GLINT R4®

Programmer's Guide - Volume III

DRAFT ONLY

PROPRIETARY AND CONFIDENTIAL INFORMATION

3Dlabs®

GLINT R4®

Programmer's Guide - Volume III

PROPRIETARY AND CONFIDENTIAL INFORMATION

Issue 3

Proprietary Notice

The material in this document is the intellectual property of 3D*labs*. It is provided solely for information. You may not reproduce this document in whole or in part by any means. While every care has been taken in the preparation of this document, 3D*labs* accepts no liability for any consequences of its use. Our products are under continual improvement and we reserve the right to change their specification without notice. 3D*labs* may not produce printed versions of each issue of this document. The latest version will be available from the 3D*labs* web site.

3D*labs* products and technology are protected by a number of worldwide patents. Unlicensed use of any information contained herein may infringe one or more of these patents and may violate the appropriate patent laws and conventions.

3D*labs* is the worldwide trading name of 3D*labs* Inc. Ltd.

3D*labs*, GLINT R4 and PERMEDIA are registered trademarks of 3D*labs* Inc. Ltd.

Microsoft, Windows and Direct3D are either registered trademarks or trademarks of Microsoft Corp. in the United States and/or other countries. OpenGL is a registered trademark of Silicon Graphics, Inc. All other trademarks are acknowledged and recognized.

© Copyright 3D*labs* Inc. Ltd. 1999. All rights reserved worldwide.

Email: info@3dlabs.com
Web: http://www.3dlabs.com

3D*labs* Ltd.
Meadlake Place
Thorpe Lea Road, Egham
Surrey, TW20 8HE
United Kingdom
Tel: +44 (0) 1784 470555
Fax: +44 (0) 1784 470699

3D*labs* K.K
Shiroyama JT Mori Bldg 16F
40301 Toranomon
Minato-ku, Tokyo, 105, Japan
Tel: +81-3-5403-4653
Fax: +91-3-5403-4646

3D*labs* Inc.
480 Potrero Avenue
Sunnyvale, CA 94086,
United States
Tel: (408) 530-4700
Fax: (408) 530-4701

Change History

Document	Issue	Date	Change
160.4.3	1	1 Dec 99	First DRAFT Issue.
160.4.3	2	20 Jan 2000	Delta Programming example, many updates from P4 to R4
160.4.3	3	18 June 2001	various corrections 280100; fixed example values for stencil position and width, GID, in Initialization, removed index entries and last vestiges of FBReadMode, deleted windowbase references, corrected GID test control no longer in Windows reg, Stencil source data field, 140601;

Contents

Proprietary Notice i
Change History ii
Contents iii

FOG, ANTIALIAS AND ALPHA TEST 13-1

13.1 Fog Unit 13-1

13.1.1 Fog Index Calculation 13-1
13.1.2 Fog Table 13-2
13.1.3 Fog Application 13-3
13.1.4 FogMode register 13-4
13.1.5 Fog Example 13-5

13.2 Alpha Test Unit 13-7

13.2.1 Alpha Test 13-7
13.2.2 Registers 13-7
13.2.3 Alpha Test Example 13-8

FRAMEBUFFER READ/WRITE 14-1

14.1.1 Standard Framebuffer Read Operation 14-1
14.1.2 Framebuffer Read Span Operations 14-2
14.1.3 Merge-copy Span Operations 14-2

ALPHA BLENDING 15-1

15.1 Introduction 15-1

15.1.1 Alpha Blend Functions 15-1
15.1.2 Alpha Blend Registers 15-2

15.2 Source Blending Functions 15-2

15.2.1 OpenGL Alpha Blending 15-2

15.3 Destination Blending Functions 15-3

15.3.1 OpenGL Destination Blending 15-3
15.3.2 QuickDraw 3D Alpha Blending 15-4
15.3.3 Image Formatting 15-4
15.3.4 Registers 15-5
15.3.5 Chroma Testing 15-9
15.3.6 Alpha Blend Example 15-11

COLOR FORMAT AND LOGICAL OPS 16-1

16.1 Color and Alpha Formats 16-1

16.1.1 Color Dithering 16-4

16.1.2 Registers....16-5
16.1.3 Dither Example....16-6
16.1.4 3:3:2 Color Format Example....16-6
16.1.5 8:8:8:8 Color Format Example....16-6
16.1.6 Color Index Format Example....16-7
16.2 Logical Op Unit....16-7
16.2.1 High Speed Flat Shaded Rendering....16-7
16.2.2 Logical Operations....16-8
16.2.3 Registers....16-8
16.2.4 XOR Example....16-9
16.2.5 Logical Op and Software Writemask Example....16-10
FRAMEBUFFER WRITEMASKS....17-1
17.1.1 Software Writemasks....17-1
17.1.2 Hardware Writemasks....17-1
17.1.3 Registers....17-1
17.1.4 Software Writemask Example....17-1
17.1.5 Hardware Writemask Example....17-2
HOST OUT....18-1
18.1 Filtering....18-1
18.1.1 Filter Mode Example....18-1
18.1.2 Statistic Operations....18-2
18.1.3 Synchronization....18-3
18.1.4 Registers....18-3
18.1.5 Picking Example....18-5
18.1.6 Sync Interrupt Example....18-6
INITIALIZATION....19-1
19.1 Initializing GLINT R4....19-1
19.1.1 Reset and initialisation....19-1
19.2 System Initialization....19-2
19.2.1 PCI bus....19-2
19.2.2 Memory Configuration....19-2
19.2.3 Internal Video Timing Registers....19-3
19.2.4 Framebuffer Depth....19-3
19.2.5 Screen Width....19-4
19.2.6 Screen Clipping Region....19-4
19.2.7 Localbuffer and Framebuffer Configuration....19-4

19.2.8 Host Out Unit ... 19-5
19.2.9 Disabling Specialized Modes ... 19-6
19.3 Window Initialization ... 19-6
19.3.1 Color Format ... 19-6
19.3.2 Setting the Window Address and Origin ... 19-6
19.3.3 Writemasks ... 19-7
19.3.4 Enabling Writing ... 19-7
19.4 Application Initialization ... 19-8
PERFORMANCE TIPS ... 20-1
20.1 Block Writes ... 20-1
20.2 Fast double buffering in a window ... 20-2
20.3 Disable FB Reads per pixel if not required ... 20-2
20.4 Improving PCI bus bandwidth for Programmed I/O and DMA ... 20-2
20.5 PCI burst transfers under Programmed I/O ... 20-2
20.6 Using PCI Disconnect Under Programmed I/O ... 20-3
20.7 Using Bus Mastership (DMA) ... 20-3
20.8 Disabling units not in use ... 20-3
20.9 Clearing the localbuffer & framebuffer ... 20-3
20.10 Use of the Framebuffer (or Localbuffer) Bypass ... 20-4
20.11 Loading Registers in Unit Order ... 20-4
20.12 Avoiding Unnecessary Register Updates ... 20-4
20.13 Hardware and Software Context Dumps ... 20-4
20.14 Use the Memory Scratchpad Registers ... 20-4
20.15 Miscellaneous Tips ... 20-5
APPENDICES ... 21-1
21.1 Pseudocode Definitions ... 21-1
21.2 Delta (Manual) Programming Example ... 21-3
21.3 Interpolation Calculation ... 21-20
21.3.1 Color Gradient Interpolation ... 21-20
21.3.2 Register Set Up for Color Interpolation ... 21-20
21.3.3 Calculating Depth Gradient Values ... 21-21
21.4 Appendix F. Accurate Rendering ... 21-22
21.5 Glossary ... 21-34
VOLUME III INDEX ... 5

13

Fog, Antialias and Alpha Test

13.1 Fog Unit

The fog unit is used to blend the incoming fragment's color or Z (generated by the color DDA unit, and potentially modified by the texture unit) with a predefined fog color. Fogging can be used to simulate atmospheric fogging, and also to depth cue images.

Fog application has two stages:

1. derive the fog index for a fragment;
2. apply the fogging effect.

The fog index is a value which is interpolated over the primitive using a DDA in the same way color and depth are interpolated. The fogging effect is applied to each fragment using one of the equations described below.

Note: *Although fog values are linearly interpolated over a primitive they can be calculated on the host using either a linear fog function (typically for simple fog effects and depth cueing) or a more complex function e.g. an exponential function to model atmospheric attenuation..*

13.1.1 Fog Index Calculation

The fog index can be derived from specified fog values in **FStart**, **dFdX** and **dFdYDom**, or from the Depth DDA values. This option is selected with the *UseZ* bit in the **FogMode** register.

The fog DDA is used to interpolate the fog index (f) across a primitive. The mechanics are similar to those of the other DDA units, as the diagram below illustrates:

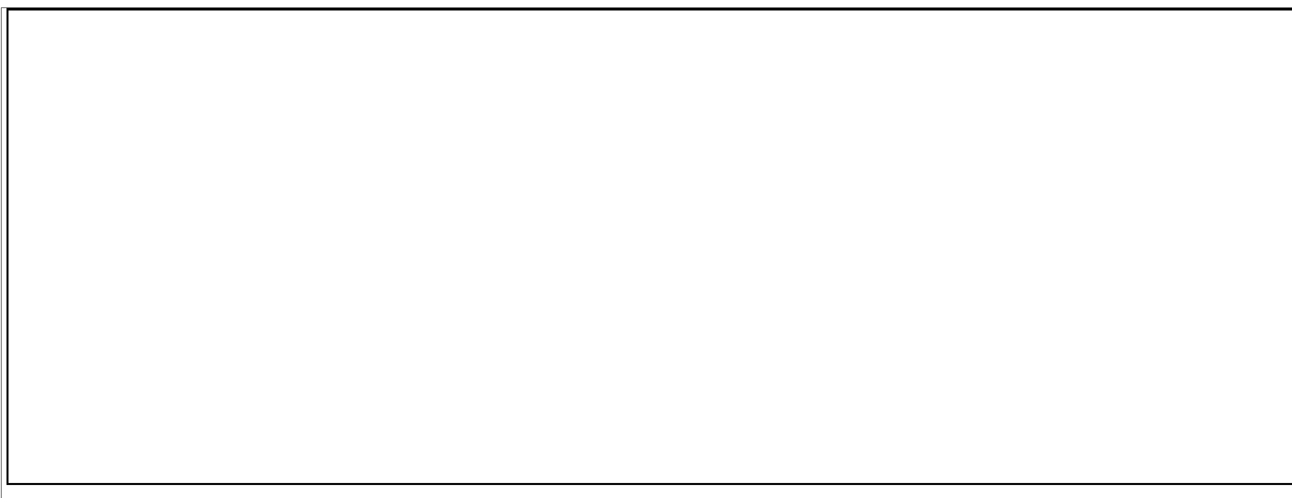


Figure 13-1 Fog Interpolation Over A Triangle

where:

- **dFdX** = Fog gradient in the X direction.
- **dFdyDom** = Fog gradient along the dominant edge of a primitive.

Note: For fogged lines the ***dFdx*** *delta is not required.*

The fog interpolation values (e.g. **Fstart**) are specified as 32bit fixed point numbers - the format is 2's complement with 10 bits integer and 22 bits fraction. However the derived fog index is an 8-bit fixed point number (0 bits integer, 8 bit fraction).

The DDA only exports a relatively narrow range (+511 to -512) compared to the range of depths so the software needs to be careful when setting up the DDA. There are four cases:

- If all the vertices are in the near range then the DDA should be set up to output 1.0 with a delta of 0.
- If all the vertices are in the far range then the DDA should be set up to output 0.0 with a delta of 0.
- If all the vertices are within the DDA's range then the DDA's parameters are set up as normal.
- One or more of the vertices are out of the DDA's range and must be clamped before the DDA's parameters are set up. (This will only occur on very large polygons which extend from near the eye point into the far distance.)

The result of clamping the input values to the DDA will be to change the effective position and width of the fog band (i.e. middle range), but this is unlikely to be noticeable. If it is noticeable then tessellating the polygon will solve the problem.

13.1.1.1 Z-controlled Fog

The fog value (direct or mapped via the table) can be derived from the interpolated Z value. If the UseZ bit is set in **FogMode** then the fog DDA is loaded by the Z DDA parameters and tracks the Z value over the primitive. The 2's complement 32 bit Z value from the DDA output is mapped to the 8 bit fog index as follows:

- Clamp Z from the DDA so it is greater than or equal to 0.
- Add in the **ZFogBias** and clamp again to be greater than or equal to 0.
- Shift right by ZShift amount.
- Clamp against 255 so the result is less than or equal to 255. This is the fog index.

The bias sets a Z value below which no blending occurs. The scale value selects the range (as a power of 2) beyond which the fog color is used (because the fog index is set to 255).

13.1.2 Fog Table

Initially, the fog values populate a span register and an increment register tracks progress along the Dominant edge. Both f-controlled and Z-controlled fog produce the 8-bit index values which can be directly applied to interpolation or stored as a table for use in

producing more complex (non-linear) fogs with host intervention. The Fog Table is selected using the *Table* bit in the **FogMode** register.

The fog table is organised as 256 x 8 so the 8 bit input fog index is mapped to an 8 bit output fog index. The fog table is held in the **FogTable(0)** to **FogTable(63)** registers and each register loads 4 entries at a time. **FogTable0**, byte 0 loads the mapping for fog index 0, byte 1 for fog index 1, etc..

13.1.3 Fog Application

Once the fog indices are calculated they are applied to interpolate the fog color and the current color, the controlling equations depending on whether the colors are represented in RGBA ro CI mode. The mode selection is made with the *ColorMode* bit in **FogMode**.

13.1.3.1 RGBA Fogging Equation

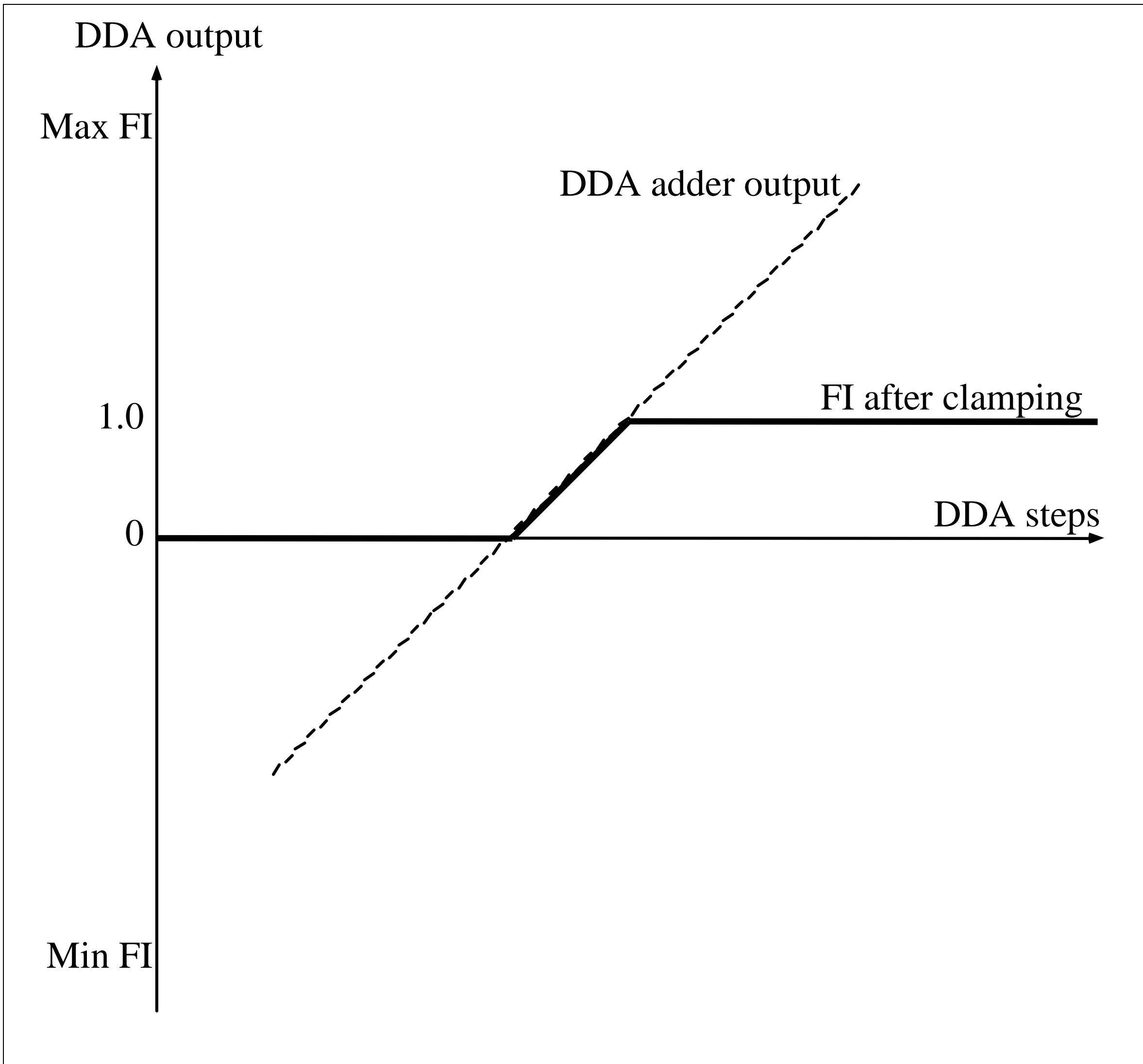
Fogging is applied differently depending on the color mode. For RGBA mode the fogging equation is:

where:

- V = outgoing color
- FC = fog color
- C = incoming fragment color
- FI = fog index

The equation is applied to the color components red, green and blue; alpha is not modified.

The diagram below shows how the fogging would typically affect fragments. Initially no fogging occurs, $f \geq 1.0$, then a region of linear combination of the fragment color and fog color occurs $0.0 < f < 1.0$, followed by a region of constant fog color, $f \leq 0.0$.



13.1.3.2 CI Fogging Equation

In CI mode the equation is:

Note: *The CI value is held only in the red channel for later use, but doing the same equation on all color channels keeps the control simpler. Clamping is needed as the result can overflow the 8 bit color component range.*

13.1.4 FogMode register

The **FogMode** register is used to enable and disable fogging (qualified by the fog application bit in the **Render** command register).

FogMode
FogModeAnd
FogModeOr

Name	Type	Offset	Format
FogMode	Fog	0x8690	Bitfield
FogModeAnd	Fog	0xAC10	Bitfield Logic Mask
FogModeOr	Fog	0xAC18	Bitfield Logic Mask
	Control registers		

Bits	Name	Read	Write	Reset	Description
0	Enable	✔	✔	x	This bit, when set, and qualified by the FogEnable bit in the *Render* command causes the current fragment color to be modified by the fog coefficient and background color.
1	ColorMode	✔	✔	x	This bit selects the color mode. The two options are: 0 = RGB. The RGB fog equation is used. 1 = CI. The Color Index fog equation is used.
2	Table	✔	✔	x	This bit, when set, causes the Fog Index to be mapped via the FogTable before it controls the blending between the fragment's color and the fog color, otherwise the DDA value is used directly.
3	UseZ	✔	✔	x	This bit, when set, causes the DDA to be loaded with the Z DDA values instead of the Fog DDA values. It also adjusts the clamping of the DDA output.
4…8	ZShift	✔	✔	x	This field specifies the amount the (z from DDA + zBias) is right shifted by before it is clamped against 255 and the bottom 8 bits used as the fog index. This should also take into account the number of depth bits there are.
9	InvertFI	✔	✔	x	This bit, when set, inverts the fog index before it is used to interpolates between the fragment's color and the fog color. This is usually 0 when fog values are used and 1 for Z values. Fog values are set up so they decrease with increasing depth and obviously Z values increase with increasing depth.
10…31	Unused	0	0	X	

Figure 13-2 FogMode Register

In addition to the *ColorMode, Table* and *UseZ* bits, FogMode allows inversion of the fog index before interpolation using *InvertFI*.

