 CPUs
  - 8-32 graphics pipes (always in pairs)
  - 4 routers

Scalable System Architecture

The single most distinguishing feature that differentiates the UltimateVision product line from competition is its high scalability. UltimateVision graphics systems offer a single system image (SSI) architecture, in which there is only one copy of the operating system (OS) running and all the memory and devices are available to all the processors. While the traditional Symmetric Multi-Processing (SMP) architectures do not scale well beyond 24 CPUs and 3 pipes, NUMAflex, an SGI implementation of Cache Coherent Non-Uniform Memory Access (ccNUMA), enables UltimateVision systems to expand efficiently well beyond the traditional bus-based architecture systems.

UltimateVision systems can be configured with as many as 64 processors, but the ccNUMA architecture, which is also used in the Origin and Altix product lines, has been shown to handle as many as 512 CPUs and 1 TB of shared memory. This is accomplished primarily by scaling system bandwidth as CPUs and memory are added provide balanced resources even for very large systems.

Some of the scalability that SGI provides can be emulated using traditional architectures by clustering multiple SSI machines together. Of course, using clusters is the hard way to achieve scalability. The difficulties start with the physical configuration of the nodes, disks, operating systems, network infrastructure, rack space, power distribution, and all the other components that SGI ships ready to plug in, boot up, and get to work. The day-to-day issues of managing a cluster—like nodes crashing and needing to be rebooted, processes hanging and needing to be reset, disks failing or filling up and needing to be maintained—make clusters a significant long-term challenge. In addition, technology updates on clusters are pretty much complete “fork-lift” upgrades due to the complications of using components with unsynchronized life cycles. An SSI machine

with a single file system, single operating system, and single view of all the processes running on the system makes working with even the largest configuration straightforward.

A cluster runs as many operating systems and instances of the application as there are nodes. When a problem can be easily sliced into smaller chunks that do not require much communication, this approach works well. Unfortunately, for most applications that want to take advantage of the power of visualization, typical graphics problems are extremely difficult to split up into small enough chunks with low enough communication requirements to make clusters feasible. On SGI systems, global shared memory and support for multiple OpenGL streams from one application removes the complications and performance issues of communicating between multiple instances of an application and an operating system.

Scalable Graphics Architecture

UltimateVision systems were architected to facilitate scaling of graphics performance by attaching multiple modules. Up to 16 graphics modules, with each module having 2 graphics pipes, can be attached to a single shared memory in order to scale various aspects of graphics performance. The pipes are fully synchronized to make the system particularly well suited for such visually challenging applications like immersive environments without distracting and disruptive image tearing. In addition, multiple pipes can be configured to participate in a single rendering task thanks to a unique graphics compositing architecture.

GPU scalability starts at the system level. SGI systems are based on small building blocks known as bricks. A brick can be CPUs with memory, only memory, CPUs with memory and graphics cards, only graphics cards, I/O units, routers, and other types of components. NUMAlink technology enables these bricks to be a single system, in contrast to cluster components connected with a standard interconnect. The sum of all the bricks and their connections creates a high-performance memory fabric. This fabric provides the SSI capabilities: a single OS, a single large pool of memory, and a single process space with all resources (GPUs, memory, disk, and so on) accessible from every other CPU. A system can be as small as 2 processors and 2 graphics cards or as large as hundreds of processors and tens of graphics cards. Scalability allows an application to drive a single GPU with a small dataset or 10 GPUs with a 10X larger data set. Memory bandwidth, disk I/O, bus bandwidth, efficient kernel locking, and many other factors go into creating this scalability.

Scalability reaches beyond rendering to system accessibility and flexibility. SGI provides hardware compositing solutions to combine multiple GPU output streams together into a single video signal that can be sent to a display device. Many system-level tools are also provided to help configure and manage systems with large numbers of GPUs and CPUs. High-performance OpenGL applications that explicitly take advantage of large CPU- and GPU-count systems will work very well in this new architecture. Conversely, there is also a software layer, OpenGL Multipipe, that allows single pipe applications to scale seamlessly across the GPUs available in a system without the need to write any explicit parallel CPU or GPU code. Additional toolkits that enable an application to choose the level of scalability assistance needed are also available. OpenGL Multipipe SDK is an API layer that provides performance-oriented scaling. OpenGL Performer is an alternative scaling API that is focused on enabling applications to manage the data through a scene graph paradigm.

Commodity Graphics Features

Silicon Graphics UltimateVision systems utilize the best-of-breed commodity GPUs with their continuously improving performance and evolving feature sets. Applications currently running on Onyx systems will, with few exceptions, be able to run on the new architecture. Taking advantage of all the subtle differences and power of scalable graphics can take more development effort. However, support for the latest version of OpenGL, fully hardware-accelerated transform, lighting, rasterization, and compositing is already present, as are many of the more esoteric features and capabilities:

- Ttransform rate of 300 Mtris/s
- Fill rate of 2.4 Gpix/s
- Vertex programmability
- Fragment programmability
- 16 textures per pass
- Two-sided lighting
- Directional and local lighting
- Up to 8 light sources
- 6 user-defined clipping planes
- Fast Z and color clears
- Floating point color in texture engine, shader engine, and the framebuffer

- 128-bit rendering precision (render pipeline with 24 bits per component and textures with 32 bits per component)
- Vertex array objects (VAOs) and element arrays
- Cubemap, multitexture, and so on
- 2x/4x/6x full-scene antialiasing modes
- Dual DVI-I support of any combination of digital and analog displays (Caveat: the second output can only be analog.)
- Maximum resolution of 2048x1536 per display
- Independent resolution and refresh rate selection for any 2 connected displays

Many of these features will be well familiar to existing Onyx users. However, the new hardware introduces enough new features and idiosyncrasies to warrant a detailed look. The following chapters describe some of these issues in greater detail.

Architectural Overview

Chapter 1, “Product Overview” provides a high-level view of the following architectural features of a Silicon Graphics UltimateVision system:

- Scalable system architecture
- Scalable graphics architecture
- Use of commodity graphics

This chapter describes more details about its hardware features along with some other hardware-related features. The following chapters describe its programming features.

Hardware Features

This section describes the hardware features of the Silicon Graphics Onyx4 UltimateVision visualization system for IRIX.

System Architecture

The Silicon Graphics Onyx4 UltimateVision system is available in a number of configurations to meet your visualization requirements:

- Power configuration
- Team configuration
- Extreme configuration

These configurations are described in section “System Configurations” on page 1. This section describes the two graphics bricks used in Onyx4 systems as well as the architecture of its graphics pipes.

G2-bricks and G2N-bricks

An Onyx4 system uses two kinds of bricks for graphics output: a graphics-only brick (G2-brick) and a graphics/node brick (G2N-brick). The G2-brick is a 2U rack-mountable enclosure containing two high-performance graphics pipes. This brick connects as an I/O device. The G2N-brick is a 2U rack-mountable enclosure which, in addition to the two high-performance graphics pipes, adds to the host system a node board with two or four CPUs and up to eight memory DIMMs. This brick connects as an integral part of the host system's compute fabric. Though it contains CPUs and memory, the G2N-brick does not have boot I/O functionality and, therefore, may not be used as a standalone system. Though internally different, the G2-brick and the G2N-brick may not be distinguished by external features.

Each G2-brick or G2N-brick contains two graphics pipes, each capable of supporting 2 display devices. An Onyx4 system may contain up to 16 such bricks, providing a maximum of 32 pipes and 64 display devices.

Graphics Architecture

Each graphics pipe in an Onyx4 brick contains an ATI FireGL X2-256 accelerator, which is a high-performance board based on the FGL 9800 Visual Processing Unit, featuring four geometry pipelines and eight pixel pipelines. The R350 chip that it contains is also used in the consumer versions of the ATI boards: Radeon 9800 Pro and 9900 Pro. R350 upgrades R300, which was the basis for Radeon 9500 Pro and 9700 Pro. This graphics pipeline supports full 24-bit floating point precision per channel throughout and 32-bit per-color component framebuffer.

256MB of on-board memory, combined with GDDR2 memory running at an effective clock rate of 700MHz or 22.4 GB/s of theoretical peak memory bandwidth makes it suitable for applications and configurations that demand high-framebuffer resolution, rich texture content, antialiasing, and/or anisotropic filtering. Of the total memory size, 128MB is reserved for textures, display lists, and vertex array objects (VAOs) with the remaining 128MB used by the framebuffer, command FIFO, and pbuffers. Any leftover framebuffer memory is available for additional texture, display lists, or VAOs.

The geometry engine on an Onyx4 system has four vertex engines, each capable of executing a single vector operation per clock. Thus, the theoretical peak transform rate is one vertex per clock. Similarly, the shader engine is capable of executing an operation (arithmetic/logical/texture interpolation) on eight pixel fragments in parallel. In both

cases, the observed performance will likely be limited by factors like memory bandwidth and cache misses.

Onyx4 graphics is equipped with a highly tuned memory controller to better arbitrate reads and writes during periods of heavy use. This should improve performance for rendering antialiased scenes. Also, caching behavior for Z-buffer and stencil buffer reads has been optimized. As a result, certain special effects like stencil shadow volumes will see substantially better performance.

Other Features

One of the biggest bottlenecks in many rendering applications is related to managing of the vertex data and feeding the pipeline efficiently. Typically, individual vertices are maintained in the client memory and copied repeatedly every time they are needed. The new GPU allows for creation of vertex arrays and index arrays in persistent server-side memory, dramatically improving the efficiency of data transfers. Refer to sections "Retained Data Model" on page 47 and "Vertex Array Objects (VAOs)" on page 50 for more details. Further, the UltimateVision GPU maintains a cache of transformed vertices. When you render the same vertex twice, the GPU uses the already transformed version of this vertex from its cache. Refer to sections "Element Arrays" on page 51 and "Vertex Caches" on page 51 for details on how to best leverage this feature.

Fill-limited applications can often benefit from another feature of UltimateVision graphics: Hyper-Z III hidden-surface removal, which subdivides the depth buffer into blocks of 8x8 pixels. Taking advantage of the spatial coherence of depth values, this feature compresses depth values within each block to get fairly high compression ratios (up to 24:1) and to dramatically lower the bandwidth requirements to and from the depth buffer. In addition, since each of the 8x8 pixel block is equipped with a flag that indicates if the block has been cleared without necessarily touching each and every pixel within the block, this features provides the "Fast Z clear" capability. Finally, each block contains a block-wide near-Z value, which is updated every time any of the pixels is written. This feature permits the fragment engine to ignore any fragments that fall within a block but fail the block-wide Z test (this is called "overdraw"). Only if the fragment is in front of the block-wide near-Z, does the entire block get fetched into the cache and decompressed, and individual depth values are inspected for detailed tests. Conceptually, this functionality is equivalent to a two-level hierarchical depth test and further reduces depth buffer traffic. Refer to section "Hyper-Z Depth Test" on page 54 to find out how your application can take further advantage of this feature.