13.1.5 Fog Example

A Gouraud shaded, fogged RGBA trapezoid, with the fog color set to white:

```
// Enable the color DDA unit in Gouraud shading
// mode
colorDDAMode.UnitEnable = GLINT R4_ENABLE
colorDDAMode.Shade = GLINT R4_GOURAUD_SHADE_MODE
ColorDDAMode(colorDDAMode)
// Enable the Fog unit
fogMode.FogEnable = GLINT R4_TRUE
fogMode.ColorMode = GLINT R4_RGBA_MODE
FogMode(fogMode)
// Set the fog color to white
FogColor(0xFFFFFFFF)
// Load the color start values and deltas for
// dominant edge and the body of the trapezoid

Rstart()    // Set-up the red component start value
dRdX()      // Set-up the red component increments
dRdYDom()
Gstart()    // Set-up the green component start value
dGdX()      // Set-up the green component increments
dGdYDom()
Bstart()    // Set-up the blue component start value
dBdX()      // Set-up the blue component increments
dBYDom()

// Load the start value and delta for dominant edge
// and the body of the trapezoid
// Note that the fog deltas are calculated in the
// same way as the color deltas

FStart()    // Set-up the fog component start value
dFdX()      // Set-up the fog component increments
dFdYDom()

// When issuing a Render command the FogEnable bit
// should be set in addition to the fog unit being
// enabled:
// render.FogEnable = GLINT R4_TRUE
```

13.2 Alpha Test Unit

The alpha test compares a fragment's alpha value with a reference value. Alpha testing is not available in color index (CI) mode.

13.2.1 Alpha Test

The alpha test conditionally rejects a fragment based on the comparison between a reference alpha value and one associated with the fragment, the available tests are:

Mode	Comparison Function	Mode	Comparison Function
0	Never	4	Greater
1	Less	5	Not Equal
2	Equal	6	Greater Than or Equal
3	Less Than or Equal	7	Always

Table 13.1 Alpha Test Comparison Tests

The sense of the test is such that if the comparison mode is set to Less and the reference value is set to 0x80, then fragments with alpha values between 0x0 and 0x7F will pass the test and fragments with alpha values between 0x80 and 0xFF will fail the test and be rejected.

13.2.2 Registers

The **AlphaTestMode** register controls the alpha test:

Name	**Type**	**Offset**	**Format**
AlphaTestMode	AlphaBlend	0x 8800	Bitfield
AlphaTestModeAnd	AlphaBlend	0x ABF0	Bitfield Logic Mask
AlphaTestModeOr	AlphaBlend	0x ABF8	Bitfield Logic Mask
	Control registers		

Bits	Name	Read	Write	Reset	Description
0	Enable	✔	✔	X	When set causes the fragment's alpha value to be tested under control of the remaining bits in this register. If the alpha test fails then the fragment is discarded. When this bit is clear the fragment alway passes the alpha test. 0 = Disable 1 = Enable
1…3	Compare	✔	✔	X	This field defines the unsigned comparison function to use. The options are: 0 = Never 1 = Less 2 = Equal 3 = Less Equal 4 = Greater 5 = Not Equal 6 = Greater Equal 7 = Always The comparison order is as follows: Result = fragment, Alpha Compare Function, reference, Alpha.
4…11	Reference	✔	✔	X	This field holds the 8 bit reference alpha value used in the comparison.

12…31	Unused	0	0	X	

Figure 13-3 AlphaTestMode Register

13.2.3 Alpha Test Example

Set the alpha test mode to be LESS and the reference value to be 0x80:

// Enable unit and set modes

alphaMode.UnitEnable = **GLINT R4**_ENABLE

alphaMode.Compare = **GLINT R4**_ALPHA_COMPARE_MODE_LESS

alphaMode.Reference = 0x80

AlphaMode(alphaMode) // Load register

// Render primitives

14

Framebuffer Read/Write

Before rendering can take place GLINT R4 must be configured to perform the correct framebuffer read and write operations. Framebuffer read and write modes affect the operation of alpha blending, logic ops, writemasks, image upload/download operations and the updating of pixels in the framebuffer.

The framebuffer read and write units are set up in different ways depending on whether Span Operations are being used. Normally, span operations are used for 2D rendering in order to maximize memory bandwidth. Span operations allow multiple pixels to be read and processed in parallel. The following sections discuss the use of the framebuffer read and write units for both standard operation and span operations.

14.1.1 Standard Framebuffer Read Operation

The **FBSourceReadMode** and **FBDestReadMode** registers allows GLINT R4 to be configured to make 0, 1 or 2 reads of the framebuffer. The following are the most common modes of access to the framebuffer:

- Rendering operations with no logical operations, software writemasking or alpha blending. In this case no read of the framebuffer is required and framebuffer writes should be enabled.
- Rendering operations which use logical ops, software writemasks or alpha blending. In these cases the destination pixel must be read from the framebuffer and framebuffer writes must be enabled. (Here set-up varies depending what functionality is required. If alpha blending, logic ops or software writemasks are used the framebuffer is read twice i.e. both the source and the destination. When alpha blending and logic ops are not needed, and hardware writemasks are used (or when the software writemask allows updating of all bits in a pixel) only one read is required.)
- Image upload. This requires reading of the destination framebuffer pixels to be enabled and framebuffer writes to be disabled.
- Image download. This case requires no framebuffer reads (as long as software writemasking, alpha blending and logic ops are disabled) but writes must be enabled.

The data read from the framebuffer may be tagged either FBDefault (data which may be written back into the framebuffer or used in some manner to modify the fragment color) or FBColor (data which will be uploaded to the host). Table 14.2 Framebuffer Read/Write Modes summarizes the framebuffer read/write control for common rendering operations:

ReadSource	Read Destination	Writes	Read Data Type	Rendering Operation
Disabled	Disabled	Enabled	-	Rendering with no logical operations, software writemasks or blending.
Disabled	Disabled	Enabled	-	Image download.
Disabled	Enabled	Disabled	FBColor	Image upload.
Enabled	Disabled	Enabled	FBDefault	Image copy with hardware writemasks and no alpha blending orlogical operations

Disabled	Enabled	Enabled	FBDefault	Rendering using logical operations, software writemasks or blending.
Enabled	Enabled	Enabled	FBDefault	Image copy with software writemasks, alpha blending or logic ops.

Table 14.2 Framebuffer Read/Write Modes

14.1.2 Framebuffer Read Span Operations

As well as performing standard, single pixel at a time, read operations the framebuffer read unit can be used to process span operations. The simplest type of operation is where a span mask is presented to the read unit and the ReadSource bit is enabled. This will cause the unit to read a complete span of pixels from the framebuffer in a 64-bit packed format. The data is always read as a set of 64 bit words. This allows maximum use of both memory and core bandwidth since multiple pixels are being processed.

Since a span mask may not necessarily have all its bits set to 1 (i.e. only a subset of pixels in the span need to be processed), it would be wasteful of memory bandwidth to always read the complete span. For example, at the right hand edge of a rectangle which is being copied, we want the read unit to only read up to the rightmost pixel but not beyond. Whether a 64 bit word is read depends on the corresponding bit values in the span mask. Since each bit in the mask represents a pixel, either 1, 2 or 4 bits will represent a 32 bit word for the depths 32, 16 and 8 bits respectively. If the group of bits representing a 32 bit word is non-zero then the corresponding 32 bits will be read from the framebuffer. Thus:

- at 32 bits per pixel, a single bit in the span mask corresponds to 32 bits in the framebuffer and 32 bit words will be read only at those locations where the corresponding bit in the span mask is a 1.
- at 16 bits per pixel, 2 bits in the span mask represent 32 bits in the framebuffer. A 32 bit word will be read only at those locations where the corresponding 2 span bits form a non-zero value.
- at 8 bits per pixel, a 32 bit word will be read only at those locations where the corresponding 4 span bits form a non-zero value.

The number of 32bit words read from the framebuffer is thus a function of the span mask and the number of bits per pixel, though this is not normally of interest to the programmer. However, the number of 32bit words becomes important for span operations where the data is downloaded from the host. For example, an image download operation using a span operation only requires those 32 bit words which contain required pixel data to be downloaded. Some examples of this are given later.

14.1.3 Merge-copy Span Operations

To understand the way in which the read units works we will examine the way in which a span operation with a logic op works. In particular we consider the case where both ReadSource and ReadDestination bits are set in the **FBReadMode** register. For example, this would be the case when copying data within the framebuffer with an xor logic op.

To perform this operation, the framebuffer read unit must read both a source span of data and a destination span of data. These spans must then be merged so that the data presented to the logic op unit consists of source and destination pairs. Since the logic op unit can combine up to 32 bits at a time, the data can be presented in the form of packed 32 bit words (at 8 bits per pixel this means that the logic op unit can work on 4 pixels at a time).

It would be wasteful of memory bandwidth to read 32 bits from the source followed by 32 bits from the destination. This would result in too many memory page breaks. So the read unit reads a complete source span and stores it internally as Pattern RAM in the local buffer. Then the destination span is read. As the destination span is read, it is merged with the saved source span data so that the data which the logical op unit sees comprises corresponding sections of source and destination data. The logic op unit can then combine this data and present a series of 32 bit results to the framebuffer write unit.

The Pattern RAM is so named because it can be used for pattern filling operations and was a distinct area of memory in previous Permedia chipsets.

15

Alpha Blending

In this chapter we discuss alpha blending. The alpha blend unit performs opacity calculations on the color and alpha components of pixel fragments according to functions defined in the color mode and alpha mode alpha blend registers:

- Source Blending Functions
- Destination Blending Functions
- Color Component Alpha Blending
- Alpha Component Alpha Blending
- Context Switching
- Registers
- Readback

15.1 Introduction

The alpha value is an opacity gradient, with the value of 0 representing complete transparency and a value of 1 representing complete opacity.

Both source and destination pixels have associated blending functions that perform calculations to set opacity values before blending the two pixel values occurs.

15.1.1 Alpha Blend Functions

Alpha blending functions are performed on both color components and alpha components.

The alpha blend unit performs the following functions:

- Calculates opacity on incoming (source) pixel information
- Calculates opacity on existing framebuffer (destination) pixel information
- Blends the source and destination pixel information into a new pixel value

There are 3 source inputs for both RGB and Alpha: Arg0, Arg1 and Arg2. Arg2 is always the interpolator. The opmodes behave as follows:

GL_REPLACE	Arg0
GL_MODULATE	Arg0 * Arg1
GL_ADD	Arg0 + Arg1
GL_ADD_SIGNED_EXT	Arg0 + Arg1 - 128
GL_INTERPOLATE_EXT	Arg0 * Arg2 + Arg1 * (1 - Arg2)

Each source can come from one of:

GL_PRIMARY_COLOR_EXT	color of incoming fragment
GL_TEXTURE	texel of corresponding stage

GL_CONSTANT_EXT	texture enviroment blend color
GL_PREVIOUS_EXT	result of combine from previous unit (always incoming fragment if stage 0)

In addition the RGB channels can specify the alpha component (i.e replicate the alpha into rgb).

The Blend unit also has an effect on compositing and border textures.

15.1.2 Alpha Blend Registers

The alpha blend registers comprise the following segments:

- Alpha Blend Color Operations
- Alpha Blend Alpha Operations
- Alpha Source Color Assignments
- Alpha Destination Color Assignments
- Chroma Test Operations
- 2D Configuration Operations
- Context Operations

Blending occurs in color mode and alpha mode alpha blend registers, called **AlphaBlendColorMode** and **AlphaBlendAlphaMode**, respectively.

The **AlphaBlendColorMode** register assigns blend functions to color components R, G and B, and the **AlphaBlendAlphaMode** register assigns a blend function to the alpha component, A.

15.2 Source Blending Functions

Source blending function components are defined in the source blend bits of the **AlphaBlendColorMode** and **AlphaBlendAlphaMode** registers. The functions correspond to OpenGL source blending parameters.

15.2.1 OpenGL Alpha Blending

The alpha blend unit combines the fragment's color value to be stored in the framebuffer, using the blend equation:

$$C_o = C_s S + C_d D$$

where: Co is the output color, Cs is the source color (calculated internally) and Cd is the destination color read from the framebuffer.

The source blending function, S, and the destination blending function, D, are defined in the following tables:

Mode	Value	R	G	B	A
0	Zero	0	0	0	0
1	One	1	1	1	1
2	Destination Color	R_d	G_d	B_d	A_d
3	One Minus Destination Color	$1 - R_d$	$1 - G_d$	$1 - B_d$	$1 - A_d$

4	Source Alpha	A_s	A_s	A_s	A_s
5	One Minus Source Alpha*	$1 - A_s$	$1 - A_s$	$1 - A_s$	$1 - A_s$
6	Destination Alpha	A_d	A_d	A_d	A_d
7	One Minus Destination Alpha	$1 - A_d$	$1 - A_d$	$1 - A_d$	$1 - A_d$
8	Source Alpha Saturate	Min of (A_s, $1 - A_d$)	min of (A_s, $1 - A_d$)	min of (A_s, $1 - A_d$)	1

Table 15.3 Source Blending Functions

The terms in the equations are in the form Cxy, where x denotes source component (s) or destination component (d), and y denotes color component r, g, b, or a, for Red, Green, Blue, or Alpha, respectively.

Note: *Values are defined as floating point numbers. All source color component values are in the range 0 to 1.0 inclusive as defined in, e.g. OGL texture environment color parameters (GL_TEXTURE_ENV_COLOR).*

Mode	Value	R	G	B	A
0	Zero	0	0	0	0
1	One	1	1	1	1
2	Source Color	R_s	G_s	B_s	A_s
3	One Minus Source Color	$1 - R_s$	$1 - G_s$	$1 - B_s$	$1 - A_s$
4	Source Alpha	A_s	A_s	A_s	A_s
5	One Minus Source Alpha	$1 - A_s$	$1 - A_s$	$1 - A_s$	$1 - A_s$
6	Destination Alpha	A_d	A_d	A_d	A_d
7	One Minus Destination Alpha	$1 - A_d$	$1 - A_d$	$1 - A_d$	$1 - A_d$

Table 8.4 Destination Blending Functions

15.3 Destination Blending Functions

Destination blending function components are defined in the *DestBlend* bits of the **AlphaBlendColorMode** register and the **AlphaBlendAlphaMode** registers. If the blend operations require any destination color components then the framebuffer read mode must be set appropriately.

15.3.1 OpenGL Destination Blending

The destination blending corresponds to OpenGL source blending parameters.

In some situations blending is desired when no retained alpha buffer is present. In this case the alpha value which is considered to be read from the framebuffer is set to 1.0. The *NoAlphaBuffer* bit in the **AlphaBlendAlphaMode** register controls this.

The terms in the blend equations are in the form Cxy, where x denotes source component (s) or destination component (d), and y denotes color component r, g, b, or a, for Red, Green, Blue, or Alpha, respectively.

* One Minus Value is sometimes referred to as Inverse Value.

Blend values are defined as floating point numbers. All source color component values should be clamped in the range 0 to 1.0 inclusive.

In addition to glBlendFunc, GLINT R4 supports OGL texture functions described in GL_TEXTURE_ENV_MODE during texture compositing and application. Support for GL_texture_ env_combine_EXT is enabled by calling TexEnv with GL_TEXTURE_ ENV_MODE set to GL_COMBINE_EXT. This allows user to explicitly set up the fragment operations for the RGB and Alpha channels - in particular, GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB and GL_RGBA. This allows full texture function implementation in both Texture0 and Texture1. The equations for each case are as described in *The OpenGL Reference Manual* and *The OpenGL Programming Guide* from Addison-Wesley.

15.3.1.1 Embossed bump-mapping

Special ENV_MODE support is available in the GLINT R4 when this extension is used for embossed bump-mapping. Normally GL_PREVIOUS_EXT maps onto GL_PRIMARY_ COLOR_EXT for stage0. However when the EnableBumpHeightAsSource flag is true, GL_PREVIOUS_EXT uses the difference between texture stage1 and stage0 alpha channels for the source input for stage0:

clamp(tex0.alpha - tex1.alpha + 128

This difference when replicated into the rgb channels can be used to modulate the other input to make it lighter or darker. The alpha channel is the same in each stage, but is read offset in the second stage relative to the first stage.

15.3.2 QuickDraw 3D Alpha Blending

When the AlphaType bit in the AlphaBlendAlphaMode register is set then QuickDraw 3D style alpha blend equations are followed. The OpenGL equations above are used for the RGB components, but the alpha channel is treated differently and has a single source and destination blend functions as follows:

$C_a = 1 - (1 - C_{sa}) * (1 - C_{da})$

The source and destination blend functions should be set as follows:

Name	Source Blend	Destination Blend
Pre-multiplied	ONE	ONE_MINUS_SRC_ALPHA
Interpolated	SRC_ALPHA	ONE_MINUS_SRC_ALPHA

Table 15.5 Source Blending Functions

The alpha calculation is the same for both modes.

15.3.3 Image Formatting

The alpha blend and color formatting units can be used to format image data into any of the supported GLINT R4 framebuffer formats, though conversion between CI and RGB modes or vice versa are not supported.

Consider the case where the framebuffer is in RGBA 4:4:4:4 mode, and an area of the screen is to be uploaded and stored in an 8 bit RGB 3:3:2 format. The sequence of operations is:

- Set the rasterizer as appropriate (described in Volume II, *Rasterizer*)
- Enable framebuffer reads

- Disable framebuffer writes and set the UpLoadData bit in the **FBWriteMode** register
- Enable the alpha blend unit with a blend function which passes the destination value and ignores the source value (source blend Zero, destination blend One) and set the color mode to RGBA 4:4:4:4
- Set the color formatting unit to format the color of incoming fragments to an 8 bit RGB 3:3:2 framebuffer format.

The upload now proceeds as normal.

The same technique can be used to download data which is in any supported framebuffer format, in this case the rasterizer is set to sync with FBData, rather than Color. In this case framebuffer writes are enabled, and the UpLoadData bit cleared.