Multitexturing allows several different textures (up to 16 on Onyx4 systems) to be bound independently and applied to the same primitive. OpenGL allows combining individual textures in a fairly arbitrary fashion. For example, level-of-detail and light maps can both be easily implemented with multitexturing. Such effects can also be reproduced with a multipass approach: the base texture is applied first, followed by the first detail texture, and so on. However, multitexturing is substantially faster because it does not need to transform or rasterize the same primitive multiple times; thus, multitexturing gets substantial transform rate savings.

UltimateVision systems offer a Programmable Jitter Multi-Sampling (PJMS) system, referred to as SmoothVision 2.0, for full-scene antialiasing (FSAA), which can dramatically decrease edge jaggedness and other aliasing artifacts. Two, four, or six samples per pixel can be computed on a per-pixel basis. Unlike super-sampling approaches used by similar chips, which essentially render the scene at a higher resolution, PJMS improves the image quality with fewer performance penalties related to increased fill/bandwidth requirements. In addition, the UltimateVision FSAA implementation also inspects the values in the Z-buffer to minimize shimmering artifacts at the interface of adjacent and intersecting polygons.

Further improvements in rendering speed are achieved using techniques similar to the Hyper-Z depth buffer. The framebuffer is subdivided into 8x8 blocks that are compressed losslessly to save the bandwidth. Finally, SmoothVision 2.0 provides gamma correction for color gradients making color ramps appear much smoother than with any other graphics accelerator.

GPU Programming

One of the most radical changes that sets UltimateVision graphics apart from any of its predecessors in the Onyx line is the ability to tailor operations of the transform engine and/or the pixel processing engine. In the previous architectures, the graphics pipeline provided fixed functionality that was determined during the chip design phase and frozen in microcode, which was fairly immutable. The old design allowed for multiple configurations selectable by setting OpenGL state. For example, the raster engine could be instructed to choose among several available blending operations or combine multiple textures in one of several possible ways. However, the number of these features was relatively small and if the feature was absent (for example, an esoteric blending function), there was little that could be done (short of implementing it with multipass algorithms, which substantially reduce performance).

An UltimateVision GPU, on the other hand, permits the application developer to extend the standard GPU functionality by writing custom programs which leverage the chip's circuitry in an arbitrary fashion. This allows the programmer to add new functionality (for example, new blending operation) or to implement an enhanced version of an existing operation. For example, see the section "Texture Lookup Tables (TLUTs)" on page 52 for a description on how to implement commonly desirable functionality that was not provided by the designers of the new chip but can be easily engineered using custom programs. In essence, with custom routines the application programmer is limited by the hardware capabilities of the chip but not by the fact that some functionality was not provided at a high-level interface (for example, OpenGL). Complex programs can be created to achieve desirable visual effects like warping of the original geometry, bump mapping, bi-cubic texture filtering, ARB imaging extensions, and life-like cinematic rendering.

An UltimateVision GPU can be programmed at the two different stages of the graphics pipeline:

- The geometry engine, which applies transformations and lighting calculations to vertex data
- The rendering engine, which operates on rasterized pixels

The associated programs are referred to as vertex and fragment programs, respectively.

In their most basic form, vertex programs and fragment programs are specialized assembly code. Instructions are provided to perform arithmetic and logic operations on the respective engine's ALU (for example, `MAD` for Multiply and Add, `DP3` for Dot Product Vec3) or to sample textures (for example, `TEX` to fetch a suitably interpolated texture value into a register, `LRP` to interpolate between two values). Once created, such special purpose programs can be loaded into the GPU circuitry using OpenGL. The programmed behavior will then be applied to any geometry sent down the graphics pipe instead of the standard fixed pipeline operations. Multiple programs can be compiled, loaded into the GPU, and bound when needed.

The obvious downside to writing long, complex programs is that they do impact performance. The more instructions in the vertex program, the lower the transform rate (that is, fewer triangles per seconds). Long vertex programs will negatively affect the transform rate.

Vertex Programs

The Programmable Transform & Light (T&L) unit allows the application developer to take full control over vertex processing. As each set of vertex data (coordinates, colors, texture coordinates, and the like) passes through the unit, an arbitrary set of manipulations can be applied to it. For example, a simple flat quad-mesh can have its height distorted with a vertex program to contain dynamic sinusoidal "ripples" that simulate wave propagation. In this way, compute-intensive simulations can be offloaded from the CPU by leveraging the heavily pipelined GPU design to get higher overall performance (the host sends the same flat mesh for each frame; all distorting is done by the GPU). Similarly, custom versions of **`glTexGen*()`** or mesh generation functionality can be implemented by inserting computed values of texture coordinates to reduce bandwidth requirements (the CPU only needs to send the control vertex coordinates).

The following are typical applications of vertex programs:

- Warping geometry (for example, ripples on water, key-frame interpolation, skinning)
- Generating tangents and binormals for bump mapping
- Orienting billboards
- Auto-generating texture coordinates

- Non-photorealistic rendering (NPR)

The following is a sample of a simple vertex program that explicitly transforms each incoming vertex coordinate from the model to screen space and passes through the vertex color unchanged:

```
!!ARBvp1.0
# Constant Parameters
PARAM mvp[4] = { state.matrix.mvp }; # modelview + proj
# Per-vertex inputs
ATTRIB inPosition = vertex.position;
ATTRIB inColor = vertex.color;
# Per-vertex outputs
OUTPUT outPosition = result.position;
OUTPUT outColor = result.color;

# Transform the vertex component by component
DP4 outPosition.x, mvp[0], inPosition;
DP4 outPosition.y, mvp[1], inPosition;
DP4 outPosition.z, mvp[2], inPosition;
DP4 outPosition.w, mvp[3], inPosition;
# Use the per-vertex color specified
MOV outColor, inColor;

END
```

The resulting text of the program is stored as a string (an array of Glubytes) and subsequently loaded into the geometry engine:

```
glGenProgramsARB(1, &vertexProgram);
glBindProgramsARB(GL_VERTEX_PROGRAM_ARB,
vertexProgram);
glProgramStringARB(GL_VERTEX_PROGRAM_ARB,
GL_PROGRAM_ASCII_ARB
strlen(programString),
programString);
/* check error output */
glEnable(GL_VERTEX_PROGRAM_ARB);
```

The maximum length of a vertex program on UltimateVision systems is 64K instructions, but keep in mind that vertex program execution speed is inversely related to the length of the program. Thus, keep the programs short. Fragment programs and vertex programs are limited to the number of hardware registers provided by the chip designers and

temporary storage cannot be virtualized; therefore, effective storage allocation is of critical importance here.

Fragment Programs

Once the vertices are transformed and lit by the T&L unit, the primitives they represent are rasterized and the resulting pixels together with their attributes (color, depth, texture coordinates, and the like) are passed along to the fragment program as "fragments." A fragment program takes its input (color and texture coordinates, for example) and produces a single RGBA value as output to be inserted into the framebuffer and other render targets. In a simple case, the color will be simply blended with the textures and will replace the previous value in the framebuffer, but programmability of the unit allows for arbitrarily complex operations to be applied instead. For example, having offset the texture coordinates somewhat to produce a simple motion blur effect, a fragment program can apply the same texture several times.

The following are typical applications of fragment programs:

- Per-pixel lighting
- Bump mapping
- Convolutions
- Reflection/refraction
- Special texture filters (for example, summed area table)
- Blending for IBR
- Complete IBR evaluation
- Motion blur

Fragment programs also allow coarser models to be used compared with older architectures. The programs can do so because the lighting effects can be interpolated across large stretches of primitives without introducing lighting artifacts.

The following code sample implements a very simple fragment program that modulates the incoming color with a texture:

```
!!ARBfp1.0
# Simple program to show how to code the default texture environment
ATTRIB tex = fragment.texcoord;      #first set of texture coordinates
```

```
ATTRIB col = fragment.color.primary; #interpolated color
PARAM zero = {0, 0, 0, 0};
OUTPUT outColor = result.color;
TEMP tmp;
TEMP tmp2;
TEMP tmp3;
TXP tmp, tex, texture, 2D;         #sample the texture
TXP tmp2, tex, texture, 2D;        #sample the texture again
ADD tmp3, tmp, tmp2;               #add together = tex *two
MUL outColor, tmp3, col;           #perform the modulation
END
```

The following code shows how the program is loaded into the fragment engine:

```
glGenProgramsARB(1, &fragmentProgram);
glBindProgramsARB(GL_FRAGMENT_PROGRAM_ARB,
vertexProgram);
glProgramStringARB(GL_FRAGMENT_PROGRAM_ARB,
GL_PROGRAM_ASCII_ARB
strlen(programString),
programString);
/* check error output */
glEnable(GL_FRAGMENT_PROGRAM_ARB);
```

The maximum length of a fragment program on UltimateVision systems is 64 instructions but, just as with vertex programs, performance (specifically, fill rate) generally will be inversely related to the length of the program.

Multipipe Programming

Because of the single-system-image design of UltimateVision graphics systems, developing multiprocessor and multipipe applications on these platforms is similar to developing on a single-CPU , single-GPU system. A single executable can access all data in memory at once and render to all GPUs at the same time with multiple draw threads. A single debugger can attach to the executable to view the whole application state. This dramatically differs from code development on a cluster, where each executable is a different process on a different node, which needs a separate debugger to determine program faults. This is not to mention resolving inter-process timing issues, which are a challenge on a shared memory machine but next to impossible to fix on distributed systems.

Scaling Dimensions

Multiple pipes allow a visualization application to scale many different aspects of its performance:

- Rendering speed
- Display resolution
- Data size

Rendering Speed Scaling

The dimension most apparent is rendering performance. More GPUs directly translate into more polygon transform and fill performance to view larger data sets at interactive rates. As data sets grow, more graphics bricks can be added with the SGI systems growing along with the data. In addition to polygon transform and fill performance, scalability also includes pixel scalability to provide large-scale display surfaces for theatres, caves, or just a single wall of a conference room. Equally important, more GPUs adds increased pixel shader and vertex shader performance; over time this will become the most relevant and important measure of performance.

Resolution Scaling

Pixel scalability provides the capability of moving beyond 1M–9M pixel desktop displays to tens or hundreds of Mpixel display walls to provide high resolution across almost any size display surface. Scalability also does not have a limit to a single dimension. A single display surface might need polygon and fill scalability with multiple GPUs to achieve the desired frame rate. At the same time, the number of pixels can be scaled to make a 4x4 display wall, where each of the 16 display surfaces on the wall have multiple GPUs driving it.

Data Size Scaling

By adding GPUs, GPU local memory is scaled. This can allow larger 3D textures to be cached in a distributed fashion and re-assembled after rendering (Monster Mode). Likewise, it allows more polygons to be cached in the GPU, where performance is highest. Adding GPUs also adds bandwidth, which enables the GPU cached data to be updated more rapidly overall. This paradigm of viewing GPU memory as a local cache will become increasingly important and widely used, particularly as vertex and pixel shaders allow more complex and higher-level algorithms to be applied to the data.

Scaling Techniques

There are many techniques for scaling visual computing problems, each having different benefits for different problem domains. The following are the most common ways to combine GPUs:

Technique	Decomposition dimension
2D tiling (screen decomposition)	Screen space
3D tiling (database decomposition)	World space
4D tiling (DPLEX, temporal decomposition)	Time
State decomposition	Graphics state
Eye-based decomposition (stereo)	Eye view
Task division	Tasks

Note that any of these techniques can be arbitrarily combined with others. For example, four sets of eight GPUs could be grouped in a 2D tile of four quadrants with each quadrant using data subdivision.

Scaling Tools

As SGI has developed expertise in building scalable compute and visualization architectures, it has been clear that tools to enable developers to build applications easily leveraging the power of scalability were critical. Over the past 10 years, SGI has been developing and improving on these toolkits to the point that there is a full range of toolkits that enable scalable applications. The choice of which toolkit to use depends on many factors including the following:

- Application architecture
- Target platform
- Level of scalability desired
- Amount of control you are willing to give to the toolkit

In general, supporting multiple pipes is a data management problem. Leveraging multiple processes running on multiple processors feeding multiple OpenGL streams on multiple GPUs is critical to getting true scaling of performance or image quality. The following items are required:

- Multiple processes for dedicated processors to feed the graphics pipes
- Data culled into chunks that are appropriate for a given GPU
- Multiple rendering streams provided by the application
- Database paging processes that potentially run asynchronously to keep the rendering pipeline filled

This section describes some tools provided by SGI to aid in this effort.

OpenGL Multipipe

At the simplest level, SGI offers the OpenGL Multipipe product, which is designed to run a traditional single-pipe application across multipipe outputs. This level of API gives you moderate scalability with very little impact to the design and architecture of the existing

application. Any single-pipe application will see moderate scalability when run with OpenGL Multipipe. To increase scalability, there are some simple guidelines to follow that allow OpenGL Multipipe to further increase application scalability. These guidelines entail some OpenGL coding idioms that generally do not affect single-pipe performance but allow OpenGL Multipipe to run more efficiently. For example, using smaller localized display lists allows OpenGL Multipipe to cull out many of the display lists on a pipe that will not be visible on that pipe. For more details on additional scalability optimizations, see the OpenGL Multipipe website (http://www.sgi.com/software/multipipe/).

OpenGL Multipipe SDK

A step further beyond OpenGL Multipipe is to have your application directly address all the graphics pipes in the system rather than leaving this up to OpenGL Multipipe. This will provide a boost in performance and scalability by sending multiple streams of OpenGL to the graphics pipes rather than a single OpenGL stream that is characteristic of a single-pipe application. The OpenGL Multipipe SDK toolkit provides a framework for you to manage multiple draw threads rendering to multiple pipes or GPUs. The OpenGL Multipipe SDK framework determines when and which GPU to render to while handling all the details necessary to manage graphics contexts and matrix stacks. The application is still responsible for all rendering, but many of the complexities of multipipe rendering are offloaded to OpenGL Multipipe SDK. For more details on scaling applications with OpenGL Multipipe SDK, see the OpenGL Multipipe SDK website (http://www.sgi.com/software/multipipe/sdk/).

OpenGL Performer

While OpenGL Multipipe SDK can allow an application to linearly scale over many GPUs, there are still many details that need to be considered to achieve the highest level of scalability. OpenGL Performer, which is a scene-graph based API, is designed to enable this ultimate level of scalability and performance. OpenGL Performer offloads all rendering responsibility from the application. An application describes the scene to be visualized to OpenGL Performer and OpenGL Performer decides the most efficient way to render that scene across all the available CPUs and GPUs in the system. For more details on scaling applications with OpenGL Performer, see the OpenGL Performer website (http://www.sgi.com/software/performer/).

Platform-Specific Porting Information

OpenGL programming on UltimateVision platforms bears a lot of similarity to programming other SGI platforms. Most basic features like sending vertex data within **glBegin()**/**glEnd()** code blocks or frame control with **glXSwapBuffers()** calls are identical. However, there UltimateVision platforms also offer a much broader suite of tools and features never before available in the Onyx product line. In addition, some features that worked well on earlier platforms may have more attractive equivalents. Subsequent sections identify many of such issues and suggest preferred ways of implementing commonly desired functionality.

InfiniteReality Programming Features Versus UltimateVision Features

This section identifies some programming features and compares their support on InfiniteReality systems with their support on UltimateVision systems. Table 5-1 provides a brief comparison.

Table 5-1 InfiniteReality Versus UltimateVision Programming Features

Feature	InfiniteReality Systems	UltimateVision Systems
Multisampling	X	slightly different
Shadow mappping	X	slightly different
Depth texture	X	slightly different
LOD bias	X	slightly different
Clip mapping	X	not supported
Video format compiler	X	not supported
8-subsample antialiasing	X	not supported

Table 5-1 InfiniteReality Versus UltimateVision Programming Features **(continued)**

Feature	InfiniteReality Systems	UltimateVision Systems
Luminance visuals	X	not supported
Detail texture (`SGIS_sharpen_texture`)	X	X (using multitexturing)
Bicubic texture filtering	X	X (using fragment programs)
ARB imaging features	X	X (using fragment programs)
Pixel texture (TLUTs)	X	X (using fragment programs)

Chapter 6, “Performance Tips” describes how you can use some of these and other features of UltimateVision systems to enhance performance.

Moving from Xsgi to XFree86

With the introduction of Onyx4 systems, SGI began using a standard distribution of the X Window System from the X Consortium called XFree86. XFree86, an implementation that was originally developed to support windowing on Unix systems based on Intel386 processors, has become a de facto standard for Linux graphical environments. Its support now extends well beyond x86 CPUs and covers other processor architectures as well as numerous graphics cards. XFree86 Version 4.2.0 implements X Version 11 Release 6.3 (X11R6.3) and is distributed for free under a fairly unrestricted license from the X Consortium. It replaces the SGI proprietary implementation of the X server marketed as Xsgi.

While XFree86 has many similarities to the commercial implementation of X Windows used in earlier SGI systems, there are several important differences that are worth mentioning. In particular, application programmers will find that the way visuals are selected may be somewhat more cumbersome. The most critical difference is that the server configures certain aspects of the graphics card’s behavior at startup time using setup files. For example, in order to reconfigure the framebuffer for stereo display or to enable full-scene antialiasing, the X server needs to be restarted using different statically configured files. This is in contrast to dynamically querying and requesting new visuals from within the application through GLX requests. Additional differences exist in the interfaces with external devices. This is true of keyboards and mice as well as other

external devices like the video compositor. This section describe these and related issues in more detail.

The remainder of this section describes general aspects of the configuration process then aspects that are specific to Onyx4 systems.

The XFree86 Configuration File

This section describes details of configuring an XFree86 server. It describes the process, locations of relevant files, syntax, and the most relevant tags (`Section` tags). Much of the information that is included here may be found in your system's man pages and on the Web, but it is provided here for your convenience. Of particular importance are the `XFree86`(1) and `XF86Config`(5) man pages.

Typically, the X server on your machine is started automatically when the machine boots. However, it is possible to restart the server at any time after that without necessarily rebooting your machine. Typically, high-level utilities that keep track of system state are used for that purpose, but it is also possible to run the `XFree86` command directly. Some of the options available to you for that purpose are described in section "Restarting Graphics" on page 23. As the X server is starting, it will locate and parse a configuration file, which contains information about your system resources like keyboards, mice, graphics cards, monitors, and so on. The section "Configuration File Format" on page 28 describes these somewhat intricate details.

Restarting Graphics

Typically, XFree86 is started automatically at boot time using the `xdm` utility of the X display manager. SGI systems provide the `startgfx` utility, which turns the `windowsystem` configuration flag on (using `chkconfig`) and executes `xdm`. The simplest way of restarting graphics is to type the following on the command line:

```
$ (/usr/gfx/stopgfx; sleep 3; /usr/gfx/startgfx) &
```

Note that this command-line entry kills all the existing windows and logs you out. The following keyboard sequence is another viable alternative:

```
Ctrl-Alt-Backspace
```

In either case, `xdm` uses the contents of `/etc/X11/XF86Config-4` to configure the new instance of the server; so, you can place your changes directly there. However, it is also

possible to use a different configuration file. Specifying a different configuration file is particularly useful if you want to switch among different configurations (for example, between `XF86Config.Stereo` and `XF86Config.FSAA`) or to debug a newly created configuration.

To use a different configuration file, you can enter the `XFree86` command directly. Among its numerous options, the `–xf86config` option allows you to specify explicitly which configuration file to use. This option will work for any file when the server is run as `root` (that is, with real UID `0`) or for files relative to a directory in the configuration search path for all other users as described in the section "Configuration File Location" on page 25.

For a description of the full set of options for the `XFree86` command, see the `XFree86`(1) man page on your system.

If you routinely use multiple different framebuffer configurations, you may want to choose from a number of pre-created configuration files, rather than renaming the selected one to `XF86Config-4` for the duration of the session. In this case, you will need to tell `xdm` which of the non-default configuration files to use. For example, if you created the file `/etc/X11/XF86Config-4.myStereo`, you will need to do the following:

1. Create a file named `/var/X11/xdm/Xservers.myStereo` and put the following entry in it:

```
:0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config-4.myStereo
```

   The value `0` refers to the first pipe.

2. Edit `/var/X11/xdm/xdm-config` and change `DisplayManager.servers` to `/var/X11/xdm/Xservers. myStereo`.
3. Restart graphics to put the changes into effect.