15.3.4 Registers

The unit is controlled by the AlphaBlendAlphaMode and AlphaBlendColorMode registers:

AlphaBlendAlphaMode
AlphaBlendAlphaModeAnd
AlphaBlendAlphaModeOr

Name	**Type**	**Offset**	**Format**
AlphaBlendAlphaMode	Alpha Blend	0x AFA8	Bitfield
AlphaBlendAlphaModeAnd	Alpha Blend	0x AD30	Bitfield Logic Mask
AlphaBlendAlphaModeOr	Alpha Blend	0x AD38	Bitfield Logic Mask
	Control registers		

Bits	Name	Read[1]	Write	Reset	Description
0	Enable	✓	✓	x	When set causes the fragment's alpha to be alpha blended under control of the remaining bits in this register. When clear the fragment alpha remains unchanged (but may later to affected by the chroma test).
1…4	SourceBlend	✓	✓	x	This field defines the source blend function to use. See the table below for the possible options.
5…7	DestBlend	✓	✓	x	This field defines the destination blend function to use. See the earlier table for the possible options.
8	Source TimesTwo	✓	✓	x	This bit, when set causes the source blend result to be multiplied by two before it is combined with the dest blend result. When this bit is clear no multiply occurs.
9	DestTimes Two	✓	✓	x	This bit, when set causes the dest blend result to be multiplied by two before it is combined with the source blend result. When this bit is clear no multiply occurs.
10	Invert Source	✓	✓	x	This bit, when set, causes the incomming source data to be inverted before any blend operation takes place.
11	Invert Dest	✓	✓	x	This bit, when set, causes the incomming dest data to be inverted before any blend operation takes place.
12	NoAlpha Buffer	✓	✓	x	When this bit is set the source alpha value is always set to 1.0. This is typically used when no retained alpha buffer is present but alsos overrides any retained alpha value if one is present. Color formats with no alpha field defined automatically have their alpha value set to 1.0 regardless of the state of this bit.
13	Alpha Type	✓	✓	x	This bit selects which set of equations are to be used for the alpha channel. 0 = OpenGL 1 = Apple

[1] Logic Op register readback is via the main register.

14	Alpha Conversion	✓	✓	x	This bit selects how alpha component less than 8 bits wide are converted to 8 bit wide values prior to the alpha blend calculations. The options are 0 = Scale 1 = Shift
15	Constant Source	✓	✓	x	This bit, when set, forces the Source color to come from the AlphaSourceColor register (in 8888 format) instead of the framebuffer. 0 = Use framebuffer alpha 1 = Use AlphaSourceColor register alpha value.
16	Constant Dest	✓	✓	x	This bit, when set, forces the destination color to come from the AlphaDestColor register (in 8888 format) instead of the fragment's color. 0 = Use fragment's alpha. 1 = Use AlphaDestColor register alpha value
17…19	Operation	✓	✓	x	This field selects how the source and destination blend results are to be combined. The options are: 0 = Add 1 = Subtract (i.e. S - D) 2 = Subtract reversed (i.e. D - S) 3 = Minimum 4 = Maximum

Notes The Alpha Conversion bit selects the conversion method for alpha values read from the framebuffer.

- The Scale method linearly scales the alpha values to fill the full range of an 8 bit value. This method is preferable when, for example, downloading an image with fewer bits per pixel into a deeper (i.e. more bits per pixel) framebuffer.
- The Shift method just left shifts by the appropriate amount to make the component 8 bits wide. This method is preferable when blending into a dithered framebuffer as it preserves the framebuffer alpha when fragment alpha does not contribute to it.

Alpha is controlled separately from color to allow, for example, the situation in antialiasing where it represents coverage - this must be linearly scaled to preserve the 100% covered state.

The logic operator equivalents behave the same way but the new mode is AND'd or OR'd with the former mode before replacing it.

The table below shows the different color modes supported. In the R, G, B and A columns the nomenclature n@m means this component is n bits wide and starts at bit position m in the framebuffer. The least significant bit position is 0 and a dash in a column indicates that this component does not exist for this mode.

In the case of the RGB formats where no Alpha is shown then the alpha field is set to 255. In this case the *NoAlphaBuffer* bit in the **AlphaBlendAlphaMode** register should be set which causes the alpha component to be set to 255.

Two color ordering formats are supported, namely ABGR and ARGB, with the right most letter representing the color in the least significant part of the word. This is controlled by the Color Order bit in the *AlphaBlendColorMode* register, and is easily implemented by just swapping the R and B components after conversion into the internal format. The only exception to this are the 3:3:2 formats where the actual bit fields extracted from the framebuffer data need to be modified as well because the R and B components are differing widths. CI processing is not effected by this swap and the result is always on internal R channel.

The format to use is held in the *AlphaBlendColorMode* register. Note that in OpenGL the alpha blending is not defined for CI mode..

When converting a Color Index value to the internal format any unused bits are set to zero

Figure 15-1 AlphaBlendAlphaMode Register

The ColorConversion bit selects the conversion method for RGB values read from the framebuffer.

The Scale method linearly scales the color values to fill the full range of an 8 bit value. This method is preferable when, for example, downloading an image with fewer bits per pixel into a deeper (i.e. more bits per pixel) framebuffer.

The Shift method just left shifts by the appropriate amount to make the component 8 bits wide. This method is preferable when blending into a dithered framebuffer as it preserves the framebuffer color when fragment color does not contribute to it. The scale method would otherwise cause the 'fraction' bits to be non zero, which may result in a different color when re-dithered again. This shows up as a faint outline of the underlying polygon, when, for example, an alpha blended texture is used with zero value to provide cut-outs.

The *AlphaConversion* bit selects the conversion method for the Alpha values in a similar way. It is controlled separately to allow, for example, the situation in antialiasing where it represents coverage - this must be linearly scaled to preserve the 100% covered state.

The alpha blend can be augmented by a chroma test, discussed next.

15.3.5 Chroma Testing

Chroma test involves testing a fragment's color against a range of colors. The fragment can then be rejected based on the outcome. The framebuffer source color, framebuffer destination color and the fragment's color before or after alpha blending can all be used for the test.

The source and destination keying are needed by DirectX for its chroma key blts.

Rejecting a fragment based on its color can be used to prevent writes where the destination color does not change. For example a fogged fragment which has the same color as the background fog color does not need to be written if the screen was cleared to the fog color.

The chroma test is given by:

where Cl is the lower chroma value held in the **ChromaLower** register, Cu is the upper chroma value held in the **ChromaUpper** register and T is the selected color to test against. Each component is tested separately and obviously a component can be excluded from the test by setting the lower and upper values to 0 and 255 respectively.

The format of the **ChromaLower** and **ChromaUpper** registers is the red byte is in the least significant byte, then the green byte and finally the blue byte. If the framebuffer format for a color component is less than 8 bits then the unused bits in the upper and lower register for this component are set to zero.

The chroma test is enabled when the *Enable* bit in the **ChromaMode** register is set. The source color to test is given by the *Source* field. The sense of the chroma test is controlled by the *Sense* bit - the effect shown in the table below:

Chroma Test Enabled	Test Result	ChromaSense	Action
N	X	X	The framebuffer is updated as normal
Y	False	Include	The framebuffer is not updated
Y	True	Include	The framebuffer is updated as normal
Y	False	Exclude	The framebuffer is updated as normal
Y	True	Exclude	The framebuffer is not updated

The format of the **ChromaTestMode** register is:

ChromaTestMode
ChromaTestModeAnd
ChromaTestModeOr

Name	**Type**	**Offset**	**Format**
ChromaTestMode	Alpha Blend	0x8F18	Bitfield
ChromaTestModeAnd	Alpha Blend	0xACC0	Bitfield Logic Mask
ChromaTestModeOr	Alpha Blend	0xACC8	Bitfield Logic Mask
	Control registers		

Bits	Name	Read[2]	Write	Reset	Description
0	Enable	✓	✓	x	When set enables chroma testing under control of the remaining bits in this register. When clear no chroma test is done.
1…2	Source	✓	✓	x	This field selects which color (after any suitable conversion) is to be used for the chroma test. The values are: 0 = FBSourceData 1 = FBData 2 = Input Color (from fragment) 3 = Output Color (after any alpha blending)
3…4	PassAction	✓	✓	x	This field defines what action is to be taken if the chroma test passes (and is enabled). The options are: 0 = Pass 1 = Reject 2 = Substitute ChromaPassColor 3 = Substitute ChromaFailColor
5…6	FailAction	✓	✓	x	This field defines what action is to be taken if the chroma test fails (and is enabled). The options are: 0 = Pass 1 = Reject 2 = Substitute ChromaPassColor 3 = Substitute ChromaFailColor
7…31	Unused	0	0	x	

Notes: Used to test the fragment's color against a range of colors after alphablending. The chroma test is enabled by the enable bit (0) in the register. Note: incompatible with MX programming.

The logic operator equivalents behave the same way but the new mode is AND'd or OR'd with the former mode before replacing it.

The color format and order is needed as the destination color is read from the framebuffer and needs to be converted into the internal GLINT R4 representation, it should therefore be set as appropriate for the framebuffer.

[2] Logic Op register readback is via the main register only

	Format	Name	Internal Color Channel R	G	B	A
Color Order: BGR	0	8:8:8:8	8@0	8@8	8@16	8@24
	1	5:5:5:5	5@0	5@5	5@10	5@15
	2	4:4:4:4	4@0	4@4	4@8	4@12
	3	4:4:4:4Front	4@0	4@8	4@16	4@24
	4	4:4:4:4Back	4@4	4@12	4@20	4@28
	5	3:3:2Front	3@0	3@3	2@6	255
	6	3:3:2Back	3@8	3@11	2@14	255
	7	1:2:1Front	1@0	2@1	1@3	255
	8	1:2:1Back	1@4	2@5	1@7	255
	13	5:5:5Back	5@16	5@21	5@26	255
Color Order: RGB	0	8:8:8:8	8@16	8@8	8@0	8@24
	1	5:5:5:5	5@10	5@5	5@0	5@15
	2	4:4:4:4	4@8	4@4	4@0	4@12
	3	4:4:4:4Front	4@16	4@8	4@0	4@24
	4	4:4:4:4Back	4@20	4@12	4@4	4@28
	5	3:3:2Front	3@5	3@2	2@0	255
	6	3:3:2Back	3@13	3@10	2@8	255
	8	1:2:1Back	1@7	2@5	1@4	255
	7	1:2:1Front	1@3	2@1	1@0	255
	13	5:5:5Back	5@26	5@21	5@16	255
CI	14	CI8	8@0	0	0	0
	15	CI4	4@0	0	0	0

Table 15.6 GLINT R4 Color Modes

The framebuffer may be configured to be RGBA or Color Index (CI). The R, G, B and A columns show the width of each color component. n@m means that n bits starting at bit position m are read and scaled to fit the 8bit internal color channel format. The least significant bit position is zero. A numerical value (0 or 255) indicates the value substituted when the corresponding channel does not exist in the framebuffer.

For the Front and Back Modes the value to be blended is read only from the low bits or high bits respectively. This is to assist with color space double buffering.

15.3.6 Alpha Blend Example

This example sets the blend mode to allow antialiasing of polygons, i.e. source blend function = Source Alpha Saturate, destination blend function = One. These blend functions are suitable for polygon antialiasing when polygons are drawn in front to back order, and the depth test is disabled.

```
// Enable framebuffer reads allow blend operation
// - Not Shown -


// Set the alpha mode.
```

```
alphaBlendColorMode.Enable = GLINT R4_ENABLE
alphaBlendColorMode.SourceBlend = GLINT R4_BLEND_SRC_ALPHA_SATURATE
alphaBlendColorMode.DestinationBlend = GLINT R4_BLEND_ONE
alphaBlendColorMode.ColorFormat = as appropriate

AlphaBlendColorMode(alphaBlendColorMode)        // Load register

// Enable antialias application and disable
// depth testing
// - Not Shown -

// Render polygons sorted front to back with
// Coverage Enable bit set in the Render command
// - Not Shown –
```

16

Color Format and Logical Ops

The color format unit converts from GLINT R4's internal color representation to a format suitable to be written into the framebuffer. This process may optionally include dithering of the color values for framebuffers with less than 8 bits width per color component. If the unit is disabled then the color is not modified in any way.

16.1 Color and Alpha Formats

GLINT R4 separates the Alpha and Color format information into two new registers (**AlphaBlendColorMode** and **AlphaBlendAlphaMode**). The **AlphaBlendMode** register is not supported.

The color format is held in the **AlphaBlendColorMode** register. Note that in OpenGL alpha blending is not defined for CI mode. Raw framebuffer formats from local memory are only converted to 8-bit formats in the AlphaBlend registers.

Alpha is controlled separately from color to allow, for example, the situation in antialiasing where it represents coverage - this must be linearly scaled to preserve the 100% covered state.

The table below shows the different color modes supported. In the R, G, B and A columns the nomenclature n@m means the component is n bits wide and starts at bit position m in the framebuffer. The least significant bit position is 0 and a dash in a column indicates that this component does not exist for this mode.

In the case of RGB formats where no Alpha is shown, the alpha field should be set to 255. Use the *NoAlphaBuffer* bit in the **AlphaBlendAlphaMode** register to do this.

GLINT R4 supports two color-ordering formats: ABGR and ARGB. The rightmost letter represents the color in the least significant part of the word. This is controlled by the *ColorOrder* bit in the **AlphaBlendColorMode** register (and elsewhere), and is easily implemented by just swapping the R and B components after conversion into the internal format. The only exception to this are the 3:3:2 formats where the actual bit fields extracted from the framebuffer data need to be modified as well because the R and B components are differing widths. CI processing is not affected by this swap and the result is always on the internal R channel.

When converting a Color Index value to the internal format any unused bits are set to zero

				Internal Color Channels			
	Format	**Color Order**	**Name**	**R**	**G**	**B**	**A**
	0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
	1	BGR	4:4:4:4	4@0	4@4	4@8	4@12
C	2	BGR	5:5:5:1	5@0	5@5	5@10	1@15
O	3	BGR	5:6:5	5@0	6@5	5@11	-
L	4	BGR	3:3:2	3@0	3@3	2@6	-
o	0	RGB	8:8:8:8	8@16	8@8	8@0	8@24

u	1	RGB	4:4:4:4	4@8	4@4	4@0	4@12
r	2	RGB	5:5:5:1	5@10	5@5	5@0	1@15
	3	RGB	5:6:5	5@11	6@5	5@0	-
	4	RGB	3:3:2	3@5	3@2	2@0	-
CI	15	X	CI8	8@0	0	0	0

The *AlphaConversion* bit in the **AlphaBlendAlphaMode** register selects the conversion method for alpha values read from the framebuffer. When the conversion bit is set the corresponding component(s) is left shifted by (8 - n) bits and zero filling.

Note *For some formats the components have different widths, hence different values of n.*

- The Scale method linearly scales the alpha values to fill the full range of an 8 bit value. This method is preferable when, for example, downloading an image with fewer bits per pixel into a deeper (i.e. more bits per pixel) framebuffer.
- The Shift method left shifts by the appropriate amount to make the component 8 bits wide. This method is preferable when blending into a dithered framebuffer as it preserves the framebuffer alpha when fragment alpha does not contribute to it. For example if a three bit component has bits B2, B1 and B0 then the 8 bit value would be made up as follows:

Bit 7 Bit 0 of output byte

B2 B1 B0 B2 B_1 B_0 B_2 B_1

If the alpha component doesn't exist in the format, or *NoAlphaBuffer* is set then the alpha value is not affected by the setting of the *AlphaConversion* bit and is always set to 255 (in the 8 bit domain) or 256 (in the 9 bit domain).

The **AlphaBlendColorMode** register controls color channel blending. It has the following format:

Name	Type	Offset	Format
AlphaBlendColorMode	Alpha Blend	0x AFA0	Bitfield
AlphaBlendColorModeAnd	Alpha Blend	0x ACB0	Bitfield Logic Mask
AlphaBlendColorModeOr	Alpha Blend	0x ACB8	Bitfield Logic Mask

Control registers

Bits	Name	Read[3]	Write	Reset	Description
0	Enable	✔	✔	x	When set causes the fragment's color to be alpha blended under control of the remaining bits in this register. When clear the fragment color remains unchanged (but may later to effected by the chroma test).

[3] Logic Op register readback is via the main register

1…4	SourceBlend	✔	✔	x	This field defines the source blend function to use. See the table in the *AlphaBlendColorMode* register for the possible options
5…7	DestBlend	✔	✔	x	This field defines the destination blend function to use. See the table in the *AlphaBlendColorMode* register for the possible options
8	Source TimesTwo	✔	✔	x	This bit, when set causes the source blend result to be multiplied by two before it is combined with the dest blend result. When this bit is clear no multiply occurs
9	DestTimes Two	✔	✔	x	This bit, when set causes the dest blend result to be multiplied by two before it is combined with the source blend result. When this bit is clear no multiply occurs
10	InvertSource	✔	✔	x	This bit, when set, causes the incomming source data to be inverted before any blend operation takes place
11	InvertDest	✔	✔	x	This bit, when set, causes the incomming dest data to be inverted before any blend operation takes place
12…15	Color Format	✔	✔	x	This field defines framebuffer color formats. See the table in the *AlphaBlendColorMode* register for the possible options
16	ColorOrder	✔	✔	x	This bit selects the color order in the framebuffer: 0 = BGR 1 = RGB
17	Color Conversion	✔	✔	x	This bit selects how color components less than 8 bits wide are converted to 8 bit wide values prior to the alpha blend calculations. The options are 0 = Scale 1 = Shift
18	Constant Source	✔	✔	x	This bit, when set, forces the Source color to come from the *AlphaSourceColor* register (in 8888 format) instead of the framebuffer. 0 = Use framebuffer 1 = Use AlphaSourceColor register
19	ConstantDest	✔	✔	x	This bit, when set, forces the destination color to come from the *AlphaDestColor* register (in 8888 format) instead of the fragment's color. 0 = Use fragment's color. 1 = Use *AlphaDestColor* register.
20…23	Operation	✔	✔	x	This field selects how the source and destination blend results are to be combined. The options are: 0 Add 1 Subtract (i.e. S - D) 2 Subtract reversed (i.e. D - S) 3 Minimum 4 Maximum
24	SwapSD	✔	✔	x	This bit, when set causes the source and destination pixel values to be swapped. The main use for this is to allow a downloaded color value to be in a format other than 8888 and use this unit to do color conversion.

The *ColorConversion* bit selects the conversion method for RGB values read from the framebuffer, similarly to the *AlphaConversion* bit for alpha values:

- The Scale method linearly scales the color values to fill the full range of an 8 bit value. This method is preferable when, for example, downloading an image with fewer bits per pixel into a deeper (i.e. more bits per pixel) framebuffer.
- The Shift method left shifts by the appropriate amount to make the component 8 bits wide. This method is preferable when blending into a dithered framebuffer as it preserves the framebuffer color when fragment color does not contribute to it[4]

16.1.1 Color Dithering

GLINT R4 uses an ordered dither algorithm to implement color dithering. The following table shows the exact type of dithering used when dither is enabled. The type of dithering depends on the width of individual color components:

Component Width	Type of Dithering
8	No Dithering
5	2x2 Ordered Dither
4	4x4 Ordered Dither
3	4x4 Ordered Dither
2	4x4 Ordered Dither
1	4x4 Ordered Dither

Table 16.7 Dither Methods

GLINT R4's ordered dither matrices are shown below:

0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

0	2
3	1

Table 16.8 Ordered Dither Matrices, 4x4 and 2x2.

If the color formatting unit is disabled, the RGBA color components are not modified. Instead, they are truncated or rounded under the control of the *RoundingMode* bit in the **DitherMode** register when they are placed in the framebuffer. This assumes that the framebuffer width is less than 8 bits per component. In CI mode the value is rounded to the nearest integer. In both cases the result is clamped to a maximum value to prevent overflow.

In some situations only screen coordinates are available, but windows-relative dithering is required. This can be implemented by adding an optional offset to the coordinates before indexing the dither tables. The offset is a two bit number which is supplied for each coordinate, X and Y. The *XOffset*, *YOffset* fields in the **DitherMode** register control this operation, if window relative coordinates are used they should be set to zero. For more information on offset calculation see section 4.2.10.1 - Address Calculation, in Volume I

[4]The scale method would otherwise cause the 'fraction' bits to be non zero, which could result in a different color when re-dithered again. This shows up as a faint outline of the underlying polygon, when, for example, an alpha blended texture is used with zero value to provide cut-outs.

Alpha channel dithering is qualified by the *AlphaDither* control bit. When cleared the alpha channel is processed in the same way as the color channels, as dictated by the *DitherEnable* bit. When the *AlphaDither* bit is set however, the alpha channel is not dithered, but is processed according to the state of the *RoundingMode* bit. The ability to disable dithering on the alpha channel is useful when using the alpha buffer to hold coverage information during antialiasing. In this situation dithering adds noise to the coverage value, which would create artifacts where a pixel which should be fully covered is reported as not fully covered.

See *The OpenGL Reference Manual* and *The OpenGL Programming Guide* from Addison-Wesley for more details on dithering.

16.1.2 Registers

Dither operations are controlled by the **DitherMode** register:

DitherMode
DitherModeAnd
DitherModeOr

Name	Type	Offset	Format
DitherMode	Global	0x8818	Bitfield
DitherModeAnd	Global	0xACD0	Bitfield Logic Mask
DitherModeOr	Global	0xACD8	Bitfield Logic Mask
	Control Register		

Bits	Name	Read	Write	Reset	Description
0	Enable	✔	✔	x	When set causes the fragment's color values to be dithered or rounded under control of the remaining bits in this register. If this bit is clear then the fragment's color is passed unchanged.
1	Dither Enable	✔	✔	x	When this bit is set any RGB format color is dithered, otherwise it is rounded to the destination size under control of the RoundingMode field. See the table below for the dither matrix and how it is combined with the color components. Color Index formats are always rounded.
2…5	Color Format	✔	✔	x	The color format which in turn is coded from the size and position of the red, green, blue and (if present) the alpha components.
6…7	Xoffset	✔	✔	x	This offset is added to the fragment's x coordinate to derive the x address in the dither table. This allows window-relative dithering using screen coordinates.
8…9	Yoffset	✔	✔	x	This offset is added to the fragment's y coordinate to derive the y address in the dither table. This allows window-relative dithering using screen coordinates.
10	Color Order	✔	✔	x	Holds the color order. The options are: 0 = BGR 1 = RGB

11…13	Reserved	0	0	x	
14	Alpha Dither	✔	✔	x	This bit allows the alpha channel to be rounded even when the color channels are dithered. This helps when antialiasing. 0 = Alpha value is dithered (if DitherEnable is set) 1 = Alpha value is always rounded.
15…16	Rounding Mode	✔	✔	x	0 = Truncate 1 = Round Up 2 = Round Down
17…31	Unused	0	0	x	

Figure 16-1 DitherMode Register

16.1.3 Dither Example

To set the framebuffer format to RGB 3:3:2 and enable dithering:

```
// 332 Dithering
ditherMode.UnitEnable = GLINT R4_TRUE
ditherMode.DitherEnable = GLINT R4_TRUE
ditherMode.ColorMode = GLINT R4_COLOR_FORMAT_RGB_332

DitherMode(ditherMode)      // Load register
```

16.1.4 3:3:2 Color Format Example

To set the framebuffer format to RGB 3:3:2 and disable dithering:

```
// 332 No Dither


ditherMode.UnitEnable = GLINT R4_TRUE
ditherMode.DitherEnable = GLINT R4_FALSE
ditherMode.ColorMode = GLINT R4_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)      // Load register
```

16.1.5 8:8:8:8 Color Format Example

To set the framebuffer to RGBA 8:8:8:8 and not dithered:

```
// 8888 Dithered (No effect as 8 bit components are
// not dithered)


ditherMode.UnitEnable = GLINT R4_TRUE
ditherMode.DitherEnable = GLINT R4_FALSE
ditherMode.ColorMode = GLINT R4_COLOR_FORMAT_RGBA_8888
```

```
DitherMode(ditherMode)     // Load register
```

The same can be achieved by disabling the color formatting unit as 8 bit components are not dithered:

```
// 8888 No dither
ditherMode.UnitEnable = GLINT R4_FALSE

DitherMode(ditherMode)     // Load register
```

16.1.6 Color Index Format Example

To set the framebuffer to 4 bit Color Index and enable dithering:

```
// 4 bit CI with dithering
ditherMode.UnitEnable = GLINT R4_TRUE
ditherMode.DitherEnable = GLINT R4_TRUE
ditherMode.ColorMode = GLINT R4_COLOR_FORMAT_CI_4
DitherMode(ditherMode)     // Load register
```

16.2 Logical Op Unit

The logical op unit performs two functions; logic ops between the fragment color (source color) and a value from the framebuffer (destination color), and, optionally control of a special GLINT R4 mode which allows high performance flat shaded rendering.

16.2.1 High Speed Flat Shaded Rendering

This mode is still supported on the GLINT R4 and is detailed below for completeness but offers no advantage over span processing. The technique uses a color value from the **FBWriteData** register instead of fragment color. It is retained for backwards compatibility only. To use the mode the following constraints must be satisfied:

- Flat shaded aliased primitive
- No dithering required or logical ops
- No stencil, depth or GID testing required
- No alpha blending

If all the conditions are met then load the **FBWriteData** register with the required framebuffer color data and set the *UseConstantFBWriteData* bit in the **LogicalOpMode** register. All unused units should be disabled.

This mode is most useful for 2D applications or for clearing the framebuffer when the memory does not support block writes. Note that the **FBWriteData** register should be considered volatile when context switching.

16.2.2 Logical Operations

The logical operations supported by GLINT R4 are:

Mode	Name	Operation	Mode	Name	Operation
0	Clear	0	8	Nor	~(S | D)
1	And	S & D	9	Equivalent	~(S ^ D)
2	And Reverse	S & ~D	10	Invert	~D
3	Copy	S	11	Or Reverse	S | ~D
4	And Inverted	~S & D	12	Copy Invert	~S
5	Noop	D	13	Or Invert	~S | D
6	Xor	S ^ D	14	Nand	~(S & D)
7	Or	S | D	15	Set	1

Where: S = Source (fragment) color, D = Destination (framebuffer) color

Table 16.9 Logical Operations

For correct operation of this unit in a mode which takes the destination color, GLINT R4 must be configured to allow reads from the framebuffer using the FBDestReadMode register. See section §14 for more details.

GLINT R4 makes no distinction between RGBA and CI modes when performing logical operations.

16.2.3 Registers

The operation of the unit is controlled by the **LogicalOpMode** register:

LogicalOpMode
LogicalOpModeAnd
LogicalOpModeOr

Name	Type	Offset	Format
LogicalOpMode	Logic Ops	0x8828	Bitfield
LogicalOpModeAnd	Logic Ops	0xAEC0	Bitfield Logic Mask
LogicalOpModeOr	Logic Ops	0xAEC8	Bitfield Logic Mask

Control registers

Bits	Name	Read	Write	Reset	Description
0	Enable	✔	✔	x	When set causes the fragment's color to be logial op'ed under control of the remaining bits in this register. When clear the fragment color remains unchanged (but may later to effected by write masking).

1…4	LogicOp	✔	✔	x	This field defines the logical op function to use. The options are: 0 = Clear (0) 1 = And(S & D) 2 = AndReverse (S & ~D) 3 = Copy (S) 4 = AndInvert (~S & D) 5 = Noop (D) 6 = Xor (S ^ D) 7 = Or (S | D) 8 = Nor (~(S | D); 9 = Equiv (~(S ^ D); 10 = Invert (~D) 11 = OrReverse (S | ~D) 12 = CopyInvert (~S) 13 = OrInvert (~S | D) 14 = Nand (~(S & D); 15 = Set (1) where: S is Color or FBSourceData D is FBData
5	UseConstantFBWriteData	✔	✔	x	There is no longer any performance advantage to using this bit but it is retained for backwards compatability.
6	BackgroundEnable	✔	✔	x	This bit, when set, enables a different logical operation to be done for background pixels. If this bit is clear then the same logical operation is applied to foreground and background pixels. Setting this bit when the Enable field is zero has no effect. A background pixel is a pixel whose corresponding bit in the color mask is zero.
7…10	BackgroundLogicalOp	✔	✔	x	This field specifies the logical operation to apply to background pixels, if this has been enabled by the BackgroundEnable field. The options and field values are the same as the LogicalOp field.
11	UseConstantSource	✔	✔	x	This field, when set, causes the source data to be taken from the ForegroundColor register, otherwise it is taken from the fragment, if needed. The color format is in the raw framebuffer format and 8 or 16 bit pixels should have their color replicated to fill the full 32 bits.
12…31	Unused	0	0	x	

16.2.4 XOR Example

To set the logical operation to XOR.

```
// Set framebuffer to allow reads
// Not shown
logicalOpMode.UnitEnable = GLINT R4_ENABLE
logicalOpMode.LogicalOp = GLINT R4_LOGICOP_XOR

LogicalOpMode(logicalOpMode)        // Load register
```

16.2.5 Logical Op and Software Writemask Example

To set the logical operation to COPY, enable the software writemask, and write to the green component in an 8 bit framebuffer configured in 3:3:2 RGB mode:

```
// Set framebuffer to allow reads
// Not shown

ditherMode.UnitEnable = GLINT R4_ENABLE
ditherMode.DitherEnable = GLINT R4_ENABLE
ditherMode.ColorMode = GLINT R4_COLOR_FORMAT_RGB_332
DitherMode(ditherMode)                    // Load register

logicalOpMode.UnitEnable = GLINT R4_ENABLE
logicalOpMode.LogicalOp = GLINT R4_LOGICOP_COPY
LogicalOpMode(logicalOpMode)         // Load register

FBSoftwareWriteMask(0xFFFFFFE3)
```

17

Framebuffer Writemasks

Two types of framebuffer writemasking are supported by GLINT R4; Software and Hardware. Software writemasking requires a read from the framebuffer to combine the fragment color with the framebuffer color before checking the bits in the mask to see which planes are writeable. Hardware writemasking is implemented using SDRAM/SGRAM writemasks and no framebuffer read is required. Refer to section 12.3, Windows Initialisation, for further information on Writemasks and Write initialisation.

17.1.1 Software Writemasks

Software writemasking is controlled by the **FBSoftwareWriteMask** register. The data field has one bit per framebuffer bit which when set, allows the corresponding framebuffer bit to be updated. When reset it disables writing to that bit. Software writemasking is applied to all fragments and is not controlled by an enable/disable bit. However it may effectively be disabled by setting the mask to all 1's. Note that the *ReadDestination* bit must be enabled in the **FBDestReadMode** register when using software writemasks, in which some of the bits are zero.

See the Framebuffer Read/Write section for details of how to enable/disable framebuffer reads.

17.1.2 Hardware Writemasks

Hardware writemasks, if present, are controlled using the FBHardwareWriteMask register. If the framebuffer supports hardware writemasks, and they are to be used, then software writemasking should be disabled (by setting all the bits in the FBSoftwareWriteMask register). This results in fewer framebuffer reads when no logical operations or alpha blending is needed.

If the framebuffer is used in 8 bit packed mode, then an 8 bit hardware writemask must be replicated to all 4 bytes of the FBHardwareWriteMask register. If the framebuffer is in 16 bit packed mode then the 16 bit hardware writemask must be replicated to both halves of the FBHardwareWriteMask register.

See the GLINT R4 Reference Guide for more details of framebuffer hardware writemasks.

17.1.3 Registers

Both **FBHardwareWriteMask** and **FBSoftwareWriteMask** are 32 bit registers in which each bit represents a bit in the framebuffer.

17.1.4 Software Writemask Example

Using software writemasks:

```
// Enable framebuffer reads (not shown)
// Set the writemask
```

```
FBSoftwareWriteMask(0x0F0F0F0F)
```

See §16.2.5 for another example

17.1.5 Hardware Writemask Example

Using hardware writemasks when neither logic ops, nor alpha blending are enabled:

```
// Disable framebuffer reads (not shown)
// Set the writemasks
FBSoftwareWriteMask(0xFFFFFFFF) // 'Disable'
FBHardwareWriteMask(0xF0F0F0F0) // Actual writemask
```

18

Host Out

The Host Out Unit controls which data is available at the output FIFO, and gathers statistics about the rendering operations (picking and extent testing) and the synchronization of GLINT R4 via the **Sync** register.

18.1 Filtering

Filtering controls the data available at the output FIFO. There are a number of categories:

- depth, stencil and color: these are data values associated with a fragment which has been read from the localbuffer or framebuffer, or generated using the UpLoadData flag in the Framebuffer Write Unit.
- A single register, **Sync**, which is used to synchronize GLINT R4 and flush the graphics pipeline.
- Statistics: The registers associated with extent and picking.

The filtering is controlled by the **FilterMode** register which is split into 2 bit fields for each category. The 2 bit field selects whether the register tag and/or register data, are passed to the output FIFO. The format of the **FilterMode** register is shown in the table below.

Register Category	Tag Control Bit	Data Control Bit	Description
Diagnostic Use Only	0	1	
Diagnostic Use Only	2	3	
Depth	4	5	This is the data from image upload of the Depth (Z) buffer.
Stencil	6	7	This is the data from image upload of the Stencil buffer.
Color	8	9	This is the data from image upload of the Framebuffer (FBColor).
Synchronization	10	11	
Statistics		13	This is the data generated following a command to read back the results of the statistic measurements: PickResult, MaxHitRegion, MinHitRegion.
Diagnostic Use Only	14	15	

Table 18.10 Filter Modes

18.1.1 Filter Mode Example

// Set up Filter mode to only permit read back of

```
// synchronization tag and data
FilterMode(0x0C00)          // Set bits 10 & 11
```

18.1.2 Statistic Operations

There are two statistic collection modes of operation; picking and extent checking. Picking is normally used to select drawn objects or regions of the screen. Typically, extent checking is used to determine the bounds within which drawing has occurred so that a smaller area of the framebuffer can subsequently be cleared. Spans are handled by GLINT R4 in a fully consistent way for picking and extent checking.

Statistic collection is controlled using the **StatisticMode** register.

18.1.2.1 Picking

In picking mode, the active and/or passive fragments and spans have their associated XY coordinates compared against the coordinates specified in the **MinRegion** and **MaxRegion** registers. If the result is true, then the PickResult flag is set, otherwise it holds its previous state. The compare function can be either Inside or Outside. Before picking picking can start, the **ResetPickResult** register must be loaded to clear the PickResult flag.

The **MinRegion** and **MaxRegion** registers are loaded to select the region of interest for picking picking. A coordinate is inside the region if:

$X_{min} \leq X < X_{max}$

$Y_{min} \leq Y < Y_{max}$

where X and Y are from the fragment and the min/max values are from **MinRegion** and **MaxRegion** registers. This comparison is identical to the one used in the scissor tests.

The following stages are required for picking picking:

1) load **ResetPickResult**, **MinRegion** and **MaxRegion** registers
2) Set up the *FilterMode* to allow statistic commands out of GLINT R4 MX
3) Draw the primitives.
4) Send a **PickResult** command.
5) Poll the output FIFO while waiting for the PickResult to have passed through the pipeline.

18.1.2.2 Extent Checking

In extent mode, active and/or passive fragments have their associated XY coordinates compared to the **MinRegion** and **MaxRegion** registers and if found to be outside the defined rectangular region, then the appropriate register is updated with the new coordinate(s) to extend the region. The Inside/Outside bit has no effect in this mode. Block fills are included in the extent checking if the StatisticMode register is set to include spans.

The **MinRegion** and **MaxRegion** registers are loaded to select the maximum value (MinRegion) and minimum value (MaxRegion) for extent checking. A coordinate is inside the region if:

$X_{min} \leq X < X_{max}$

$Y_{min} \leq Y < Y_{max}$

where X and Y are from the fragment and the min/max values are from **MinRegion** and **MaxRegion** registers. This comparison is identical to the one used in the scissor tests.

Once all the necessary primitives have been rendered the results can be found using the **MinHitRegion** and **MaxHitRegion** commands, which cause the contents of the **MinRegion** and **MaxRegion** registers respectively to be written into the output FIFO (under control of the **FilterMode** register).

18.1.3 Synchronization

The **Sync** register is filtered and written to the output FIFO in a similar fashion to the other registers. If an interrupt is required then the most significant bit of the **Sync** command register must be set, and the filtering must be set up to write something into the FIFO. If nothing is written to the FIFO (because of the FilterMode) then no interrupt is generated.

The actual interrupt is not generated until the **Sync** data or tag has passed through. It is on the output of the FIFO, which allows low level resynchronization between the core and PCI clock domains. The FIFO has an extra bit in width to accommodate the interrupt signal. When both the data and tag are written into the FIFO only the first entry in the FIFO will cause the interrupt (assuming an interrupt was requested).

The remaining bits in the data field are free and can be used by the host to identify the reason for the Sync.

18.1.4 Registers

Filtering is controlled by the **FilterMode** register:

FilterMode
FilterModeAnd
FilterModeOr

Name	Type	Offset	Format
FilterMode	Output	0x8C00	Bitfield
FilterModeAnd	Output	0xAD00	Bitfield Logic Mask
FilterModeOr	Output	0xAD08	Bitfield Logic Mask
	Control registers		

Bits	Name	Read[5]	Write	Reset	Description
0…3	Reserved	✔	✔	x	Reserved for diagnostic use – set to 0
4	LBDepthTag	✔	✔	x	When set allows the *LBDepth* tag to be written into the output FIFO.
5	LBDepthData	✔	✔	xx	When set allows the data upload from the Depth buffer to be written into the output FIFO.
6	StencilTag	✔	✔	x	When set allows the LBStencil tag to be written into the output FIFO.

[5] Logic Op register readback is via the main register only

7	StencilData	✔	✔	x	When set allows the data upload from the Stencil buffer to be written into the output FIFO.
8	FBColorTag	✔	✔	x	When set allows the *FBColor* tag to be written into the output FIFO.
9	FBColorData	✔	✔	x	When set allows the data upload from the framebuffer to be written into the output FIFO.
10	SyncTag	✔	✔	x	When set allows Sync tag to be written into the output FIFO.
11	SyncData	✔	✔	x	When set allows the Sync data to be written into the output FIFO.
12	StatisticsTag	✔	✔	x	When set allows the *PickResult, MaxHitRegion* and *MinHitRegion* tags to be written into the output FIFO.
13	StatisticsData	✔	✔	x	When set allows the *PickResult, MaxHitRegion* and *MinHitRegion* data to be written into the output FIFO.
14	RemainderTag	✔	✔	x	When set allows any tags not covered by the categories in this table to be written into the output FIFO.
15	RemainderData	✔	✔	x	When set allows any data not covered by the categories in this table to be written into the output FIFO.
16…17	ByteSwap	✔	✔		This field controls the byte swapping of the data field when it is written into the output FIFO. The options are: 0 = ABCD (i.e. no swap) 1 = BADC 2 = CDAB 3 = DCBA
18	ContextTag	✔	✔	x	When set allows the *ContextData* and *EndOfFeedback* tags to be written into the output FIFO.
19	ContextData	✔	✔	x	When set allows the ContextData and *EndOfFeedback* data to be written into the output FIFO.
20	RunLength Encode Data	✔	✔	x	This bit, when set, will write run length encoded data into the host out FIFO.
21…31	Unused	0	0	x	

Notes: This register can only be updated if the *Security* register is set to 0.