Alternatively, you can choose not to use `xdm` at all. You can start the X server by running `xinit` manually rather than automatically from `xdm` as shown in the following:

1. Log in as `root`.
2. Enter the following command sequence:

```
# /usr/gfx/stopgfx
# chkconfig xdm off
# cp /usr/bin/X11/X ~/.xinitrc
```

3. Transfer all startup programs from `.xsession/.sgisession` to `.xinitrc`.

4. Enter the following command:

```
# XFree86 -xf86config /etc/X11/XF86Config-4
```

If your graphics are not coming back, try the following:

- Check files `/var/log/XFree.*.log` for any error messages.
- From the network or the console, enter the following command sequence:

```
# /usr/gfx/stopgfx
# killall -9 xdm
# killall -9 XFree86
# /usr/gfx/startgfx
```

- Replace your current `XF86Config-4` file with one you are reasonably sure works (for example, freshly created with `XFree86 –config`). For more details, see section "Bootstrapping" on page 27.

Configuration and Run-Time Parameters

XFree86 supports the following mechanisms for supplying/obtaining configuration and run-time parameters:

1. Command-line options
2. Environment variables
3. The XFree86 configuration file
4. Auto-detection
5. Fallback defaults

The list of mechanisms is ordered from highest precedence to lowest. When the same information is supplied in more than one way, the highest precedence mechanism is used. Note that not all parameters can be supplied by all methods. The available command-line options and environment variables (and some defaults) are described in the `Xserver`(1) and `XFree86`(1) man pages.

Configuration File Location

When a non-root user starts the server, the following directories will be searched in attempt to find a configuration file:

/etc/X11/*cmdline*
/usr/X11R6/etc/X11/*cmdline*
/etc/X11/$XF86CONFIG
/usr/X11R6/etc/X11/$XF86CONFIG
/etc/X11/XF86Config-4
/etc/X11/XF86Config
/etc/XF86Config
/usr/X11R6/etc/X11/XF86Config.*hostname*
/usr/X11R6/etc/X11/XF86Config-4
/usr/X11R6/etc/X11/XF86Config
/usr/X11R6/lib/X11/XF86Config.*hostname*
/usr/X11R6/lib/X11/XF86Config-4
/usr/X11R6/lib/X11/XF86Config

The variable fields are defined as follows:

cmdline	A relative path (with no ".." components) specified with the –xf86config command-line option. For details, see the section "Restarting Graphics" on page 23.
$XF86CONFIG	The relative path (with no ".." components) specified by that environment variable
hostname	The machine's hostname as reported by **gethostname()**

If, on the other hand, the server was started by the root user, a much less restricted set of paths is considered, as shown by the following:

cmdline
/etc/X11/cmdline
/usr/X11R6/etc/X11/*cmdline*
$XF86CONFIG
/etc/X11/$XF86CONFIG
/usr/X11R6/etc/X11/$XF86CONFIG
$HOME/XF86Config
/etc/X11/XF86Config-4
/etc/X11/XF86Config
/etc/XF86Config
/usr/X11R6/etc/X11/XF86Config.*hostname*
/usr/X11R6/etc/X11/XF86Config-4
/usr/X11R6/etc/X11/XF86Config
/usr/X11R6/lib/X11/XF86Config.*hostname*
/usr/X11R6/lib/X11/XF86Config

The variable fields are defined as follows:

cmdline	The path specified with the –`xf86config` command-line option (which may be absolute or relative)
`$XF86CONFIG`	The path specified by that environment variable (absolute or relative)
`$HOME`	The path specified by that environment variable (usually the home directory)
hostname	The machine's hostname as reported by **gethostname()**

Most examples in this document assume for simplicity that your default configuration file will be `/etc/X11/XF86Config-4` while alternative files are also placed in `/etc/X11`, but this certainly does not need to be the case. In particular, you are well advised to debug any new variant configuration files elsewhere before moving them into the standard location.

Bootstrapping

Onyx4 systems provide a `gen-XF86Config` script for generating a default `XF86Config-4` file. The default installation runs the `gen-XF86Config` script during system startup to verify that a valid `XF86Config-4` file is installed.

The following steps describe how to use the script:

1. Stop graphics.
2. Make a backup copy of the file with the following entry:

   ```
   # mv /etc/X11/XF86Config-4 my_backup_config_file
   ```

3. Invoke the script as follows:

   ```
   # /etc/X11/gen-XF86Config
   ```

4. Start graphics.

The behavior of `gen-XF86Config` is controlled by the following `chkconfig` variables:

- `xf86config_autoconfig`
- `xf86config_autoreplace`

Consult the contents of `gen-XF86Config` for details and useful tricks. You may also find the `XFree86`(1) man page helpful.

Error Logging

The X server sends its standard and error outputs into a log file. The default location of the log file is the following file:

```
/var/log/XFree86.*.log
```

The * in the preceding path matches the X server number (numbered from `0`). Inspect this file to verify that the new configuration you created worked or to identify any problems and get clues as to what could have gone wrong.

Configuration File Format

The configuration file is composed of a number of sections, which may be present in any order. Each section has the following form:

```
Section “SectionName”
    keyword value
    . . .
EndSection
```

The following sections are supported:

`Files`	File pathnames
`ServerFlags`	Server flags
`Module`	Dynamic module loading
`InputDevice`	Input device description
`Device`	Graphics device description
`VideoAdaptor`	Xv video adaptor description
`Monitor`	Monitor description
`Modes`	Video mode descriptions
`Screen`	Screen configuration
`ServerLayout`	Overall layout
`DRI`	Direct Rendering Infrastructure configuration
`Vendor`	Vendor-specific configuration

The `ServerLayout` sections are at the highest level. They bind together the input and output devices that will be used in a session. The input devices are described in the `InputDevice` sections. Output devices usually consist of multiple independent components (for example, and graphics board and a monitor). These multiple components are bound together in the `Screen` sections, and these sections are referenced by the `ServerLayout` section. Each `Screen` section binds together a graphics board and a monitor. The graphics boards are described in the `Device` sections, and the monitors are described in the `Monitor` sections.

The following are noteworthy syntax-related items:

- Configuration file keywords are case-insensitive, and "_" characters are ignored. Most strings (including `Option` names) are also case-insensitive and insensitive to white space and "_" characters.
- Each configuration file entry is usually a single line in the file. An entry consist of a keyword, which is possibly followed by one or more arguments. The number of arguments and their types depend on the keyword. The following are the argument types:

Integer	An integer number in decimal, hex or octal
Real	A floating point number
String	A string enclosed in double quote marks (")

  Note that hex integer values must be prefixed with "0x", and octal values with "0".
- A special keyword called `Option` may be used to provide free-form data to various components of the server. The `Option` keyword takes either one or two string arguments. The first is the option name, and the optional second argument is the option value. Some commonly used option value types include the following:

Integer	An integer number in decimal, hex or octal
Real	A floating point number
String	A sequence of characters
Boolean	A boolean value
Frequency	A frequency value

  Note that all `Option` values, not just strings, must be enclosed in quotes.
- Boolean options may optionally have a value specified. When no value is specified, the option's value is `TRUE`. The following boolean option values are recognized as `TRUE`:

```
1, on, true, yes
```

The following boolean option values are recognized as FALSE:

```
0, off, false, no
```

If an option name is prefixed with “No”, then the option value is negated.

For example, the following option entries are equivalent:

```
Option “Accel” “Off”
Option “NoAccel”
Option “NoAccel” “On”
Option “Accel” “false”
Option “Accel” “no”
```

- Frequency option values consist of a real number that is optionally followed by one of the following frequency units:

  ```
  Hz, k, kHz, M, MHz
  ```

  When the unit name is omitted, the correct units will be determined from the value and the expected range of the value. When using frequency option values, specify the units to avoid any errors in a system-derived value.

ServerLayout Section

A server layout is the top-level description and represents the binding of one or more screens (`Screen` sections) and one or more input devices (`InputDevice` sections) to form a complete configuration. In multipipe configurations, it also specifies the relative layout of the heads. For example, it can indicate that the four available screens should be arranged in a 2x2 grid. Multiple `ServerLayout` sections may be present, but only one of them will be active within a session. A specific `ServerLayout` section can be explicitly specified from the X server command using the `–layout` option. If no such option is specified, it is assumed that the system has only one screen, and the first available `Screen` section will be used together with two active (core) input devices. Each `Screen` line specifies a `Screen` section identifier, and optionally, its relative position to other screens. For example, the following shows how a two-pipe system could be described:

```
Section “ServerLayout”
    Identifier “Layout 1”                    Unique identifier
    Screen “Screen SG-1”
    Screen “Screen SG-2” RightOf “Screen SG-1”
    InputDevice “Keyboard 1” “CoreKeyboard”   Do not change.
    InputDevice “Mouse 1” “CorePointer”       Do not change.
    InputDevice “Mouse 2” “SendCoreEvents”    Do not change.
EndSection
```

For format details, see the `XF86Config`(5) man page.

Screen Section

The `Screen` section groups multiple components required to display graphical output. It allows you to select combinations of the monitor, resolution (modeline), graphics card, and bit depth that you require as shown in the following example:

```
Section “Screen”
   Identifier “Screen SG-0”  Identifies Screen in ServerLayout section.
   Device “SGI SG-0”     References the graphics board for this Screen.
   Monitor “Generic Monitor” References monitor used by the Screen.
   DefaultDepth 24
   SubSection
       Depth 24
       Modes “1280x1024”
   EndSubSection
EndSection
```

The `Identifier` tag is used by a higher-level section, typically the `ServerLayout` section, to refer to this particular screen. Any number of screen sections may be present. A specific screen configuration can be specified from the X server command line with the –`screen` option. There should be as many `Screen` sections within your configuration file as there are pipes, each with an unique ID.

For resolutions less than 1024x768, the X server might choose to automatically use a larger virtual screen. To prevent this, simply add keyword `Virtual` as shown in the following example:

```
SubSection
    Depth 24
    Modes “800x600”
    Virtual 800 600      Size of the virtual desktop
EndSubSection
```

Device Section

The `Device` section of the XFree86 configuration file describes with the driver configuration of graphics boards. Therefore, most of graphics mode configuration would take place there. A simple example of a configuration file that sets the default settings and enables full-scene antialiasing with six samples:

```
Section “Device”
     Identifier “SGI SG-4”  Identifies board in the Screen section.
     Driver “fglrx”         Driver name (Do not change.)
     BusId “PCI:2:4:0”      Bus location of the graphics card
     Option “FSAAScale” “6” Place the list of options here.
EndSection
```

Naturally, each pipe will have its own `Device` section. The number of `Device` sections is arbitrary and some of them can be unreferenced, but there must be at least one. The `Identifier` and `Device` tags are required; all others are optional. The `Identifier` value needs to uniquely identify each graphics card (there can be no clashes).