Figure 18-1 FilterMode Register

Statistic collection is controlled by the StatisticMode register:

StatisticMode
StatisticModeAnd
StatisticModeOr

Name	**Type**	**Offset**	**Format**
StatisticMode	Output	0x8C08	Bitfield
StatisticModeAnd	Output	0xAD10	Bitfield Logic Mask
StatisticModeOr	Output *Command*	0xAD18	Bitfield Logic Mask

Bits	Name	Read	Write	Reset	Description
0	Enable	✔	✔	x	When set allows the collection of statistics information.
1	StatsType	✔	✔	x	Selects the type of staticstics to gather. The options are: 0 = Picking 1 = Extent
2	ActiveSteps	✔	✔	x	When set includes active fragments in the statistics gathering, otherwise they are excluded.
3	PassiveSteps	✔	✔	x	When set includes culled fragments in the statistics gathering, otherwise they are excluded.
4	Compare Function	✔	✔	x	Selects the type of compare function to use. The options are: 0 = Inside region 1 = Outside region
5	Spans	✔	✔	x	When set includes spans in the statistics gathering, otherwise they are excluded.
6..31	Unused	0	0	x	

Figure 18-2 StatisticMode Register

MinRegion, **MaxRegion** registers are used to load picking/extent regions, and **MaxHitRegion** and **MinHitRegion** are used to read the registers back. The format is 16 bit 2's complement numbers, X in the least significant end of the word.

PickResult is used to read the results of picking, the pick flag is placed in the least significant bit of the 32 bit register. **ResetPickResult** is used to clear the picking flag, the data field is not used.

The **Sync** register is 32 bits with the most significant bit set to indicate an interrupt is to be generated, bits 0-30 are available for the user.

18.1.5 Picking Example

Set the statistic mode to picking and detect any active fragments in the region 0x0 <= x < 0x100, 0x0 <= y < 0x100. Render some primitives then read back the results.

```
// Set filter mode as above
FilterMode(0x0C00)          // Set bits 10 & 11

// Set statistic mode
MinRegion(0)
MaxRegion(0x100 | 0x100 << 16)

// Clear the picking flag
ResetPickResult(0x0)      // Data not used
```

```
// Now render primitives.... ...
Render (render)   // All units set as appropriate

// All rendering finished.
// Set the filter mode to allow read back of Syncs
// and statistic information (tag and data)
FilterMode(0x3C00)          // Set bits 10 to 13

// Write to the PickResult register
PickResult(0x0)   // Data not used

// Now read the PickResult from the output FIFO (not shown)
```

18.1.6 Sync Interrupt Example

Generate a synchronization interrupt and encode some user defined data (0x34) in the lower 31 bits of the Sync register.

```
// Set up Filter mode to only permit read back of
// synchronization tag and data
FilterMode(0x0C00)          // Set bits 10 & 11

// Write to the Sync register with the top bit
// (bit 31) set and user data encoded into the
// lower bits (0-30)

sync = (0x1 << 31) | (0x34 & 0x7FFFFFFF)
Sync (sync)

// Now wait for the sync interrupt. (Not shown.)
```

19 Initialization

19.1 Initializing GLINT R4

This section illustrates how to initialize **GLINT R4** following reset, prior to carrying out rendering operations.

Initialization falls broadly into three areas, though in different systems precise responsibilities can vary:

- System initialization covers the PCI bus, memory set-up and video output. This information typically is only initialized once following reset.
- Window initialization covers the base address of the current rendering window and its color format. This must be initialized at reset and needs to be updated each time GLINT R4 starts drawing to a new window.
- Application initialization covers state that is typically dynamic, enabling and disabling depth testing for example. Again this state must be set at reset, but is likely to be updated relatively frequently.

To make use of the full functionality of GLINT R4 consult the relevant sections of Chapter 1 - Graphics Programming. Examples are given which make use of the pseudocode conventions given in Appendix B.

Note: *In general the graphics registers are not hardware initialized to specific values at reset. In the examples below it is assumed that the data structures used to load these registers are initialized to zero. Thus bit fields which are not set explicitly default to zero.*

19.1.1 Reset and initialisation

The units and FIFOs can be reset under software control or by a hardware reset signal, usually as a result of power-on.

During reset all the inter-unit FIFOs, the FIFOs between the core and the memory controller, and the host interface are emptied. Some of the units (Local Buffer Read and Framebuffer Read) also have internal FIFOs and these are cleared as well.

All the state machines in each unit are forced into their idle state so this together with the FIFOs being empty guarantees a safe start when the first message is received.

Note: *A reset does not, in general, change the contents of any state information which can be readback. After a power-on reset* all *these registers must be initialised by software to place them in a well defined state* before *any rendering is done. Units are* not *automatically disabled on a reset.*

19.2 System Initialization

19.2.1 PCI bus

There are a set of PCI related registers which can be interrogated for information about the chip, for example its revision and device ID. Some of these PCI related registers need to be set up at reset, for instance to configure the base addresses of the different memory regions of the chip. However, the subject of PCI bus initialization is beyond the scope of this document. For more details refer to the Reset chapter of the *GLINT R4 Reference Guide*, and the *PCI Local Bus Specification Rev2.1*.

19.2.2 Memory Configuration

There are no memory hardware configuration pins. Memory parameters are set through a group of registers in Region 0. These parameters are described in detail in the *GLINT R4 Reference Guide*, chapter 9 (Memory Systems) including register bitfields and sample configurations. The primary registers are **LocalMemCaps** and **LocalMemControl**. **LocalMemCaps** is show below.

LocalMemCaps

Name	Type	Offset	Format
LocalMemCaps	Memory Control Command register	0x1018	Bitfield

Bits	Name	Read	Write	Reset	Description
0..3	Column Address	✓	✓	0	Address bits to use for column address.
4..7	RowAddress	✓	✓	0	Address bits to use for row address.
8..11	BankAddress	✓	✓	0	Address bits to use for bank address.
12..15	ChipSelect	✓	✓	0	Address bits to use for chip select.
16..19	PageSize	✓	✓	0	Page size (units = full width of memory) 0 = 32 units 1 = 64 units, etc
20..23	RegionSize	✓	✓	0xF	Region size (units = full width of memory) 0 = 32 units 1 = 64 units, etc
24	NoPrecharge Opt	✓	✓	0	0 = off 1 = on
25	SpecialMode Opt	✓	✓	0	0 = off 1 = on
26	TwoColor BlockFill	✓	✓	0	0 = off 1 = on
27	Combine Banks	✓	✓	0	0 = off 1 = on
28	NoWriteMask	✓	✓	0x1	0 = off 1 = on
29	NoBlockFill	✓	✓	0x1	0 = off 1 = on
30	HalfWidth	✓	✓	0x1	0 = off 1 = on
31	NoLookAhead	✓	✓	0x1	0 = off 1 = on

Notes:
1. The ColumnAddress, RowAddress, BankAddress, and ChipSelect fields select the bits of the absolute physical address that are to be used to define corresponding parameters. Each value follows on from the previous one, so the ChipSelect value starts at ColumnAddress + RowAddress + BankAddress and continues for ChipSelect bits.
2. The PageSize field defines the size of the page, and the RegionSize field defines the size of the region of memory that each of the four page detectors should be assigned to (so that it is set to one quarter of the memory size).

19.2.3 Internal Video Timing Registers

Video Timing initialization is described in Volume I, chapter 5 (Video System).

19.2.4 Framebuffer Depth

The size of each pixel to be written into the framebuffer is set up using the **PixelSize** register. The two bit pixel size encoding field sets the pixel size to be used for merging the pixel data into the memory. It is normally set to the same value for all functions, but for generating texture maps it may be advantageous to use a different write pixel size.

The pixel size is taken from bits 0…1 when bit 31 is 0 or taken from subsequent bites for local functionality when bit 31 is 1. The two bit pixel size is encoded as follows:

- 0 = 32 bpp
- 1 = 16 bpp
- 2 = 8 bpp

During readback bits 0…17 and 31 return values as loaded and bits 18…30 return zero.

19.2.5 Screen Width

The visible screen width depends on the framebuffer configuration, screen clipping dimensions and RAMDAC setup. Framebuffer configuration is described in Volume I, section 3.5.1 (Framebuffer Dimensions and Depth).

19.2.6 Screen Clipping Region

R4 supports a screen scissor clip which should be set at system initialization, and a user scissor clip which should initially be disabled. Assuming that the relevant framebuffer registers[6] are set appropriately (see the *P4 Programmer's Guide* Volume I, chapter 4, "Buffer and Cache Management") then setting the screen clip prevents writing outside framebuffer memory. The following example would be appropriate for a resolution of 1024 by 768 pixels:

```
screenSize.X = 1024
screenSize.Y = 768
ScreenSize(ScreenSize)

scissorMode.ScreenScissorEnable = R4_ENABLE
scissorMode.UserScissorEnable =   R4_DISABLE
ScissorMode(ScissorMode)
```

19.2.7 Localbuffer and Framebuffer Configuration

R4 supports a range of localbuffer configurations. During initialization, fields in the **LBWriteFormat, LBWriteBufferWidth** and **LBReadFormat** registers should be set to appropriate values which reflect the depth of memory on the board design, and the initial manner in which it is to be used.

N.B. *The width of the Local and Frame buffers is needed toconvert x.y coordinates into a physical address (= Y * FBWriteBufferWidth[buffer] + X). The frame buffer height is not needed for this calculation.*

For example if the hardware is designed to support a 32 bit localbuffer, and initially this is to be divided into a 24 bit Depth buffer, 4 bit stencil and 4 GID planes then the registers must be set as follows (where "[mode]" = either destination or source):

```
lb[mode]ReadFormat.DepthWidth          = 1     // 24 bit depth buffer
lb[mode]ReadFormat.StencilPosition     = 8     // Stencil @ 24
```

[6] Framebuffer and Localbuffer memory is defined using source and destination read and write base addresses, offsets and widths for various formats and layouts. ScreenSize will then be a subset of the memory allocated to the buffers..

```
lb[mode]ReadFormat.StencilWidth      = 4     // 4 bit stencil
lb[mode]ReadFormat.GIDWidth          = 4
lb[mode]ReadFormat.GIDPosition       = 12    //GID @ 29

LB[MODE]ReadFormat(lb[mode]ReadFormat)

lbWriteFormat.DepthWidth             = 1     // 24 bit depth buffer
lbWriteFormat.StencilPosition        = 8     // Stencil @ 24
lbWriteFormat.StencilWidth           = 4     // 4 bit stencil
lbWriteFormat.GIDWidth               = 4
lbWriteFormat.GIDPosition            = 12    //GID @ 29

LBWriteMode(lbWriteFormat)
```

Note that within the limits of the memory depth that is physically available, it is possible to dynamically change the allocation of the bits, for instance on a per window basis.

Set the framebuffer and localbuffer source and/or destination read units to their default data sources:

```
fbSourceReadMode.DataType =   R4_FBSourceDATA
FBSourceReadMode(fbSouceReadMode)

lbSourceReadMode.DataType =   R4_LBSourceDEFAULT
LBSourceReadMode(lbSourceReadMode)
```

19.2.8 Host Out Unit

Under some circumstances it is necessary for the host to synchronize with GLINT R4. This is controlled using the **Sync** command which causes data to be written to the host out FIFO once all processing has completed. The host out FIFO should normally be initialized to pass these pieces of data (they can be filtered out).

The host out unit should normally be set to filter out all other output data, otherwise the host software must regularly poll the output FIFO to keep it drained and prevent it freezing the pipeline. For example:

```
filterMode.Depth            =   GLINT R4_NULL
filterMode.Stencil          =   GLINT R4_NULL
filterMode.Color            =   GLINT R4_NULL
FilterMode
      Synchronization       =   GLINT R4_FILTER_TAG_AND_DATA
                            // Allow syncs through
filterMode.Statistics       =   GLINT R4_NULL
FilterMode(filterMode)
```

19.2.9 Disabling Specialized Modes

The Graphic ID should normally be initially disabled using the **GIDMode** *FragmentEnable* bit. Refer to chapter 1 - Graphics Programming - for more details.

19.3 Window Initialization

GLINT R4 supports the concept of a window origin and makes it relatively simple to implement systems which allow different color formats to coexist in different windows.

19.3.1 Color Format

The Color formatting unit and the Alpha blend unit should be initialized to an appropriate color format at reset. The units support a variety of different formats - see the *GLINT R4 Reference Guide*, **AlphaBlendColor** register *ColorFormat* bitfield and related tables.

For example to render in 3:3:2, 8 bit color format, the following would be needed:

ditherMode.ColorFormat = **GLINT R4**_COLOR_FORMAT_RGB_332_FRONT

DitherMode(ditherMode)

alphaBlendColorMode.ColorFormat = **GLINT R4**_COLOR_FORMAT_RGB_332_FRONT

AlphaBlendColorMode(alphaBlendColorMode)

To enable dithering use the following:

```
ditherMode.Xoffset          =     0
ditherMode.Yoffset          =     0
ditherMode.DitherEnable     =     GLINT R4_ENABLE
ditherMode.UnitEnable       =     GLINT R4_ENABLE
DitherMode(ditherMode)
```

Note: *The color formatting unit is normally always enabled even if dithering itself is not. This is because the unit handles color formatting as well as the dithering operation.*

19.3.2 Setting the Window Address and Origin

R4 supports the concept of a current window origin. The origin of the window can be specified either as being in the Top Left or Bottom Left corner and (for Framebuffer functions) one of four destination buffers. This allows the user to pick the most appropriate coordinate system to use; for OpenGL it would typically be bottom left, whereas for an X windows implementation it would be Top Left. Thus for OpenGL set:

fbDestReadMode.Origin[1] = **R4**_BOTTOM_LEFT_WINDOW_ORIGIN

FBDestReadMode(fbDestReadMode)

lbDestReadMode.Origin[1] = **R4**_BOTTOM_LEFT_WINDOW_ORIGIN

LBDestReadMode(lbDestReadMode)

The window dimensions for clipping are set in the scissor unit. The ScissorMinXY register holds the minimum XY scissor coordinate - i.e. the rectangle corner closest to the screen origin. This information usually is provided by the window system. It needs updating if the

window moves. As an example if the position of the window is (200, 600 to 480,960) (using a bottom left coordinate system), the clipping coordinate is specified as follows:

ScissorMinXY = 200, 600

ScissorMaxXY = 480,960

To set the buffer origin using the **BufferAddress** and **BufferOffset** registers see *P10 Programmer's Guide* Volume 1, "Buffer and Cache Management". Unpatched addresses can be held using only the BufferAddress register(s). Patched address offsets must be held in the BufferOffset registers to convert the absolute memory address into a scree-relative address which can be used for patching.

19.3.3 Writemasks

Normally both the hardware (if present) and the software writemasks are initially set to make all bitplanes writeable:

FBSoftwareWriteMask(**GLINT R4**_ALL_WRITEMASKS_SET)

FBHardwareWriteMask(**GLINT R4**_ALL_WRITEMASKS_SET)

See Chapter 10, Framebuffer Writemasks, for more information.

19.3.4 Enabling Writing

Which buffers are enabled at any given time is window specific and should be considered for performance reasons. Performance will be improved if unnecessary reads from, and writes to, buffers are disabled. For example if the current rendering does not use depth, stencil, or pixel ownership testing, then reading and writing to the localbuffer may be disabled. The following example initializes the buffers to allow Z buffering and alpha blending:

```
fbWriteMode.WriteEnable          =   GLINT R4_ENABLE
FBWriteMode(fbWriteMode)
lbWriteMode.WriteEnable          =   GLINT R4_ENABLE
LBWriteMode(lbWriteMode)

lbSourceReadMode.Enable          =   GLINT R4_DISABLE
lbDestReadMode.Enable            =   GLINT R4_ENABLE
LBReadMode(lbReadMode)

fbSourceReadMode.ReadEnable      =   GLINT R4_DISABLE
fbDestReadMode.ReadEnable        =   GLINT R4_ENABLE
FBDestReadMode(fbDestReadMode)
```

Note: *to use software writemasking, the **FBDestReadMode** register's* ReadEnable *field needs to be set if the writemask is set to other than all 1's.*

19.4 Application Initialization

While an application is running it may dynamically use features of GLINT R4 such as depth buffering, alpha blending, logical operations, etc. Initially, however, it is recommended that the respective units be disabled to ensure they are in a known state:

```
areaStippleMode.Enable              =  GLINT R4_DISABLE
AreaStippleMode(areaStippleMode)

lineStippleMode.StippleEnable       =  GLINT R4_DISABLE
LineStippleMode(lineStippleMode);

routerMode.Sequence                 =  GLINT R4_SET
RouterMode(routerMode)                 //Set to skip texture since stencil and depth disabled//

stencilMode.UnitEnable              =  GLINT R4_DISABLE
StencilMode(stencilMode)

depthMode.Enable                    =  GLINT R4_DISABLE
DepthMode(depthMode)

colorDDAMode.Enable                 =  GLINT R4_DISABLE
ColorDDAMode(colorDDAMode)

textureCoordMode.Enable             =  GLINT R4_DISABLE
TextureCoordMode(textureCoordMode)

textureIndexMode.Enable             =  GLINT R4_DISABLE
TextureIndexMode(textureIndexMode)

textureReadMode.Enable              =  GLINT R4_DISABLE
textureReadMode(textureReadMode)

TextureCompositeColorMode.Enable    =  GLINT R4_DISABLE
TextureCompositeColorMode(TextureApplicationMode)

fogMode.Enable                      =  GLINT R4_DISABLE
FogMode(fogMode)

antialiasMode.Enable                =  GLINT R4_DISABLE
AntialiasMode(antialiasMode)

alphaTestMode.Enable                =  GLINT R4_DISABLE
AlphaTestMode(alphaTestMode)

alphaBlendAlphaMode.Enable          =  GLINT R4_DISABLE
AlphaBlendAlphaMode(alphaBlendAlphaMode)

alphaBlendColorMode.Enable          =  GLINT R4_DISABLE
AlphaBlendColorMode(alphaBlendColorMode)

logicalOpMode.Enable                =  GLINT R4_DISABLE
LogicalOpMode(logicalOpMode)
```

20
Performance Tips

The following is a list of software programming tips and techniques which can be applied to maximize GLINT R4 performance. Many of these are debug aids and the importance of effective debug techniques cannot be overemphasised:

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

- Maurice Wilkes discovers debugging, 1949

The list is intended to be suggestive only and refers back to the *GLINT R4 Reference Guide* and earlier chapters of the *Programmers Guide*.

- Using Block Writes - e.g. for clears
- Fast double buffering in a window
- Disable FB reads-per-pixel if they are not required
- Incrementing addresses when writing to the FIFO to enable PCI burst transfers
- Using PCI Disconnect under PIO
- Using bus mastership (i.e. DMA)
- Improving DMA bus bandwidth utilization using the indexed FIFO modes
- Disabling units that are not in use (e.g. Framebuffer reads)
- Use of the extent register to minimize the area in the localbuffer and framebuffer that needs to be cleared
- Use of the GLINT R4 graphics pipeline in preference to the framebuffer (and/or localbuffer) bypass when possible
- Loading registers in unit order (i.e. Rasterizer first - Host Out last)
- Avoiding unnecessary register updates
- Miscellaneous debug and generic graphics tips

20.1 Block Writes

GLINT R4 boards are equipped with either SGRAM that supports block writes or SDRAM which does not. This allows up to 32 pixels at a time to be filled with a constant color by a single framebuffer write access. This can lead to roughly a 32fold increase in the speed of, for instance, clearing a large area of the framebuffer.