`BusId` options describes the bus location of the graphics card. The following steps describe an easy way to obtain the `BusID` values suitable for use in the `Device` section:

1. Stop graphics.
2. Enter the following command:

   ```
   $ /usr/X11R6/bin/XFree -logfile /dev/null -scanpci -verbose
   ```

3. Restart graphics when done.

Monitor Section

The `Monitor` section specifies the characteristics of your monitor(s). There will be one such section for each monitor type that can be connected to your system. A monitor description together with a device description is used to define a screen. The following illustrates a sample `Monitor` section.

```
Section "Monitor"
    Identifier "GenericMonitor" Identifies monitor in Screen section.
    HorizSync 30-96             Multisync (important for stereo)
    VertRefresh 50-160          Multisync
    Mode "640x480"
        DotClock 25.175
        HTimings 640 664 760 800
        VTimings 480 491 493 525
    EndMode
    ModeLine "1024x768i" 45 1024 1048 1208 1264 768 776 784 817
             Interlace
EndSection
```

The `HorizSync` value specifies the horizontal sync frequencies in kHz. This can be a range of values (as shown in the preceding example) or a comma-separated list enumerating valid settings. Your monitor manual should list these settings in the technical specifications section of the user's manual. Handle the `VertRefresh` value in a parallel manner.

Modeline Section

Modelines are used to describe timings/resolution of a particular output device. They are defined in the `Monitor` section and used in the `Screen` section.

The format of a modeline is as follows:

`ModeLine` "*format-name*" *dot-clock-in-MHz H-active-end H-front-porch-end H-sync-end V-active-end V-front-porch-end V-sync-end options*

The horizontal values are specified in pixels and the vertical values are specified in lines. The dot clock value is calculated by multiplying the total pixels by the total lines and the

vertical refresh frequency. For example, for 1280x1024 format, you may use the following specification:

```
ModeLine “1280x1024” 108 1280 1328 1440 1688 1024 1025 1028 1066 +hsync
         +vsync
```

The dot clock value is set to 108 MHz (1688*1066*60 = 107964480, which is rounded to 108 MHz). Therefore, the actual refresh rate will be 60.02 Hz.

Onyx4 Configuration Specifics

Note: Systems running IRIX 6.5.22 or 6.5.23 should have SGI patch 5448 or greater.

Onyx4 systems require a number of system-specific configuration settings. The following sections describe step-by-step instructions on how to configure your system to enable a specific feature. It is possible to combine some configuration settings together. For example, you can instruct your system to enable both stereo and full-scene antialiasing. The following topics are described:

Configuring Pipes for Stereo

To configure your pipes for stereo display, use the following procedure:

1. Create a stereo-specific configuration file as shown in the following:

```
$ cp /etc/X11/XF86Config-4 XF86Config-4.Stereo
```

2. Modify the `Device` section in file `XF86Config-4.Stereo` for each pipe as shown in the following:

```
Section “Device”
    Identifier “SGI SG-4” Identifies board in the Screen section.
    Driver “fglrx”        Driver name (Do not change.)
    BusId “PCI:2:4:0”     Bus location of the graphics card
    Option “Stereo” “1”
    Option “StereoSyncEnable” “1”
EndSection
```

   Other relevant options include the following:

   `Option “MonitorLayout”` *layout*

   The value *layout* can be one of the following:

“CRT, CRT”	All analog
“TMDS, NONE”	Digital for the first channel, analog for the second
“CRT, NONE”	Analog for the first channel, disabled for the second

   Note that it is not possible to configure “TMDS” for the second channel.

3. Add stereo modelines to the `Monitor` section as shown in the following:

```
Section “Monitor”
    Identifier “Stereo Monitor” Monitor in Screen section
    HorizSync 30-96             Multisync (important for stereo)
    VertRefresh 50-160          Multisync
    Modeline “1024x768@96” 103.5 1024 1050 1154 1336 768 771 774 807
    Modeline “1280x1024@96” 163.28 1280 1300 1460 1600 1024 1027
              1033 1063
    Modeline “1024x768@100” 113.309 1024 1096 1208 1392 768 769 772
              814
```

```
    Modeline “1024x768@120” 139.054 1024 1104 1216 1408 768 769 772
              823 +hsync + vsync

    Modeline “1280x1024@100” 190.960 1280 1376 1520 1760 1024 1025
              1028 1085 +hsync +vsync

    Mode “1280x1024_96s_mirage”

        DotClock 152.928

        Htimings 1280 1330 1390 1500

        Vtimings 1024 1026 1030 1062

    EndMode

EndSection
```

4. Make sure your monitor supports the high horizontal rate and modify `HorizSync` value in the `Monitor` section as follows:

   ```
   HorizSync 22-105
   ```

5. As shown in the following, modify your `Screen` section to use one of the newly created modelines:

   ```
   Section “Screen”

       Identifier “Screen SG-0” The screen in ServerLayout section

       Device “SGI SG-0”          References graphics board for screen.

        Monitor “Stereo Monitor” References monitor used by screen.

       DefaultDepth 24

       SubSection

           Depth 24

           Modes “1280x1024”

           Modes “1280x1024@96”

       EndSubSection

   EndSection
   ```

6. Create a file named `/var/X11/xdm/Xservers.Stereo` and put the following entry in it:

   ```
   :0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config-4.Stereo
   ```

7. Edit `/var/X11/xdm/xdm-config` and change `DisplayManager.servers` to `/var/X11/xdm/Xservers.Stereo`

8. Restart graphics to put the changes into effect.

9. Run an application that uses stereo.

   Even after you configure the stereo correctly, there is not going to be a stereo sync signal until you run an application that actually uses stereo. For example, you can enter the following command to run such an application:

   ```
   $ ivview /usr/share/data/models/X29.iv
   ```

   Right click and activate stereo on the preferences pane.

10. If your changes did not take effect, try moving this file.

    The modeline/resolution can be overridden by the `/var/X11/Xvc/SG2_`*N*`_TimingTable`, which is created by `xsetmon`, where *N* is the pipe number.

Configuring a Mirage Projector

To configure a Mirage projector, use the same instructions described in the section "Configuring Pipes for Stereo" on page 35, but use the following modeline:

```
Mode "1280x1024_96s_mirage"
    DotClock 152.928
    Htimings 1280 1330 1390 1500
    Vtimings 1024 1026 1030 1062
EndMode
```

Enabling Full-Scene Antialiasing

There are two possible modes or full-scene antialiasing (FSAA) for Onyx4 systems:

- Have the system to force all applications to use FSAA.
- Allow the application to request a visual that has multisample buffers.

Setting the option "`FSAAScale`" to `1` enables visuals with multisample buffers. Setting the option "`FSAAScale`" to `2`, `4`, or `6` forces all window to have either 2,4, or 6 sample antialiasing, respectively.

To enable full-scene antialiasing, use the following steps:

1. Make an antialiasing version of the configuration file as follows:

   ```
   $ cp /etc/X11/XF86Config-4 XF86Config-4.AA
   ```

2. Modify the `Device` section in file `XF86Config-4.AA` for each pipe as follows:

```
Section “Device”
    Identifier “SGI SG-4”  Identifies the board in Screen section.
    Driver “fglrx”         Driver name (Do not change.)
    BusId “PCI:2:4:0”      Bus location of the graphics card
    Option “FSAAScale” “2” Allowable values: 1, 2, 4, or 6
EndSection
```

   Other relevant options include the following:

```
Option “FSAADisableGamma” “yes” number
Option “FSAACustomizeMSpos” “yes” number
Option “FSAAMSPOSX0” ...
```

3. Create a file named `/var/X11/xdm/Xservers.AA` and put the following line in it:

```
:0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config-4.AA
```

4. Edit `/var/X11/xdm/xdm-config` and change `DisplayManager.servers` to `/var/X11/xdm/Xservers.AA`.
5. Restart graphics to put the changes into effect.

Enabling Overlay Planes

To enable overlay planes, use the following steps:

1. Make an overlay planes version of the configuration file as follows:

```
$ cp /etc/X11/XF86Config-4 XF86Config-4.OP
```

2. Modify the `Device` section in file `XF86Config-4.OP` for each pipe as follows:

```
Section “Device”
    Identifier “SGI SG-4” Identifies the board in Screen section.
    Driver “fglrx”        Driver name (Do not change.)
    BusId “PCI:2:4:0”     Bus location of the graphics card
Option “OpenGLOverlay” “On”
EndSection
```

3. Create a file named `/var/X11/xdm/Xservers.OP` and put the following entry in it:

```
:0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config-4.OP
```

4. Edit `/var/X11/xdm/xdm-config` and change `DisplayManager.servers` to `/var/X11/xdm/Xservers.OP`.
5. Restart graphics to put the changes into effect.

Configuring Dual Channels

The following steps describe how to configure dual channels:

1. Create a dual-channel version of the configuration file as follows:

   ```
   $ cp /etc/X11/XF86Config-4 XF86Config-4.DC
   ```

2. Modify the `Device` section in file `XF86Config-4.DC` for each pipe as follows:

   ```
   Section “Device”
       Identifier “SGI SG-4” Identifies the board in Screen section.
       Driver “fglrx”        Driver name;(Do not change.)
       BusId “PCI:2:4:0”     Bus location of the graphics card
       Option “DesktopSetup” mode
   EndSection
   ```

   The *mode* value can be one of the following:

   “0x00000100” Clone the managed area.

   “0x00000200” Scale the managed area by 2 horizontally.

   “0x00000300” Scale the managed area by 2 vertically.

   Note that the same mode must be specified for all the pipes managed by a particular X server. Both channels will have the same display resolution.

   Also, note that only the first channel is capable of digital (TMDS) output; the second channel has analog output only. To configure analog output for both channels, append the following entry in the `Device` section:

   ```
   Option “MonitorLayout” “CRT, CRT”
   ```

3. When using monitor cables that do not conform to the VESA Display Data Channel (DDC) standard, append the following entry in the `Device` section to allow for proper display configuration:

   ```
   Option “NoDDC”
   ```

4. Create a file named `/var/X11/xdm/Xservers.DC` and put the following entry in it:

   ```
   :0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config-4.DC
   ```

5. Edit `/var/X11/xdm/xdm-config` and change `DisplayManager.servers` to `/var/X11/xdm/Xservers.DC`.
6. Restart graphics to put the changes into effect.

Configuring Edge Blending or Channel Overlapping

The following steps describe how to configure edge blending or channel (screen) overlapping:

1. Configure dual channels, as described in a preceding section, before attempting to configure screen/channel overlapping.