While this technique is most useful when clearing the framebuffer, it can be used to fill any trapezoid. See volume I, section 4.3.3 - Block Writes.

20.2 Fast double buffering in a window

Double buffering is a technique used to achieve visually smooth animation, by rendering a scene to an offscreen buffer, before quickly displaying it.

GLINT R4 board designs can readily support a variety of double buffering mechanisms depending on the memory configuration and LUT-DAC used, including:

- BLT
- Full Screen

Note: The best results can often be achieved by combining double buffering techniques.

20.3 Disable FB Reads per pixel if not required

The *AlphaFiltering* bit in **FBDestReadMode** can reduce unnecessary FB reads. When set, it compares the fragment's alpha value and if it is equal to the AlphaReference value (held in **FBReadEnables**) then no read is done. This saves memory bandwidth when the destination color doesn't contribute to the fragment's color during blending.

20.4 Improving PCI bus bandwidth for Programmed I/O and DMA

Writing data values into the memory mapped registers is appropriate for primitives which require few set-up parameters such as 2D lines.

For more complex primitives such as Gouraud shaded triangles where a significant number of registers must be loaded for each primitive, it may be more efficient to write directly to the FIFO input.

The advantage of this mechanism is that it is then possible to use DMA burst transfers.

The disadvantage is that both the address of the register and the data value to be loaded must be written, apparently doubling the amount of data to be loaded.

However, to improve DMA bus bandwidth utilization, the registers have been grouped into blocks which frequently all need to be updated together, and an indexed addressing mode is supported which allows a single "address" to be loaded, followed by the data for a whole set of registers.

An additional mode is supported which allows a large number of data values to be loaded to the same register. This is useful for image downloads.

It may also be possible to reduce DMA overhead by re-using DMA buffers and vertex buffers. The HostInID register can be used to mark any point in the command stream so that the use of index and vertex buffers can be monitored. This register is loaded with an ID field; like the DMA address register, which can be read at any time to check the progress of the command stream.

20.5 PCI burst transfers under Programmed I/O

PCI bus burst transfers typically allow up to four times the bandwidth of individual transfers.

However burst transfers are only initiated on the PCI bus when successive addresses are being written to (i.e. the byte address is incremented by 4). To facilitate the use of burst transfers when using programmed I/O to load the GLINT R4 FIFOs, GLINT R4 multiply maps the FIFO input register throughout the range:

0x00002000 to 0x00002FFF in region 0

Thus when data is being loaded into the FIFO a software loop should be written which starts by writing the first data item at the lower extreme of this address range, and works towards the upper.

20.6 Using PCI Disconnect Under Programmed I/O

The PCI bus protocol incorporates a feature known as PCI Disconnect which is supported by GLINT R4. Once GLINT R4 is in this mode, if the host processor attempts to write to the full FIFO then instead of the write being lost, GLINT R4 asserts PCI Disconnect which forces the host processor to retry the write cycle until it succeeds.

This feature allows faster download of data to GLINT R4 since the host need not poll the **InFIFOSpace** register. However it should be used carefully because the bus is then effectively hogged by the host processor until GLINT R4 frees up an entry in its FIFO.

20.7 Using Bus Mastership (DMA)

Most GLINT R4 boards support PCI bus mastership, allowing the on-board DMA of GLINT R4 to be used to copy data from host memory into GLINT R4 FIFO.

Bus mastership mode is asserted in the **CFGCommand** register using bit 2, *BusMasterEnable*.

The use of PCI bus mastership has a number of benefits:

- PCI bus bandwidth utilization is generally much improved. GLINT R4 has been measured achieving transfer rates of up to 30-40MBytes/sec with a fast host slave (P90 Neptune chipset).
- PCI bus bandwidth is further improved because the driver software no longer needs to poll the FIFO flags to find how many entries are empty, before loading it.
- Overall system performance may benefit through increased parallelism between GLINT R4 and the host, as the host can often perform useful work preparing the next DMA buffer once it has initiated one DMA transfer.

20.8 Disabling units not in use

As a general rule any units within GLINT R4 which are not actively in use for the current rendering should be disabled. Each unit has a bit in a control register for this purpose. This will maximize pixel throughput in the graphics core.

In particular it is important to check that unnecessary reads of the localbuffer are not taking place. For instance it is perfectly possible to set up the localbuffer read unit such that GLINT R4 reads per pixel information (such as Z, stencil and GID data) which is then discarded. The effect will be the same visually, but the cost in performance of making the memory accesses will be very high.

Similar comments apply for the framebuffer read unit which again should only be enabled to read pixel data when it is essential.

Note *GLINT R4 boards typically support hardware writemasks and these should be used in preference to the software writemasks.*

20.9 Clearing the localbuffer & framebuffer

GLINT R4 can be instructed in the StatisticsMode StatsType register field to maintain a record of the minimum bounding box (**MinRegion** and **MaxRegion** registers) that has

been rendered to, in a given period. This can be used to limit the area that must be cleared down using span fill.

For further details see chapter 11, Host Out, on Extent Checking

20.10 Use of the Framebuffer (or Localbuffer) Bypass

Whenever possible rendering should be done through the GLINT R4 graphics pipeline. This is because reading and writing the framebuffer (or localbuffer) using the bypass is relatively slow. In some cases performance may even be improved if a small area of the framebuffer (and/or localbuffer) is uploaded through the graphics pipeline into a bitmap, rendered to, and then downloaded again through the graphics pipeline.

20.11 Loading Registers in Unit Order

To maximize performance, the control registers for the next primitive should be loaded into the GLINT R4 FIFO in unit order. Thus the registers associated with the Rasterizer unit should be loaded first, then Scissor unit, Stipple unit, Color DDA, and so on until the last unit to be loaded is the Host Out unit (if necessary). Then finally the relevant command register should be loaded.

For the order of the units refer to Volume II, *Graphics Programming*.

20.12 Avoiding Unnecessary Register Updates

GLINT R4 control registers retain their value between one primitive and the next. Thus it is not necessary to reload registers that are unchanged between primitives. e.g. the dY register usually is set to either +1 or -1 (except when antialiasing).

In addition calculations of register values can often be shared across primitives, for instance between edges in adjacent polygons in meshes.

20.13 Hardware and Software Context Dumps

GLINT R4 supports **ContextDump** and **ContextRestore** commands and a **StatisticsMode** register, with enables for extent checking and picking set in the **FilterMode** register. These allow the selection of active and passive fragments by screen area and other parameters at specific points in the render process, and state switching to halt and resume graphic processing while examining the collected data.

The decision to use hardware context management may depend on the software regime being supported. In the D3D environment it may be more effective to save all the context state in software copies. When a context is switched to, simply set up the chip again. This avoids the need to wait for a Context Dump before switching away from a context and takes advantage of D3D's capabilities. However the hardware-assisted route is generally preferred by OGL developers.

20.14 Use the Memory Scratchpad Registers

By keeping track of which primitives have finished rendering it is often possible to avoid waiting for chip syncs. When applications do procedural tetxuring they need to change the texture every frame. Normally host access to a texture that has been rendered with requires a chip-sync. Using scratchpad memory to keep track of primitives which have finished rendering allows the driver to confirm that the last primitive to use that texture has indeed completed and the application can now access the texture immediately with no sync. As long as applications only want to change the texture some time after they

rendered with it (the best time is just before rendering the new version) then chip-syncs can be almost entirely avoided.

The same approach can be used when the application is changing render target and doing a render-to-texture or blit-to-texture. Similarly, when the driver is texture swapping, it can tell which textures it can and can't touch using this tracking information.

20.15 Miscellaneous Tips

The following is a set of miscellaneous tips that are not GLINT R4 specific but well worth using.

- Avoid polling for Vblank whenever possible but if you have to poll, consider whether your application is taking just longer than an integer number of Vblank intervals to draw a frame - slightly simplifying the frame to make it just under an integer multiple can dramatically improve performance.
- Another way of looking at the same problem is, if you remove your SwapBuffers() calls, does your application render many more frames per second? If so, you might be spending a lot of time waiting for buffer swaps, and you should tune your app so that it draws just enough to fit in one less frame time.
- When using DMA it may be best to flush the DMA buffer to the chip after entering a large primitive in the buffer (e.g. screen clear), so that the chip is doing useful work while further primitives are being prepared on the host.
- Minimize the use of the Sync command.
- Does making your window smaller cause things to speed up? If so, you're probably fill-limited (bottlenecked by filling the pixels in the window). Speed things up by reducing the depth complexity of your scene or by using simpler drawing operations wherever possible (e.g., avoiding depth-buffering for the background or ground plane).
- Does making your window smaller have no effect on the time it takes to draw a frame? If so, you're probably geometry-limited (bottlenecked by transformations, clipping, or lighting) or host-limited.
- Measure the time it takes your application to draw a frame. Now comment out all the drawing calls, and measure again. If most of the elapsed time per frame is spent doing things other that drawing, your application is probably host-limited rather than geometry-limited.
- If you're geometry-limited, you can speed things up by using simpler models with fewer vertices, by reducing the amount of clipped geometry, by using fewer light sources, etc. If you're host-limited you should use profiling tools to figure out where your application is spending its time and then tune those areas.

21 Appendices

21.1 Pseudocode Definitions

In many areas of the document we use fragments of pseudocode to describe register loading. These are based on a C interface to GLINT R4 in which each 32 bit register is represented as a C structure, potentially split into a series of bit fields.

Where in an example only a subset of the bit fields in a register are set, it is assumed either that a software copy of the register is being modified, or that the current contents of the register have first been read back. This style has been chosen for clarity; there are often more efficient strategies.

The constant definitions and register bit field definitions are based upon those used in the 3Dlabs driver software. Sources including header files are available under source license agreement.

Loading of a GLINT R4 register is expressed as:

```
register-name(value)
```

When writing directly to the register file (i.e. to a FIFO) this would be implemented by writing "value" to the mapped-in address of the register called "register-name".

Fragmentary examples are not in strict C syntax, a typical example is:

```
// Sample code to rasterize a 10x10 rectangle at the
// framebuffer origin.

StartXDom(0)        // Start dominant edge
StartXSub(1<<16)    // Start of subordinate
dXDom(0x0)
dXSub(0x0)
Count(0xA)
YStart(0)
dY(1<<16)

// Set-up to render an aliased trapezoid.
render.AreaStippleEnable = GLINT R4_DISABLE
render.LineStippleEnable = GLINT R4_DISABLE
render.PrimitiveType = GLINT R4_TRAPEZOID
render.FastFillEnable = GLINT R4_DISABLE
render.FastFillIncrement = don't care
```

render.UsePointTable = **GLINT R4**_FALSE

render.AntialiasEnable = **GLINT R4**_DISABLE

render.AntialiasingQuality = *don't care*

render.ResetLineStipple = **GLINT R4**_FALSE

render.SyncOnBitMask = **GLINT R4**_FALSE

render.SyncOnHostData = **GLINT R4**_FALSE

Render(render) // Render the rectangle

Code is shown in roman face and comments are C++ style '//' indicating that the rest of the line is a comment. Any statement which ends in parenthesis is a register update, other statements will generally be variable assignments.

A variable, say render, is of a type associated with the register being modified. This will usually be clear by the context and will not usually be declared as such. All the type definitions are in the header files. The values assigned to a register will be either a variable as described above, a macro i.e. GLINT R4_TRUE , as found in the headers, or an immediate constant in C style format (e.g. 0x45). In registers with several fields some of which are not relevant to a particular example, the field can be ignored completely or set to *don't care*. In some registers values for fields which need to be set are not readily available. These are typically set as appropriate.

For some fragments we simply give a list of register updates e.g.:

```
// Sample code to rasterize a rectangle

StartXDom()          // Start dominant edge
StartXSub()          // Start of subordinate
dXDom()
dXSub()
Count()
YStart()
dY()

// Set-up to render an aliased trapezoid.

Render()             // Render the rectangle
```

This technique is used to give a feel for the registers involved in a particular operation and where a detailed treatment is not warranted.

To take the address of a register, the name is used, thus this example stores the address of the StartXDom register in the buffer pointed to by the variable buf and increments the pointer:

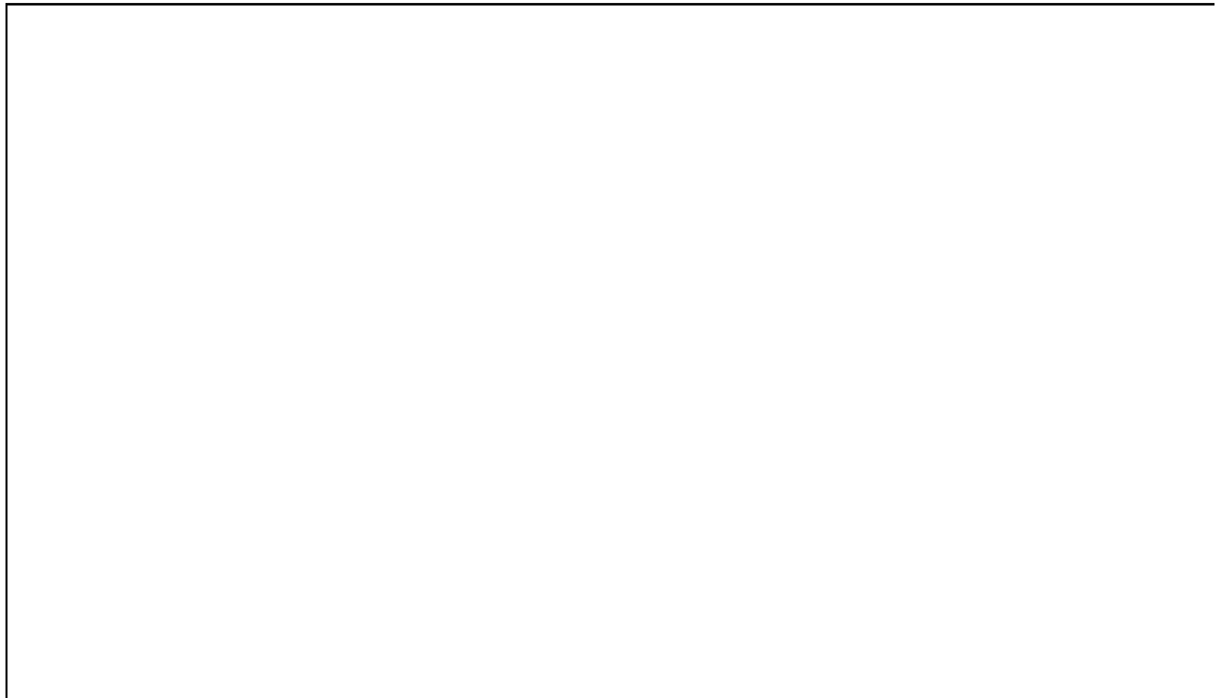
```
*buf++ = StartXDom
```

To test the value of a register the register name is dereferenced using the C '*' operator as for instance in this example which tests for the completion of a DMA operation:

```
while( *DMACount != 0 ) ;
```

21.2 Delta (Manual) Programming Example

The following examples demonstrates how to render a depth buffered, Gouraud shaded triangle mesh using the R4 in standalone mode. The window into which the rendering takes place is partially obscured and hence is clipped by two clip rectangles. The geometry of the mesh and clip regions is shown in the following diagram:



The three examples cover drawing the mesh as a set of points at the vertices, as connected line segments and finally as filled triangles. Note that the triangles can have any orientation or shape and are not restricted to flat topped or bottomed ones used in the example.

```
// This is the header file for the GLINT R4 example code.
// It only contains the necessary items to support the examples.

#ifdef BIG_ENDIAN
// The DeltaMode register fields.

typedef struct {
  unsigned int pad:                     15;
  unsigned int TextureParameterMode:    2;
  unsigned int ClampEnable:             1;
  unsigned int NoDraw:                  1;
  unsigned int DiamondExit:             1;
  unsigned int SubPixelCorrectionEnable: 1;
  unsigned int DiffuseTextureEnable:    1;
  unsigned int SpecularTextureEnable:   1;
  unsigned int DepthEnable:             1;
  unsigned int SmoothShadingEnable:     1;
  unsigned int TextureEnable:           1;
  unsigned int FogEnable:               1;
  unsigned int DepthFormat:             2;
  unsigned int TargetChip:              2;
} __DeltaModeFmat;

// The DrawTriangle and DrawLine command fields.

typedef struct {
  unsigned int pad:                     15;
  unsigned int SubPixelCorrectionEnable: 1;
  unsigned int CoverageEnable:          1;
  unsigned int FogEnable:               1;
  unsigned int TextureEnable:           1;
  unsigned int SyncOnHostData:          1;
  unsigned int SyncOnBitMask:           1;
  unsigned int UsePointTable:           1;
  unsigned int AntialiasingQuality:     1;
  unsigned int AntialiasEnable:         1;
  unsigned int PrimitiveType:           2;
  unsigned int reserved:                2;
```

```
  unsigned int FastFillEnable:          1;
  unsigned int ResetLineStipple:        1;
  unsigned int LineStippleEnable:       1;
  unsigned int AreaStippleEnable:       1;
} __DeltaRenderFmat;

#else

// The DeltaMode register fields.

typedef struct {
  unsigned int TargetChip:              2;
  unsigned int DepthFormat:             2;
  unsigned int FogEnable:               1;
  unsigned int TextureEnable:           1;
  unsigned int SmoothShadingEnable:     1;
  unsigned int DepthEnable:             1;
  unsigned int SpecularTextureEnable:   1;
  unsigned int DiffuseTextureEnable:    1;
  unsigned int SubPixelCorrectionEnable: 1;
  unsigned int DiamondExit:             1;
  unsigned int NoDraw:                  1;
  unsigned int ClampEnable:             1;
  unsigned int TextureParameterMode:    2;
  unsigned int pad:                     16;
} __DeltaModeFmat;

// The DrawTriangle and DrawLine command fields.

typedef struct {
  unsigned int AreaStippleEnable:       1;
  unsigned int LineStippleEnable:       1;
  unsigned int ResetLineStipple:        1;
  unsigned int FastFillEnable:          1;
  unsigned int reserved:                2;
  unsigned int PrimitiveType:           2;
  unsigned int AntialiasEnable:         1;
  unsigned int AntialiasingQuality:     1;
```

```
  unsigned int UsePointTable:            1;
  unsigned int SyncOnBitMask:            1;
  unsigned int SyncOnHostData:           1;
  unsigned int TextureEnable:            1;
  unsigned int FogEnable:                1;
  unsigned int CoverageEnable:           1;
  unsigned int SubPixelCorrectionEnable: 1;
  unsigned int pad:                      15;
} __DeltaRenderFmat;
#endif

// The tag values for the registers.
#define __Delta_V0FloatTag              0x230
#define __Delta_V1FloatTag              0x240
#define __Delta_V2FloatTag              0x250
#define __DeltaTagDeltaMode             0x260
#define __DeltaTagDrawTriangle          0x261
#define __DeltaTagRepeatTriangle        0x262
#define __DeltaTagDrawLine01            0x263
#define __DeltaTagDrawLine10            0x264
#define __DeltaTagRepeatLine            0x265
#define __DeltaTagBroadcastMask         0x26f

// Some temp defines to keep things compiling easily.
#define DrawTriangleTag     __DeltaTagDrawTriangle
#define DrawLine01Tag       __DeltaTagDrawLine01
#define DrawLine10Tag       __DeltaTagDrawLine10
#define RepeatTriangleTag   __DeltaTagRepeatTriangle
#define RepeatLineTag       __DeltaTagRepeatLine
```

```
#include "delta.h"
#include <stdio.h>

extern unsigned long *dmaPtr;
extern DMA *dma;

// Change these macros to what is needed to write the values to GLINT Delta,
// or add them to a dma buffer.
#define LD_REG(reg, value)  dmaPtr = dma->Space (2); *dmaPtr++ = reg; \
                                         *dmaPtr++ = value;
#define LD_PARAM(reg, value) dmaPtr = dma->Space (2); *dmaPtr++ = reg;
\
                                           *dmaPtr++ = *((unsigned long *) &value);

// Prototypes
void PointMesh (gal &cx);
void LineMesh (gal &cx);
void TriangleMesh (gal &cx);


// Simple structure to use in the example code

typedef struct { float  x, y, z, r, g, b, a; }  Vertex;
typedef struct { short  x, y; }                 XY;
typedef struct { XY scissorMin, scissorMax; }   ClipRectangle;


// Define some test data.

#define verticesInMesh 7
Vertex  mesh[verticesInMesh] = {
        // x    y  z    r    g    b    a
        {  10, 300, 0.1, 1.0, 1.0, 1.0, 1.0 },
        {  60, 100, 0.2, 1.0, 1.0, 0.0, 1.0 },
        { 110, 300, 0.3, 1.0, 0.0, 1.0, 1.0 },
        { 160, 100, 0.4, 1.0, 0.0, 0.0, 1.0 },
        { 210, 300, 0.5, 0.0, 1.0, 1.0, 1.0 },
        { 260, 100, 0.6, 0.0, 1.0, 0.0, 1.0 },
```

```
        { 310, 300, 0.7, 0.0, 0.0, 1.0, 1.0 }};

#define numberClipRectangles 2

ClipRectangle clipRectangles[numberClipRectangles] = {
        { {110, 0}, {400, 150} },
        { {0, 150}, {400, 350} }};


enum {paramS, paramT, paramQ, paramKs, paramKd, paramR, paramG,
paramB,
        paramA, paramF, paramX, paramY, paramZ};
```

```
// This function draws the vertices in the mesh as points.  There is no direct
// support for points in R4 as they do not need any set up calculations.
// The R4 Delta unit can be used to plot points (maybe because you want to
// always work in floating point) by having GLINT Delta do the set up
// calculations for
// a line, but tell the rendering device to render points.

void PointMesh (gal &cx)
{
    __DeltaModeFmat         deltaMode;
    __DeltaRenderFmat       drawCmd;
    int                     rect, v;


    // Assume the rendering device is already initialized.
    // Note we expect the BiasCoords mode in the RasterizerMode register
    // to be set to add a bias of zero.

    // Set up the DeltaMode register.
    deltaMode.pad                       = 0;
    deltaMode.TextureParameterMode      = 1;    // Clamp.
    deltaMode.ClampEnable               = 1;    // Clamp enabled.
    deltaMode.NoDraw                    = 0;    // Do drawing.
    deltaMode.DiamondExit               = 0;    // Not needed for this example.
    deltaMode.SubPixelCorrectionEnable  = 0;    // No sub pixel correction.
    deltaMode.DiffuseTextureEnable      = 0;    // Disable.
    deltaMode.SpecularTextureEnable     = 0;    // Disable.
    deltaMode.DepthEnable               = 1;    // Enable.
    deltaMode.SmoothShadingEnable       = 1;    // Enable.
    deltaMode.TextureEnable             = 0;    // Disabled.
    deltaMode.FogEnable                 = 1;    // Enabled, but controlled from
                                                // the draw command.
    deltaMode.DepthFormat               = 2;    // 24 bit Z buffer.
    deltaMode.TargetChip                = 0;    // GLINT 300SX

    LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));

    // Set up the draw command data.
```

```
drawCmd.pad                      = 0;
drawCmd.SubPixelCorrectionEnable = 0;    // Enable.
drawCmd.CoverageEnable           = 0;    // Disable.
drawCmd.FogEnable                = 0;    // Disable.
drawCmd.TextureEnable            = 0;    // Disable.
drawCmd.SyncOnHostData           = 0;    // Disable.
drawCmd.SyncOnBitMask            = 0;    // Disable.
drawCmd.UsePointTable            = 0;    // Disable.
drawCmd.AntialiasingQuality      = 0;    // Not used.
drawCmd.AntialiasEnable          = 0;    // Disable.
drawCmd.PrimitiveType            = 2;    // ** Point **
drawCmd.reserved                 = 0;
drawCmd.FastFillEnable           = 0;    // Disable.
drawCmd.ResetLineStipple         = 0;    // Disable.
drawCmd.LineStippleEnable        = 0;    // Disable.
drawCmd.AreaStippleEnable        = 0;    // Disable.
```

```
// We need to ensure that the end vertex of the line (in V1) can
// never be the same as the point vertices.  Any X (or Y) coordinate
// which is out of the normal range (0.0 to screen width) will do
// so in this case an X of -1.0 has been used.

float   tempEndCoord = -1.0;

LD_PARAM ((__Delta_V1FloatTag + paramX), tempEndCoord);
```

```
    for (v = 0; v < verticesInMesh; v++)
    {

        LD_PARAM((__Delta_V0FloatTag + paramR), mesh[v].r);
        LD_PARAM((__Delta_V0FloatTag + paramG), mesh[v].g);
        LD_PARAM((__Delta_V0FloatTag + paramB), mesh[v].b);
        LD_PARAM((__Delta_V0FloatTag + paramA), mesh[v].a);
        LD_PARAM((__Delta_V0FloatTag + paramX), mesh[v].x);
        LD_PARAM((__Delta_V0FloatTag + paramY), mesh[v].y);
        LD_PARAM((__Delta_V0FloatTag + paramZ), mesh[v].z);

        for (rect = 0; rect < numberClipRectangles; rect++)
        {
            // Load in the scissor rectangle.
            LD_REG(ScissorMinXYTag, (clipRectangles[rect].scissorMin.y << 16
|
                            clipRectangles[rect].scissorMin.x));
            LD_REG(ScissorMaxXYTag, (clipRectangles[rect].scissorMax.y <<
16 |
                            clipRectangles[rect].scissorMax.x));

            if (rect == 0)
            {
                LD_REG(DrawLine01Tag, *((long *) &drawCmd));
            }
            else
            {
                LD_REG(RepeatLineTag, 0);       // data field not used.
            }
        }
    }
}
```

```
// This array holds the order we are going to visit the
// vertices in to draw each line segment.

int lineOrder[12] = {1, 0, 2, 4, 6, 5, 4, 3, 2, 1, 3, 5};

// This function draws the mesh as a series of lines.  The order the lines are
// drawn in is hardcoded (this is only an example!).

void LineMesh (gal &cx)
{
   __DeltaModeFmat          deltaMode;
   __DeltaRenderFmat     drawCmd;
   int                      vertexStore;
   int                   rect, i, v;


   // Assume the rendering device is already initialized.  Note we expect the
   // BiasCoords mode in the RasterizerMode register to be set to
   // add a bias of zero.

   // Set up the DeltaMode register.
   deltaMode.pad                    = 0;
   deltaMode.TextureParameterMode      = 2;    // Auto normalize.
   deltaMode.ClampEnable               = 1;    // Clamp enabled.
   deltaMode.NoDraw                    = 0;    // Do drawing.
   deltaMode.DiamondExit               = 1;    // Not needed for this example.
   deltaMode.SubPixelCorrectionEnable  = 1;    // Enable sub pixel correction.
   deltaMode.DiffuseTextureEnable      = 0;    // Disable.
   deltaMode.SpecularTextureEnable     = 0;    // Disable.
   deltaMode.DepthEnable               = 1;    // Enable.
   deltaMode.SmoothShadingEnable       = 1;    // Enable.
   deltaMode.TextureEnable             = 0;    // Disabled.
   deltaMode.FogEnable                 = 1;    // Enabled, but controlled from
                                               // the draw command.
   deltaMode.DepthFormat               = 2;    // 24 bit Z buffer.
   deltaMode.TargetChip                = 0;    // GLINT 300SX

   LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));
```

```
// Set up the draw command data.

drawCmd.pad                        = 0;
drawCmd.SubPixelCorrectionEnable   = 1;    // Enable.
drawCmd.CoverageEnable             = 0;    // Disable.
drawCmd.FogEnable                  = 0;    // Disable.
drawCmd.TextureEnable              = 0;    // Disable.
drawCmd.SyncOnHostData             = 0;    // Disable.
drawCmd.SyncOnBitMask              = 0;    // Disable.
drawCmd.UsePointTable              = 0;    // Disable.
drawCmd.AntialiasingQuality        = 0;    // Not used.
drawCmd.AntialiasEnable            = 0;    // Disable.
drawCmd.PrimitiveType              = 0;    // Line.
drawCmd.reserved                   = 0;
drawCmd.FastFillEnable             = 0;    // Disable.
drawCmd.ResetLineStipple           = 0;    // Disable.
drawCmd.LineStippleEnable          = 0;    // Disable.
drawCmd.AreaStippleEnable          = 0;    // Disable.
```

```
for (i = 0; i < 12; i++)
{
   v = lineOrder[i];
   vertexStore = __Delta_V0FloatTag + 16 * (i % 2);

   LD_PARAM((vertexStore + paramR), mesh[v].r);
   LD_PARAM((vertexStore + paramG), mesh[v].g);
   LD_PARAM((vertexStore + paramB), mesh[v].b);
   LD_PARAM((vertexStore + paramA), mesh[v].a);
   LD_PARAM((vertexStore + paramX), mesh[v].x);
   LD_PARAM((vertexStore + paramY), mesh[v].y);
   LD_PARAM((vertexStore + paramZ), mesh[v].z);

   if (i >= 1)
   {
      // We now have enough vertices to draw a line.
      for (rect = 0; rect < numberClipRectangles; rect++)
      {
         // Load in the scissor rectangle.
         LD_REG(ScissorMinXYTag, (clipRectangles[rect].scissorMin.y <<
16 |
                         clipRectangles[rect].scissorMin.x));
         LD_REG(ScissorMaxXYTag, (clipRectangles[rect].scissorMax.y <<
16 |
                         clipRectangles[rect].scissorMax.x));

         if (rect == 0)
         {
            if (i & 1)
            {
               LD_REG(DrawLine01Tag,  *((long *) &drawCmd));
            }
            else
            {
               LD_REG(DrawLine10Tag,  *((long *) &drawCmd));
            }
         }
         else
```

```
                {
                    LD_REG(RepeatLineTag, 0);        // data field not used.
                }
            }
        }
    }
}
```

```
// This function draws the mesh as a series of shaded triangles.

void TriangleMesh (gal &cx)
{
   __DeltaModeFmat   deltaMode;
   __DeltaRenderFmat      drawCmd;
   int                    vertexStore;
   int                    rect, v;


   // Assume the rendering device is already initialized.  Note we expect the
   // BiasCoords mode in the RasterizerMode register to be set to
   // add a bias of zero.

   // Set up the DeltaMode register.
   deltaMode.pad                        = 0;
   deltaMode.TextureParameterMode                    = 2;    // Auto normalize.
   deltaMode.ClampEnable               = 1;    // Clamp enabled.
   deltaMode.NoDraw                    = 0;    // Do drawing.
   deltaMode.DiamondExit               = 1;    // Not needed for this example.
   deltaMode.SubPixelCorrectionEnable  = 1;    // Enable sub pixel correction.
   deltaMode.DiffuseTextureEnable        = 0;    // Disable.
   deltaMode.SpecularTextureEnable       = 0;    // Disable.
   deltaMode.DepthEnable               = 1;    // Enable.
   deltaMode.SmoothShadingEnable         = 1;    // Enable.
   deltaMode.TextureEnable             = 0;    // Disabled.
   deltaMode.FogEnable                 = 1;    // Enabled, but controlled from
                                   // the draw command.
   deltaMode.DepthFormat               = 2;    // 24 bit Z buffer.
   deltaMode.TargetChip                = 0;    // GLINT 300SX

   LD_REG (__DeltaTagDeltaMode, *((long *) &deltaMode));

   // Set up the draw command data.

   drawCmd.pad                         = 0;
   drawCmd.SubPixelCorrectionEnable                        = 1;    // Enable.
   drawCmd.CoverageEnable                = 0;    // Disable.
```

```
drawCmd.FogEnable              = 0;   // Disable.
drawCmd.TextureEnable          = 0;   // Disable.
drawCmd.SyncOnHostData         = 0;   // Disable.
drawCmd.SyncOnBitMask          = 0;   // Disable.
drawCmd.UsePointTable          = 0;   // Disable.
drawCmd.AntialiasingQuality    = 0;   // Not used.
drawCmd.AntialiasEnable        = 0;   // Disable.
drawCmd.PrimitiveType          = 1;   // Trapezoid.
drawCmd.reserved               = 0;
drawCmd.FastFillEnable         = 0;   // Disable.
drawCmd.ResetLineStipple       = 0;   // Disable.
drawCmd.LineStippleEnable      = 0;   // Disable.
drawCmd.AreaStippleEnable      = 0;   // Disable.
```

```
    for (v = 0; v < verticesInMesh; v++)
    {
        vertexStore = __Delta_V0FloatTag + 16 * (v % 3);

        LD_PARAM((vertexStore + paramR), mesh[v].r);
        LD_PARAM((vertexStore + paramG), mesh[v].g);
        LD_PARAM((vertexStore + paramB), mesh[v].b);
        LD_PARAM((vertexStore + paramA), mesh[v].a);
        LD_PARAM((vertexStore + paramX), mesh[v].x);
        LD_PARAM((vertexStore + paramY), mesh[v].y);
        LD_PARAM((vertexStore + paramZ), mesh[v].z);

        if (v >= 2)
        {
            // We now have enough vertices to draw a triangle.
            for (rect = 0; rect < numberClipRectangles; rect++)
            {
                // Load in the scissor rectangle.
                LD_REG(ScissorMinXYTag, (clipRectangles[rect].scissorMin.y <<
16 |
                            clipRectangles[rect].scissorMin.x));
                LD_REG(ScissorMaxXYTag, (clipRectangles[rect].scissorMax.y <<
16 |
                            clipRectangles[rect].scissorMax.x));

                if (rect == 0)
                {
                    LD_REG(DrawTriangleTag,  *((long *) &drawCmd));
                }
                else
                {
                    LD_REG(RepeatTriangleTag, 0);       // data field not used.
                }
            }
        }
    }
}
```

21.3 Interpolation Calculation

21.3.1 Color Gradient Interpolation

To draw from left to right, top to bottom, the color gradients (or deltas) required are:

And from the plane equation:

where, to be independent of the order the vertices are provided:

These values allow the color of each fragment in the triangle to be determined by linear interpolation. For example, to calculate the red component color value of a fragment at Xn,Ym:

- add dRdy, for each scanline between Y1 and Yn, to R1, then
- add dRdx for each fragment along scanline Yn from the left edge to Xn.

The example chosen has the 'knee' i.e. vertex 2, on the right hand side, and drawing is from left to right. If the knee were on the left side (or drawing was from right to left), then the Y deltas for both the subordinate sides would be needed to interpolate the start values for each color component (and the depth value) on each scanline. For this reason GLINT R4 always draws triangles starting from the dominant edge and towards the subordinate edges. For the example triangle, this means left to right.

21.3.2 Register Set Up for Color Interpolation

For the example triangle, GLINT R4 registers must be set as follows for color interpolation. Note color values are in 24 bit, fixed point 2's complement 9.15 format.

```
// Load the color start and delta  values to draw
// a triangle
```

Rstart (R_1)

Gstart (G_1)

Bstart (B_1)

dRdyDom ($dRdy_{13}$) // To walk up the dominant edge

dGdyDom ($dGdy_{13}$)

dBdyDom ($dBdy_{13}$)

dRdx (dRdx) // To walk along the scanline

dGdx (dGdx)

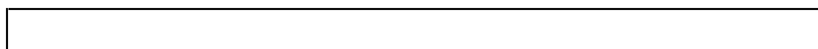
dBdx (dBdx)

21.3.3 Calculating Depth Gradient Values

To draw from left to right and top to bottom, the depth gradients (or deltas) required for interpolation are:

And from the plane equation:

where, as before:



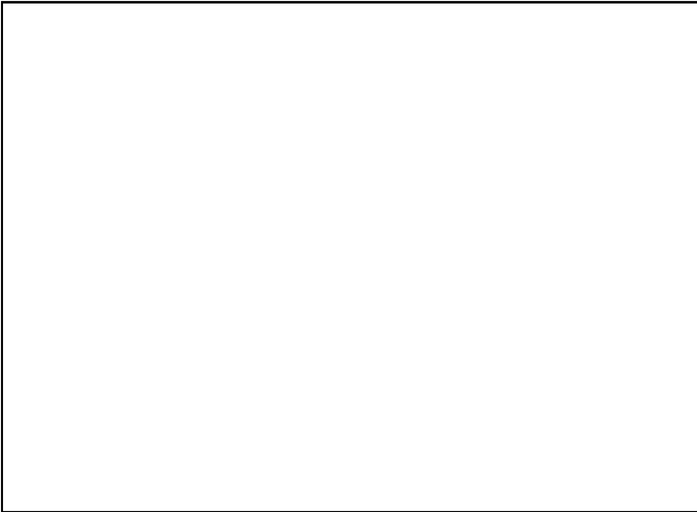
The divisor, shown here as c, is the same as for color gradient values. The two deltas, dZdy13 and dZdx allow the Z value of each fragment in the triangle to be determined by linear interpolation as was described for the color interpolation above.

returned by the *CFGDeviceId* register in the is 0006h in bits 31-16.

21.4 Appendix F. Accurate Rendering

This appendix describes how to calculate the various parameters needed to define a Gouraud shaded triangle. This topic is covered in section 1.1.2, however in the interest of simplicity some of the finer details were glossed over. The quality of the rasterization and shading suffers where these fine details are not included and will give rise to 'stitch marks' and 'bright edge' artifacts. The main area where simplifications were made earlier relates to the fact that vertices are not, in general, coincident with pixel centers so sub pixel corrections are necessary. The initial values being interpolated (RGB for example) need to be adjusted to account for this. GLINT R4 does the necessary X corrections when moving from scan line to scan line when the SubPixelCorrection bit is set, but the initial Y correction must be done in software.

Consider a sample triangle, highly magnified to emphasize the sub pixel corrections needed:



The vertices are sorted into Y order and the dominant edge is AC. Scan conversion will start at vertex A and proceed upwards. The origin is bottom left.