   Your total managed area would be the sum of the width of both channels minus the number of pixels specified in the overlap option.

2. Make a channel-overlap version of the configuration file as follows:

   ```
   $ cp /etc/X11/XF86Config-4 XF86Config-4.CO
   ```

3. Modify the `Device` section in file `XF86Config-4.CO` for each pipe to contain the following:

   ```
   Section “Device”
       Identifier “SGI SG-4” Identifies the board in Screen Section.
       Driver “fglrx”        Driver name (Do not change.)
       BusId “PCI:2:4:0”     Bus location of the graphics card
       Option “DesktopSetup” “0x00000200”
       Option “ScreenOverlap” 256 The number of pixels of overlap
   EndSection
   ```

4. Restart graphics.

Multiple Keyboards/Mice and Multiple X Servers

To configure a multiuser environment, multiple keyboards/mice and multiple X servers, perform the following steps:

1. Make a backup copy of `/etc/ioconfig.conf` and then edit the original file to remove all keyboard and mouse entries.
2. Attach all the intended keyboards and mice (PS2 devices may require a reboot), then enter the following command:

   ```
   $ /sbin/ioconfig -f /hw
   ```

3. Verify that the links were created in `/dev/input`.
4. In `/etc/X11`, make multiple copies of `XF86Config-4` named `XF86Config-4.server`*N*, where *N* refers to each server, starting with 0.
5. Edit each `XF86Config-4.server`*N* file and change the following line in the `InputDevice` section:

   ```
   Option “Device” “/dev/mouse”
   ```

   For `XF86Config-4.server0`, change the line to the following:

   ```
   Option “Device” “/dev/input/mouse”
   ```

   For the other `XF86Config-4.server`*N* files, change the line to the following:

   ```
   Option “Device” “/dev/input*N*/mouse”
   ```

6. For each of the `XF86-4.server`*N* files , edit the `Device`, `Screen`, and `ServerLayout` sections to reflect the pipes that are to be managed by a particular server.
7. Create a file named `/var/X11/xdm/Xservers.`*N*`key` and add the following entries, one for each of the servers:

```
:0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config-4.server0
-devdir /dev/input
:1 secure /usr/bin/X11/X :1 -xf86config /etc/X11/XF86Config-4.server1
-devdir /dev/input1
```

   ...

8. Edit `/var/X11/xdm/xdm-config` and change the `DisplayManager.servers` setting to `/var/X11/xdm/Xservers.`*N*`key`
9. Restart graphics or, to be on the safe side, reboot.

Configuring the ImageSync IS1 Card

Perform the following general steps to configure the ImageSync IS1 card:

1. Physically connect the card to the proper components of the system.

   The following manual describes this procedure:

   SGI Onyx Next Generation Scalable Graphics Platform User's Guide
2. Install the Onyx4 patches for your OS level.
3. Enable SwapReady.

   See the subsection Enabling SwapReady.

4. Enable FrameLock.

   See the subsection Enabling FrameLock.

Enabling SwapReady

Use the `xsetmon` GUI to enable SwapReady in each X screen (pipe) connected to the IS1 board. Perform the following steps:

1. Open a `winterm` window on the X screen 0 desktop, then use the `–target` option to point to the X screen that you want `xsetmon` to configure while using `–display` to indicate where you want the GUI to appear. For example, in order to configure pipe 1 from pipe 0 use the following entry:

   ```
   $ xsetmon -target :0.1 -display :0.0
   ```

2. Using the `xsetmon` GUI, set the following options:

   ```
   Swap Buffers on Vertical Blank: On
   GLX Swap Barrier Extension: On
   ```

3. Click **Load** button to save the configuration.
4. Repeat steps 1-3 for each pipe connected to the IS1 board.

Enabling FrameLock

If an application uses an API such as OpenGL Performer, then you can force each pipe to begin a new frame of video output on the vertical sync pulse that the IS1 board outputs. Use the following procedure to configure your system for this kind of operation:

1. If the application uses a startup script like `RUN`, use a text editor to set the following environment variable in the script:

   ```
   setenv __GL_SYNC_TO_BLANK 1
   setenv PFVIDEORATE v<vertical_refresh_rate>
   ```

   The *vertical_refresh_rate* value corresponds to the value of your current video format in Hz (for example, 60, 72, and so on).

2. Start the application.
3. If you want the pipe outputs to frame-lock regardless of whether an ImageSync-aware application is running (for example, compositor configurations), you can open a `winterm` window on each X screen and set the variables shown in step 1 in every pipe, or else, add them to a `.xsession` file to enable ImageSync automatically when a user logs in.
4. To enable sync on the compositor, use the `sgcombine` GUI.

Configuring the Compositor

The following are preconditions for compositor configuration:

- The ImageSync card must be wired and configured.
- You must have version 1.5 of the compositor.
- Both the DVI and USB ports must be wired.
- The latest patch list must be installed.

Use the following procedure to configure four compositors with two pipes each:

1. Edit the `Device` section as follows if you want to use an analog monitor on Head 1:

   ```
   Option “MonitorLayout” “TMDS, CRT”
   Option “DesktopSetup” “0x00000100”
   ```

2. Add compositor modelines to the `Monitor` section as shown in the following:

   - Stereo Modeline for the Compositor

   ```
   ModeLine “1280x1024_96” 164.966 1280 1300 1460 1600 1024 1026 1029
            1074
   ```

   - Mono Modeline for the Compositor (1280x1024)

   ```
   ModeLine “1280x1024” 108 1280 1328 1440 1688 1024 1025 1028 1066
            -hsync -vsync
   ```

   - Modeline for the Compositor (1600x1200)

   ```
   ModeLine “1600x1200_60” 161.5356 1600 1664 1856 2160 1200 1201 1204
            1250 -hsync -vsync
   ```

3. Modify the `Screen` section so that you can use the correct modeline. The following is an example:

   ```
   Modes “1280x1024@60”
   ```

4. As shown in the following entries, load the `.cmb`, previously created with `/usr/gfx/sgcombine`, and send it to the compositors with `setmon`:

```
/usr/gfx/setmon -p 0 -S /usr/gfx/ucode/STINGRAY/my_1280_1024_60_2_tiles
/usr/gfx/setmon -p 2 -S /usr/gfx/ucode/STINGRAY/my_1280_1024_60_2_tiles
/usr/gfx/setmon -p 4 -S /usr/gfx/ucode/STINGRAY/my_1280_1024_60_2_tiles
/usr/gfx/setmon -p 6 -S /usr/gfx/ucode/STINGRAY/my_1280_1024_60_2_tiles
```

Alternatively, you can use `sgcombine` to indicate which screen to configure and where to display the GUI, respectively, as shown in the following:

`/usr/gfx/sgcombine -target` #*xserver.screen* **`-display`** #*xserver.screen*

5. Restart graphics to put the changes into effect.

Configuring an IBM T221 9-Megapixel Monitor

The following is a sample `Monitor` section for an IBM T221 9-Megapixel monitor:

```
Section “Monitor”
    Identifier “Bertha”
    VendorName “IBM”
    ModelName “T221”
    HorizSync 22-105
    VertRefresh 9-64
# Modeline description: pxlclk Hres Hfp Hsnc Htot Vres Vfp Vsnc Vtot
# 1/2 screen 1920x1200
    Modeline “960x1200_56” 70.98 960 968 1000 1056 1200 1200 1202 1206
    Modeline “960x1200_64” 82.00 960 968 1000 1056 1200 1201 1203 1212
# 1/4 screen 3840x2400
    Modeline “960x1200_56” 104.78 960 968 1000 1056 2400 2402 2404 2424
# full screen 1920x1200 or ¿ screen 3840x2400
    Modeline “1920x1200_41” 104.78 1920 1928 1960 2112 1200 1201 1203
              1212
# 1/2 screen 3840x2400
    Modeline “1920x2400_20” 99.54 1920 1928 1980 2044 2400 2402 2405
              2435
    Modeline “1920x2400_24” 119.40 1920 1928 1980 2044 2400 2402 2405
              2435
    Modeline “1920x2400_25” 125.02 1920 1928 1980 2044 2400 2402 2405
              2434
    Modeline “1920x2400_30” 149.31 1920 1928 1980 2044 2400 2402 2405
              2435
```

```
# stuff from EDID, 3840x2400 at 12.7 Hz
    Modeline “3840x2400” 148.00 3840 3944 4328 4816 2400 2401 2404 2418
EndSection
```

Adding or Removing Pipes

The following procedure describes how you add or remove pipes:

1. Stop graphics.
2. Make a backup configuration file as shown in the following:

   ```
   $ mv /etc/X11/XF86Config-4 /etc/X11/XF86Config-4.bak
   ```

   Use `chkconfig` to make sure that `xf86config_autoconfig` is `on` and `cf86config_autoreplace id` is `off`.

3. Generate a new configuration file as follows:

   ```
   $ /etc/x11/gen-XF86Config
   ```

4. Start the graphics using the newly generated `XF86Config-4` file to configure the X server.

Performance Tips

Certain graphics features take advantage of the underlying hardware more effectively than others, and UltimateVision graphics systems are no exception. It is important to be able to pick the optimal alternative among the many available. For example, you can feed your data model to the GPU within a **glBegin()/glEnd()** sequence, but vertex arrays are likely to result in better performance. The following subsections describe techniques that will improve the performance of your application:

- "Retained Data Model"
- "Display List Optimizer"
- "Vertex Array Objects (VAOs)"
- "Element Arrays"
- "Vertex Caches"
- "Strip Consolidation"
- "Texture Lookup Tables (TLUTs)"
- "Fragment Program Tips"
- "Hyper-Z Depth Test"
- "Timing"
- "Pixel Formats"

Retained Data Model

If possible, use retained data mode instead of immediate mode. Retained mode avoids the need to repeatedly copy geometry and instructions from the host to the graphics board memory for every frame, thus eliminating the system bus bandwidth bottleneck. In addition, the system virtualizes most retained mode structures and pages them in and out as needed. While using retained mode is always a good practice, it is particularly important with UltimateVision systems, which use PCI as the interconnecting bus. Given

that the theoretical bandwidth for the PCI bus is only 266 MB/s (and effective bandwidth lower yet) geometry traffic should be optimized at all costs.

Vertex array objects (VAOs) and display lists are the two most attractive ways of providing vertex data to the graphics engine. Geometry-limited scenes that fit entirely within GPU memory should see around 100 Mtris/s. For details, see the performance benchmarks at http://www.ati.com/developer/ATIVertexArrayObject.pdf.

Given the choice between the VAOs and display lists, VAOs are preferable from the performance standpoint. However, if your application already uses display lists, which, unlike vertex arrays, can store instructions as well as data, they can be converted to vertex arrays completely transparently, as described in the section "Display List Optimizer" on page 48.

Applications that can benefit the most from switching to retained mode need to satisfy the following conditions:

- Geometry is static or changes infrequently.
- Geometry data fits within the GPU memory.
- Application is geometry-limited, rather than fill-limited.

For more details, see the document
http://www.ati.com/developer/ATIVertexArrayObject.pdf.

A similar rule applies to dealing with textures as well as vertex data: using texture objects will generally improve performance by eliminating repeated texture downloads. Also, large or numerous texture objects will be virtualized—that is, swapped in and out as needed by the application—without the need for explicit texture management on the developer's part. This can be a mixed blessing, though. When texture memory is oversubscribed, the resource manager will make its own decisions as to what and when to swap out; this may not suit your application well.

Display List Optimizer

The display list optimizer will attempt to turn display lists into equivalent representations that can be processed more efficiently by the underlying hardware (for example, VAOs). If the process is successful, the resulting geometry will run optimally; otherwise, performance will be similar to that in the immediate mode.

You can determine if your display lists fell within the fast path by analyzing the debug output of the optimizer. This can be accomplished by setting your environment variable `FGL_DLOPT_INFO` to `1`. The OpenGL driver detects the variable and prints information about the degree of optimization applied to the display list. After setting the environment variable, expect printouts of the following form:

```
List 2153: DL_OPT_CONV_TO_DRAWARRAY: num_begins = 2128, num_prims =
2128
List 2153: DL_OPT_CONNECT_DRAWARRAYS: num_prims = 532
List 2153: DL_OPT_CONV_TO_HW: num_prims = 532, num_hw_prims = 532,
bytes_cached = 1327872
```

`DL_OPT_CONV_TO_DRAWARRAY` and `DL_OPT_CONNECT_DRAWARRAYS` are stages in the optimization process. `DL_OPT_CONV_TO_HW` indicates the most optimized hardware-accelerated lists that are cached in graphics memory.

The following are two of the operations that the optimizer attempts:

- Converting **glBegin()**/**glEnd()** pairs into vertex array objects
- Connecting and compressing contiguous arrays of the same primitive type (`POINTS`, `LINES`, `LINESTRIP`, `TRIANGLES`, and `TRIANGLE_STRIP`). Note that `QUADS` are not optimized.

Certain data organizations will prevent the optimizer from doing its job. For example, using vertex attributes (colors, normals, texture coordinates) with layouts that vary within the same display list will inhibit optimizations. Therefore, it is best to select a single one even if it means padding data within some lists by replicating per-primitive data. Obviously, this will only work if the replicated geometry data still fits within GPU memory.

It is important to realize that in certain rare situations the optimizer can actually degrade performance. For example, converting very short **glBegin()**/**glEnd()** sequences (for example, containing a single `QUAD`) into vertex array objects can result in a performance penalty due to costly VAO setup.

Since the optimizer's performance relies heavily on the content of the scene, it may not be possible to realize any gains for a specific application. As a matter of fact, some applications may be slower. If this is the case for your application, you can disable the display list optimizer entirely by setting the environment variable `FGL_DEBUG_DLOPT` to `0`.

In order to use these environment variables, systems running IRIX 6.5.22 or 6.5.23 should have SGI patch 5448 or greater.

Vertex Array Objects (VAOs)

Traditionally, vertex arrays are allocated in the client memory and are copied to the server memory whenever they are needed. This substantial overhead can be eliminated by placing such data directly in the persistent server-side memory with help of `ATI_vertex_array_object` and `ATI_element_array` OpenGL extensions.

Object buffers can be of any standard type (for example, `GL_NORMAL_ARRAY`) and are allocated with a **glNewObjectBufferATI()** call, which returns a handle to the created buffer for subsequent use. If your vertex array data changes during the course of the application, you can update it with a call to **glUpdateObjectBufferATI()**, but if it is done frequently, you should explicitly declare such an object as dynamic with the help of **glArrayObjectATI()** with the parameter usage set to `GL_DYNAMIC_ATI`.

VAOs can be rendered in the same way as standard OpenGL vertex arrays: **glDrawElements()**, **glDrawArrays()**, and **glDrawElement()** are all available, even though the first option is generally the best choice. It is best to create one VAO per entire model part; that is, allocate one object, store, and set up all the vertex components for a part within that object, as shown in the following code:

```
struct Vertex {
float c[4];
float n[3];
float v[3];
    };

    Vertex *vertices;
    int vertexCount;
    unsigned int vobj;

    vertices = new Vertex[vertexCount];
    /* fill in vertex data */

    vobj = glNewObjectBufferATI(sizeof(Vertex) * vertexCount,
            (void *)vertices, GL_STATIC_ATI);

    glEnableClientState(GL_VERTEX_ARRAY);
```

```
glArrayObjectATI(GL_VERTEX_ARRAY, 3, GL_FLOAT, sizeof(Vertex),
                 vobj, offsetof(Vertex, v));
glEnableClientState(GL_NORMAL_ARRAY);
glArrayObjectATI(GL_NORMAL_ARRAY, 3, GL_FLOAT, sizeof(Vertex),
                 vobj, offsetof(Vertex, n));
glEnableClientState(GL_COLOR_ARRAY);
glArrayObjectATI(GL_COLOR_ARRAY, 4, GL_FLOAT, sizeof(Vertex), vobj,
                 offsetof(Vertex, c));

glDrawArrays(GL_TRIANGLES, 0, vertexCount);
```

Element Arrays

Another way to improve performance for geometry-limited applications is to collect as many vertices into a single VAO and provide an additional array of indices that reference it. In this way, any vertices that occur multiple times (for example, shared by adjacent primitives) are only transformed once and cached by the GPU.

UltimateVision graphics systems provide a mechanism for treating a vertex array as a set of indices into another vertex array using the `ATI_element_array` OpenGL extension. Such an array of indices behaves like any other vertex array in just about all respects. In particular, an element array can be created in the persistent server-side memory as a VAO. The only difference in treatment of element arrays is that there is a dedicated means of rendering indexed objects using **glDrawElementArrayATI()** or **glDrawElementArrayRangeATI()**, instead of the traditional **glDrawElements()**.

For more information and code samples, see the document http://www.ati.com/developer/ATIVertexArrayObject.pdf.

Vertex Caches

A GPU on UltimateVision graphics systems has the ability to cache a small number of recently transformed vertices. Applications can leverage this feature by suitably rearranging the input geometry. For example, drawing a non-indexed grid of 120x120 quads will explicitly transform each vertex up to four times (as each internal vertex belongs to four different quads). You will get a significant improvement in performance by using indexed quadstrips and splitting the grid into short swaths that can be cached between adjacent rows, thus requiring them to be transformed only once.

For this technique to work, the following two conditions must be satisfied:

- Vertices need to be specified in an indexed VAO.
- Strips need to be short enough so that they are still in the cache when they are needed again.

For example, a 120x120 indexed quadstrip requires sending 120 vertices for a single row. By the time the second row reaches the vertex processor, the previous row will already have been flushed from the cache. Splitting the grid into short spans of 6 quads (12 vertices) assures that the bottom vertices of the first row are still in the vertex cache when the second row is sent, avoiding costly vertex cache misses.

Strip Consolidation

Consolidating multiple, short, adjacent triangle strips into a single longer strip can also result in improved performance. Converting each strip to a VAO individually replicates the setup overhead for indexed objects. Therefore, reducing the number of strips improves performance.

Suppose you have two strips with the following vertices:

1, 2, 3, 4, 5 and 6, 7, 8, 9

The merged strip looks like the following:

1, 2, 3, 4, 5, 5, 6, 6, 7, 8, 9

The new strip contains some degenerate triangles (for example—5, 5, 6), that the GPU skips. Overall, the gain from avoiding multiple VAO setups is larger than the cost of skipping the degenerate triangles.

Texture Lookup Tables (TLUTs)

Unlike InfiniteReality graphics, the UltimateVision GPU does not support the `SGI_texture_color_table` extension and, thus, texture lookup tables are not implemented intrinsically. However, if your application relies on this functionality, you

can still achieve the same effect by writing a simple fragment program that implements dependent texture lookups. Specifically, you need to do the following:

1. Initialize and bind a 1D texture, A, containing the lookup table values.
2. Initialize and bind the original data texture, B.

   For volume data sets, this will likely be a 3D GL_LUMINANCE texture.
3. Create and bind a fragment program that will use the interpolated value of B to index into the lookup texture A.

The following code shows a simple ARB_fragment_program for dependent texture lookup:

```
!!ARBfp1.0
TEMP volume;
TEX volume, fragment.texcoord[1], texture[1], 3D;
TEX result.color, volume, texture[0], 1D;
END
```

In the preceding code, a temporary register, volume, is used to store the interpolated value of texture B (which is bound to texture unit 1). Subsequently, the 1D texture A (bound to texture unit 0) is sampled using the value stored in volume.

Fragment Program Tips

While fragment programs provide great flexibility, they can affect performance. Long, complex programs can slow things down considerably, particularly for applications that are fill-limited (like volume rendering). The most critical issue to good performance of a custom routine is limiting the total number of instructions.

When writing performance-critical fragment programs, use the following guidelines:

- Minimize the number of operations.
- Minimize the number of constants and temporaries.
- The MOV operation is not needed and should be avoided.
- Reorder operations to interleave arithmetic operations with texture sampling to hide the latency of texture access.

- Wherever possible, try accessing textures in a cache-friendly fashion.
- Unused texture samples are ignored by the driver. So, issuing a TEX operation and ignoring the results will not actually sample the texture.

The driver will attempt to collapse and optimize the code (for example, temporary registers) but it is always safer to start with a good initial code.

Hyper-Z Depth Test

Fill-limited applications pose significant demands on the bandwidth between the framebufffer, depth buffer, and the fragment processor. Not only does each pixel have to be read and written, but also, occlusion calculations need to be performed and require data to be read and written from the depth buffer. Hyper-Z circuitry eliminates processing overhead for fragments which are known to be occluded by looking at the depth coordinates of 8x8 blocks of the depth buffer. In addition, the rendering engine applies early Z termination to fragments that fail the block-wide Z test. None of such fragments are rendered at all. This may result in substantial savings, particularly if texturing or complex custom shading operations are being applied. While such conditional evaluation tends to stall the pipeline by introducing an additional overhead of flushing the pipeline, the approach may be justified as the shader complexity increases.

As a result, fill rate for a scene that is drawn from front to back can be much higher (by as much as 10x) than that for back-to-front or in random order. Performance-sensitive developers may choose to pre-sort their geometry to take advantage of this feature. This approach can be particularly effective if the viewpoint does not change dramatically from frame to frame so that visibility sorting can only be repeated periodically (obviously, the scene does not have to be perfectly sorted for each view, but the more the better). It may be best to update the sort in a separate asynchronous thread to eliminate dependency of each frame on the sorting time. Even a coarse, object-level sort may be very effective in taking advantage of Hyper-Z.

Timing

On UltimateVision systems, the first operation to touch the framebuffer followed by a **glXSwapBuffers()** and then a **glFlush()** call is where the pipe blocks for the vertical retrace (if sync to vblank is enabled), not at a **glXSwapBuffers()** call. Therefore, if you are

timing a frame, use the following calls after **glXSwapBuffers()** and before the timing check:

```
glNormal3f(0, 0, 1); glFlush();
```

The call **glNormal3f(**0, 0, 1**)** is not enough alone.

Pixel Formats

Transferring pixel data (for example, textures) to and from the video memory can have a substantial impact on the performance of your application. The following are some rules that will help you avoid many pitfalls:

- For `GL_UNSIGNED_BYTE` reading, the `GL_RGB` and `GL_RGBA` formats are the current fastest modes.
- For `GL_UNSIGNED_BYTE` writing, the fastest modes are `GL_RGBA` followed by `GL_RGB`.
- For `GL_DEPTH_COMPONENT` writing, the `GL_UNSIGNED_SHORT` type is the fastest type.
- When downloading `GL_RGB` and `GL_RGBA GL_UNSIGNED_BYTE` data to a texture, it is best to download to a `GL_RGBA` internal format.

For performance numbers, see Appendix A, “Performance Benchmarks”.

Performance Benchmarks

This appendix lists a collection of performance benchmarks for Onyx4 systems. The following benchmarks are included:

Each section cites the operating system level used for the associated benchmarks. Depending your operating system level, the rates you see on your system may differ from those shown in this appendix but the rates shown here should be representative.

glDrawPixels() and glReadPixels() Rates

This section charts the rates for **glDrawPixels()** and **glReadPixels()**. The benchmarks were captured on an IRIX 6.5.24 operating system plus patch 5448. Table A-1 shows the rates for **glDrawPixels().**

Table A-1 Rates for **glDrawPixels()**

Resolution	Format	Rate
512x512	RGBA UBYTE	36.044800 Mpixels/s, 144.179200 Mbytes/s
512x512	RGBA USHORT	18.022400 Mpixels/s, 144.179200 Mbytes/s
512x512	RGBA USHORT4444	48.059736 Mpixels/s, 96.119472 Mbytes/s
512x512	RGBA USHORT5551	0.735608 Mpixels/s
512x512	RGBA UINT	3.067642 Mpixels/s, 49.082280 Mbytes/s
512x512	RGBA FLOAT	5.767168 Mpixels/s, 92.274688 Mbytes/s
512x512	BGRA UBYTE	36.044800 Mpixels/s, 144.179200 Mbytes/s
512x512	BGRA USHORT	18.022400 Mpixels/s, 144.179200 Mbytes/s

Table A-1 Rates for **glDrawPixels() (continued)**

Resolution	Format	Rate
512x512	BGRA USHORT4444	48.059736 Mpixels/s, 96.119472 Mbytes/s
512x512	BGRA USHORT5551	0.748983 Mpixels/s
512x512	BGRA UINT	3.067642 Mpixels/s, 49.082280 Mbytes/s
512x512	BGRA FLOAT	5.767168 Mpixels/s, 92.274688 Mbytes/s
512x512	RGB UBYTE	24.029868 Mpixels/s, 72.089600 Mbytes/s
512x512	RGB UBYTE332	0.728178 Mpixels/s
512x512	RGB USHORT	3.067642 Mpixels/s, 18.405856 Mbytes/s
512x512	RGB UINT	3.067642 Mpixels/s, 36.811712 Mbytes/s
512x512	RGB FLOAT	3.035351 Mpixels/s, 36.424220 Mbytes/s
512x512	LUMINANCE UBYTE	0.807727 Mpixels/s
512x512	LUMINANCE USHORT	48.059736 Mpixels/s, 96.119472 Mbytes/s
512x512	LUMINANCE UINT	12.537322 Mpixels/s, 50.149288 Mbytes/s
512x512	LUMINANCE FLOAT	36.044800 Mpixels/s, 144.179200 Mbytes/s
512x512	LUMINANCEALPHA UBYTE	48.059736 Mpixels/s, 96.119472 Mbytes/s
512x512	LUMINANCEALPHA USHORT	36.044800 Mpixels/s, 144.179200 Mbytes/s
512x512	LUMINANCEALPHA UINT	6.135285 Mpixels/s, 49.082280 Mbytes/s
512x512	LUMINANCEALPHA FLOAT	16.962258 Mpixels/s, 135.698064 Mbytes/s
512x512	DEPTH UBYTE	0.805470 Mpixels/s
512x512	DEPTH USHORT	48.059736 Mpixels/s, 96.119472 Mbytes/s
512x512	DEPTH UINT	12.014934 Mpixels/s, 48.059736 Mbytes/s

Table A-1 Rates for **glDrawPixels() (continued)**

Resolution	Format	Rate
512x512	DEPTH FLOAT	36.044800 Mpixels/s, 144.179200 Mbytes/s
512x512	STENCIL UBYTE	0.942348 Mpixels/s
512x512	STENCIL USHORT	41.194056 Mpixels/s, 82.388112 Mbytes/s
512x512	STENCIL UINT	12.014934 Mpixels/s, 48.059736 Mbytes/s
512x512	STENCIL FLOAT	22.181416 Mpixels/s, 88.725664 Mbytes/s

Table A-2 shows the rates for **glReadPixels()**.

Table A-2 Rates for **glReadPixels()**

Resolution	Format	Rate
512x512	RGBA UBYTE	28.835840 Mpixels/s, 115.343360 Mbytes/s
512x512	RGBA USHORT	0.876469 Mpixels/s, 7.011754 Mbytes/s
512x512	RGBA USHORT4444	0.871173 Mpixels/s, 1.742347 Mbytes/s
512x512	RGBA USHORT5551	0.871173 Mpixels/s, 1.742347 Mbytes/s
512x512	RGBA UINT	0.850615 Mpixels/s, 13.609836 Mbytes/s
512x512	RGBA FLOAT	0.873813 Mpixels/s, 13.981014 Mbytes/s
512x512	BGRA UBYTE	26.214400 Mpixels/s, 104.857600 Mbytes/s
512x512	BGRA USHORT	0.876469 Mpixels/s, 7.011754 Mbytes/s
512x512	BGRA USHORT4444	0.871173 Mpixels/s, 1.742347 Mbytes/s
512x512	BGRA USHORT5551	0.871173 Mpixels/s, 1.742347 Mbytes/s
512x512	BGRA UINT	0.850615 Mpixels/s, 13.609836 Mbytes/s
512x512	BGRA FLOAT	0.871173 Mpixels/s, 13.938775 Mbytes/s

Table A-2 Rates for **glReadPixels() (continued)**

Resolution	Format	Rate
512x512	RGB UBYTE	32.039820 Mpixels/s, 96.119464 Mbytes/s
512x512	RGB UBYTE332	0.887257 Mpixels/s, 0.887257 Mbytes/s
512x512	RGB USHORT	0.889995 Mpixels/s, 5.339970 Mbytes/s
512x512	RGB UINT	0.873813 Mpixels/s, 10.485760 Mbytes/s
512x512	RGB FLOAT	0.884535 Mpixels/s, 10.614420 Mbytes/s
512x512	LUMINANCE UBYTE	0.898313 Mpixels/s, 0.898313 Mbytes/s
512x512	LUMINANCE USHORT	0.903945 Mpixels/s, 1.807890 Mbytes/s
512x512	LUMINANCE UINT	0.903945 Mpixels/s, 3.615779 Mbytes/s
512x512	LUMINANCE FLOAT	0.903945 Mpixels/s, 3.615779 Mbytes/s
512x512	LUMINANCEALPHA UBYTE	0.889995 Mpixels/s, 1.779990 Mbytes/s
512x512	LUMINANCEALPHA USHORT	0.887257 Mpixels/s, 3.549026 Mbytes/s
512x512	LUMINANCEALPHA UINT	0.887257 Mpixels/s, 7.098053 Mbytes/s
512x512	LUMINANCEALPHA FLOAT	0.889995 Mpixels/s, 7.119960 Mbytes/s
512x512	DEPTH UBYTE	0.997780 Mpixels/s, 0.997780 Mbytes/s
512x512	DEPTH USHORT	0.994339 Mpixels/s, 1.988679 Mbytes/s
512x512	DEPTH UINT	0.994339 Mpixels/s, 3.977357 Mbytes/s
512x512	DEPTH FLOAT	1.001244 Mpixels/s, 4.004977 Mbytes/s
512x512	STENCIL UBYTE	1.015346 Mpixels/s, 1.015346 Mbytes/s
512x512	STENCIL USHORT	1.015346 Mpixels/s, 2.030693 Mbytes/s

Table A-2 Rates for **glReadPixels() (continued)**

Resolution	Format	Rate
512x512	`STENCIL UINT`	1.015346 Mpixels/s, 4.061386 Mbytes/s
512x512	`STENCIL FLOAT`	1.018934 Mpixels/s, 4.075737 Mbytes/s

Texture Download Rates

Table A-3 shows texture download rates for various texture formats. The benchmarks were captured on an IRIX 6.5.23 operating system plus patch 5448.

Table A-3 Texture Download Rates

Internal Format	Format	Value Type	Rate (Mtexels/s)
GL_RGB	GL_RGB	GL_UNSIGNED_BYTE	22.74
GL_RGBA	GL_RGBA	GL_UNSIGNED_BYTE	26.43
GL_RGBA	GL_RGBA	GL_UNSIGNED_SHORT _5_5_5_1	7.26
GL_LUMINANCE	GL_LUMINANCE	GL_UNSIGNED_BYTE	51.30
GL_LUMINANCE16	GL_LUMINANCE	GL_UNSIGNED_SHORT	34.12
GL_LUMINANCE_ALPHA	GL_LUMINANCE_ALPHA	GL_UNSIGNED_BYTE	23.52
GL_LUMINANCE16_ALPHA16	GL_LUMINANCE_ALPHA	GL_UNSIGNED_SHORT	16.13

Index

O

P

R

S

T

V

X