The usual parameters to interpolate (denoted P in the diagram) across the triangle would include color (R, G, B and alpha), depth (Z), fog (F), and texture (S, T, Q, Ks and Kd). The source code to set up GLINT R4 to achieve the best quality rendering will only calculate the parameters for RGBA and Z to keep the size of the code down.

```
#include <stdio.h>
#include <float.h>

// A simple macro which just prints out the register name and value.
// Replace this with some code to write to GLINT R4.

#define LD_GLINT R4_REG(name, value)                    \
    printf ("%s = %08x\n", #name, value)



// This software is part of the application note which describes
// how GLINT R4 is set up to get the best quality rendering. Particular
// care is taken to avoid cracks, stitch marks and bright edge artifacts
// from occurring. The OpenGL rasterization rules are used.
// The software has not been written with maximum performance in mind,
// but as a clear, well documented example covering the nuances
// which are easily overlooked.

// Simple vertex structure used to interface parameters to the RenderTriangle
// function.

typedef struct {    float   x, y, z;        // in device coords
                    float   r, g, b, a;     // in the range 0.0 to 1.0
                } Vertex;



// Prototypes.

long IntToFixedPoint16 (long i);
long FloatToColor (float f);
long FloatToCoordinate (float f);
void FloatToDepth (float f, long *zi, long *zf);
void RenderTriangle (Vertex *v0, Vertex *v1, Vertex *v2);



// Defines some simple function to convert from floating point numbers
```

```
// to various fixed point formats. These can be inlined if necessary.

long IntToFixedPoint16 (long i)
{
   return i << 16;
}

// These functions perform the conversion from floating point numbers
// to the various fixed point format numbers required in GLINT R4.  They
// are implemented as simple operations on the binary representation
// of IEEE single precision floating point number so the floating
// point rounding mode doesn't need to be set up first and in many
// cases they are faster than using the built in conversion functions,
// especially when the range checking and clamping is taken into account.

//  Format of IEEE single-precision (32-bit) real number.

#define F_BIAS      127
#define F_SIGN_BIT  31
#define F_EXPONENT_BITS 23
#define F_FRACTION_BITS 0



// Convert 32-bit floating-point value to 9.15 fixed-point value used
// for the color parameters.  The input range is assumed to be 0.0
// to 1.0.  The algorithm is:
// If exponent < -15 then return (0x00000000), otherwise
// if exponent < 8 then return (-1**(s) * 1.f * 2**(e - 127)), otherwise
// return ((s == 1) ? 0xff800000 : 0x007fffff).

long FloatToColor (float fi)
{
   long          f = *((long *) &fi);
   long          sign;
   unsigned char   exponent;

   sign = (f >> F_SIGN_BIT);
```

```
    exponent = (unsigned char)(f >> F_EXPONENT_BITS);
    if (exponent < (F_BIAS-15))
        return (0);
    if (exponent < (F_BIAS+8))
    {
        f = ((unsigned long)((f | 0x00800000) << 8)
                                    >> ((F_BIAS+16) - exponent));
        if (sign < 0)
            f = -f;
        return (f);
    }
    return (0x007fffff ^ sign);
}



// Convert 32-bit floating-point value to 16.16 fixed-point value used
// for the rasterizer parameters.
// If exponent < 0 then return (0x00000000), otherwise
// if exponent < 31 then return (-1**(s) * 1.f * 2**(e - 127)), otherwise
// return ((s == 1) ? 0x80000000 : 0x7fffffff).

long FloatToCoordinate (float fi)
{
    long            f = *((long *) &fi);
    long            sign;
    unsigned char   exponent;
    long            res;

    sign = f >> F_SIGN_BIT;
    exponent = (unsigned char) (f >> F_EXPONENT_BITS);
    if (exponent < (F_BIAS-16))
        return (0);
    if (exponent < (F_BIAS+15))
    {
        res = ((unsigned long)((f | 0x00800000) << 8)
                                    >> ((F_BIAS+15) - exponent));
```

```
        if (sign < 0)
            res = -res;
        return (res);
    }
    return (0x7fffffff ^ sign);
}


// Convert 32-bit floating-point value to 24.16 fixed-point value as
// used by the Z values.  Note that this assumes a 24 bit Z buffer.
// If exponent < -16 then return (0x0000000000000000), otherwise
// if CLAMP_24_16 is defined and is non-zero:
// if exponent < 23 then return (-1**(s) * 1.f * 2**(e - 127)), otherwise
// return ((s == 1) ? 0xff80000000000000 : 0x007fffffffff0000).
// otherwise:
// return (-1**(s) * 1.f * 2**(e - 127)).

void FloatToDepth (float fi, long *zi, long *zf)
{
    long            f = *((long *) &fi);
    long            sign;
    unsigned char   exponent;
    long            resh;
    unsigned long   resl;

    sign = (f >> F_SIGN_BIT);
    exponent = (unsigned char)(f >> F_EXPONENT_BITS);
    if (exponent < (F_BIAS-16))
    {
        *zi = 0;
        *zf = 0;
        return;
    }
    if (exponent < (F_BIAS+23))
    {
        f = ((f | 0x00800000) << 8);
        if (exponent < (F_BIAS+0))
```

```
        {
            resh = 0;
            resl = ((unsigned long) f >> ((F_BIAS-1) - exponent));
        }
        else
        {
            unsigned char shift;

            shift = ((F_BIAS+31) - exponent);   // 8 <= shift < 32
            resh = ((unsigned long) f >> shift);
            resl = (f << (31 - shift));         // shifts >= 32 undefined
            resl <<= 1;                         // so we must shift twice
        }
        if (sign < 0)
        {
            unsigned long old_resl;

            resl = ~resl;
            resh = ~resh;
            old_resl = resl;
            resl += 0x00010000;
            if (resl < old_resl)        // overflow
                ++resh;
        }
    }
    else
    {
        resh = (0x007fffff ^ sign);
        resl = (0xffff0000 ^ sign);
    }
    resl &= 0xffff0000;
    *zi = resh;
    *zf = resl;
}
```

```
#define SAME 0
#define REVERSED ~SAME
#define ORDER(v0, v1, v2, order) {a = v0; b = v1; c = v2; windingOrder = order;}

void RenderTriangle (Vertex *v0, Vertex *v1, Vertex *v2)
{
   float   dxAB, dyAB, dxBC, dyBC, dxAC, dyAC; // Diff in x,y for each edge.
   float   drAC, dgAC, dbAC, daAC, dzAC;      // Diff in rgbz for dominant edge
   float   drBC, dgBC, dbBC, daBC, dzBC;      // Diff in rgbz for the BC edge.
   float   dxdyAC, dxdyAB, dxdyBC;            // Edge gradients for unit
                                               // set in y
   float   drdxdy, dgdxdy, dbdxdy;
   float   dadxdy, dzdxdy;
   float   drdx, dgdx, dbdx, dadx, dzdx;      // Gradients for unit step in x.
   float   r0, g0, b0, a0, z0;                // Start values
   float   area, oneOverArea, t1, t2;
   float   oneOverdyAC;
   Vertex  *a, *b, *c;                        // Sorted vertices.
   long    xDomFixed, xSubFixed;
   float   dyErr, yBottom, yTop;
   long    iyBottom, iyTop;
   int     windingOrder;                      // Not used.
   long    zi, zf;
   long    temp;

   // Sort vertices into ascending Y order. *a points to the vertex with the
   // lowest y value. Compare winding order of the pre and post sorted vertices
   // and set winding order flag as appropriate (this is only needed if culling
   // based on the winding order is to be done).

   if (v0->y < v1->y)
   {
      if (v1->y < v2->y)
         ORDER (v0, v1, v2, SAME)
      else
         if (v0->y < v2->y)
            ORDER (v0, v2, v1, REVERSED)
```

```
        else
            ORDER (v2, v0, v1, SAME)
}
else
{
    if (v1->y < v2->y)
    {
        if (v0->y < v2->y)
            ORDER (v1, v0, v2, REVERSED)
        else
            ORDER (v1, v2, v0, SAME)
    }
    else
        ORDER (v2, v1, v0, REVERSED)
}

// Compute signed area of the triangle.
// Form vectors for two edges of the triangle.
dxAC = a->x - c->x;
dxBC = b->x - c->x;
dyAC = a->y - c->y;
dyBC = b->y - c->y;

// Form the cross product of the two edges.
area = dxAC * dyBC - dxBC * dyAC;

if (area == 0.0)
    return;             // Reject zero area triangles.

// A negative area just means the order of the vertices, after sorting, was
// clockwise. Note this may be different from original input order.
if (area < 0.0)
    area = -area;           // Make positive.

// The dx/dy value (change in x for unit change in y) are needed for
// each edge so the rasterizer can compute the new left and right hand
// x coordinates as it steps from one scan line to the next. Horizontal
```

```
// or near horizontal edges will have very large gradients but these will
// be handled later. Values for AC and BC have already been calculated so
// just do the remaining edge.

dxAB = a->x - b->x;
dyAB = a->y - b->y;

// The dominant edge is always AC (i.e. the edge with the maximum Y extent).
// Compute the change in rgbaz along this edge for unit change in y.
oneOverdyAC = 1.0 / dyAC;

// Differences along edge AC
drAC = a->r - c->r;
dgAC = a->g - c->g;
dbAC = a->b - c->b;
daAC = a->a - c->a;
dzAC = a->z - c->z;

// Gradient along edge AC for each parameter.
drdxdy = drAC * oneOverdyAC;
dgdxdy = dgAC * oneOverdyAC;
dbdxdy = dbAC * oneOverdyAC;
dadxdy = daAC * oneOverdyAC;
dzdxdy = dzAC * oneOverdyAC;
dxdyAC = dxAC * oneOverdyAC;

// Difference along edge BC
drBC = b->r - c->r;
dgBC = b->g - c->g;
dbBC = b->b - c->b;
daBC = b->a - c->a;
dzBC = b->z - c->z;

// Compute the change in rgbaz when taking unit steps in x.
oneOverArea = 1.0 / area;

t1 = dyAC * oneOverArea;
```

```
t2 = dyBC * oneOverArea;

drdx = drAC * t2 - drBC * t1;
dgdx = dgAC * t2 - dgBC * t1;
dbdx = dbAC * t2 - dbBC * t1;
dadx = daAC * t2 - daBC * t1;
dzdx = dzAC * t2 - dzBC * t1;

// A general triangle will need to be split into two trapezoids for
// rendering. Either of these trapezoids may have a zero height in
// which case the triangle has a flat top or bottom.  The rasterizer
// and DDAs are still set up, however the count may be zero.

// Fill lower trapezoid.
yBottom = a->y;
yTop = b->y;

// The y coordinates are converted to integer values, taking into
// account the openGL rules which determine which pixels fall within
// the boundary.

temp = FloatToCoordinate (yBottom);     // float to 16.16 fixed point
temp += 0x00007fff;                     // add in nearly a half
iyBottom = temp >> 16;                  // extract integer part

temp = (int) FloatToCoordinate (yTop);  // float to 16.16 fixed point
temp += 0x00007fff;                     // add in nearly a half
iyTop = temp >> 16;                     // extract integer part

dyErr = iyBottom + 0.5 - yBottom;

// Check for the case when AB is a true horizontal edge to prevent a divide
// by zero.
if (dyAB == 0.0)
   dyAB = FLT_MIN;          // set to a very small number.

dxdyAB = dxAB / dyAB;
```

```
// Move the rgbaz values at vertex a along the edge AC in proportion
// to how far the vertex a is from the pixel center in the y direction
// to do the sub pixel adjustment in Y.  GLINT R4 does the sub pixel
// adjustment in X automatically, if enabled.

r0 = a->r + dyErr * drdxdy;
g0 = a->g + dyErr * dgdxdy;
b0 = a->b + dyErr * dbdxdy;
a0 = a->a + dyErr * dadxdy;
z0 = a->z + dyErr * dzdxdy;

// Similarly for the start values for the left and right hand edges.
xDomFixed = FloatToCoordinate (a->x + dyErr * dxdyAC);
xSubFixed = FloatToCoordinate (a->x + dyErr * dxdyAB);

// Load up GLINT R4 with the parameters.

// Rasterizer. Note that the RasterizerMode is set to add
// __GLINT R4_START_BIAS_ALMOST_HALF to the XDom, XSub and
// Y Start values to conform to the OpenGL rasterization rules.

LD_GLINT R4_REG(StartXDom, xDomFixed);
LD_GLINT R4_REG(dXDom.     FloatToCoordinate (dxdyAC));
LD_GLINT R4_REG(StartXSub, xSubFixed);
LD_GLINT R4_REG(dXSub,     FloatToCoordinate (dxdyAB));
LD_GLINT R4_REG(StartY,    IntToFixedPoint16 (iyBottom));
LD_GLINT R4_REG(dy,        IntToFixedPoint16 (1));
LD_GLINT R4_REG(Count,     (iyTop - iyBottom));

// Color DDA.
LD_GLINT R4_REG(Rstart,    FloatToColor (r0));
LD_GLINT R4_REG(dRdx,      FloatToColor (drdx));
LD_GLINT R4_REG(dRdyDom,   FloatToColor (drdxdy));
LD_GLINT R4_REG(Gstart,    FloatToColor (g0));
LD_GLINT R4_REG(dGdx,      FloatToColor (dgdx));
LD_GLINT R4_REG(dGdyDom,   FloatToColor (dgdxdy));
```

```
LD_GLINT R4_REG(Bstart,    FloatToColor (b0));
LD_GLINT R4_REG(dBdx,      FloatToColor (dbdx));
LD_GLINT R4_REG(dBdyDom,   FloatToColor (dbdxdy));
LD_GLINT R4_REG(AStart,    FloatToColor (a0));
LD_GLINT R4_REG(dAdx,      FloatToColor (dadx));
LD_GLINT R4_REG(dAdyDom,   FloatToColor (dadxdy));

// Depth DDA.
FloatToDepth (z0, &zi, &zf);
LD_GLINT R4_REG(ZStartU,   zi);
LD_GLINT R4_REG(ZStartL,   zf);

FloatToDepth (dzdx, &zi, &zf);
LD_GLINT R4_REG(dZdxU,     zi);
LD_GLINT R4_REG(dZdxL,     zf);

FloatToDepth (dzdxdy, &zi, &zf);
LD_GLINT R4_REG(dZdyDomU,  zi);
LD_GLINT R4_REG(dZdyDomL,  zf);

// Render the trapezoid ...
LD_GLINT R4_REG(Render,    0x00014041);

// Fill upper trapezoid.
yBottom = b->y;
yTop = c->y;

// The y coordinates are converted to integer values, taking into
// account the openGL rules which determine which pixels fall within
// the boundary.

temp = FloatToCoordinate (yBottom); // float to 16.16 fixed point
temp += 0x00007fff;                 // add in nearly a half
iyBottom = temp >> 16;              // extract integer part

temp = FloatToCoordinate (yTop);    // float to 16.16 fixed point
temp += 0x00007fff;                 // add in nearly a half
```

```
    iyTop = temp >> 16;                 // extract integer part

    // Find the dyErr value for vertex B so that the start value for x can be
    // corrected.
    dyErr = iyBottom + 0.5 - yBottom;

    // Check for the case when BC is a true horizontal edge to prevent a divide
    // by zero.
    if (dyBC == 0.0)
        dyBC = FLT_MIN;          // set to a very small number.

    dxdyBC = (dxBC / dyBC);

    // Set up the rasterizer for the upper trapezoid.  All other DDA units
    // can carry on with their parameters as they are walking up the same
    // edge.
    xSubFixed = FloatToCoordinate (b->x + dyErr * dxdyBC);
    LD_GLINT R4_REG(StartXSub, xSubFixed);
    LD_GLINT R4_REG(dxSub, FloatToCoordinate (dxdyBC));
    LD_GLINT R4_REG(ContinueNewSub, (iyTop - iyBottom));
}
```

21.5 Glossary

accumulation buffer A color buffer of higher resolution than the displayed buffer (typically 16bits per component for an 8bit per component display). Typically used to sum the result of rendering several frames from slightly different viewpoints to achieve motion blur effects or eliminate aliasing effects.

active fragment A fragment which passes all the various culling tests, such as scissor, depth(Z), alpha, etc., is written to/combined with the corresponding pixel in the framebuffer. See also "fragment" and "passive fragment".

aliasing A phenomena resulting from a rendering style which ignores the fact that a pixel may not be wholly covered by a primitive, leading to jagged edges on primitives.

alpha buffer A memory buffer containing the fourth component of a pixel's color in addition to Red, Green and Blue. This component is not displayed, but may be used for instance to control color blending and antialiasing.

alpha test A test used to cull selected fragments from being drawn, based on a comparison of a fixed value with the alpha value of the fragment.

antialiasing A rendering style which weights the color of a pixel by the fraction of its area that is covered by primitives, leading to reduction or elimination of jagged edges.

bitblt Bit aligned block transfer. Copy of a rectangular array of pixels in a bitmap from one location to another.

block write A feature provided in some SGRAM devices which allows multiple pixels to be set to a given value by a single write. See also fast fill which is an alternative name for the same feature.

command register A register which when loaded triggers activity in GLINT R4. For instance the **Render** command register when loaded will cause GLINT R4 to start rendering the specified primitive with the parameters currently set up in the control registers.

context The state information associated with a particular task. Typically in a system more than one task will be using GLINT R4 to render primitives. Software on the host must save away the current contents of the GLINT R4 control registers when suspending one task to allow another to run, and must restore the state when that task is next scheduled to run.

control register A register which contains state that dictates how **GLINT R4** will execute a command.

culling The process of eliminating a fragment, object face, or primitive, so that it is not drawn.

DDA Digital Differential Analyzer. An algorithm for determining the pixels to draw along a line or polygon edge. Also used to interpolate linearly varying values such as color and depth.

depth (Z) buffer A memory buffer containing the depth component of a pixel. Used to, for example, eliminate hidden surfaces.

depth-cueing A technique which determines the color of a pixel based on its depth. Used, for instance, to fade far away objects into the background. See also fogging.

dithering A rendering style which increases the perceived range of displayed colors at the cost of spatial resolution. The technique is similar to the use of stippled patterns of black and white pixels, to achieve shades of grey on a black and white display.

double-buffering A technique for achieving smooth animation, by rendering only to an undisplayed back buffer, and then swapping the back buffer to the front once drawing is complete.

fast fill A feature provided in SGRAM devices which allows multiple pixels to be set to a given value by a single write. See also block write which is an alternative name for the same feature.

fogging A technique which determines the color of a pixel based on its depth. Used, for instance, to fade far away objects into the background. See also depth-cueing.

Fast Clear Planes Used to allow higher animation rates by enabling localbuffer pixel data, such as depth (Z), to be cleared down - not required or supported in GLINT R4

fragment A fragment is an object generated as a result of the rasterization of a primitive. It corresponds to and contains all the components of a single pixel. If a fragment passes all the various culling tests, such as scissor, depth(Z), alpha, etc., it will be written to/combined with the corresponding pixel in the framebuffer.

framebuffer An area of memory containing the displayable color buffers (front, back, left, right, overlay, underlay), their (optional) associated alpha components, and any associated (optional) window control information. This memory is typically separate from the localbuffer.

Graphic ID (GID) A component of a pixel containing information used for per pixel clipping.

host The processor which controls GLINT R4.

localbuffer An area of memory which may be used to store the following non-displayable pixel information: depth(Z), stencil, Graphic ID.

passive fragment A fragment which fails one or more of the various culling tests, such as scissor, depth(Z), alpha, etc., is nor written to/combined with the corresponding pixel in the framebuffer. See also "fragment" and "active fragment".

pixel Picture element. A pixel comprises the bits in all the buffers (whether stored in the localbuffer or framebuffer), corresponding to a particular location in the framebuffer.

primitive A geometric object to be rendered. The GLINT R4 primitives are points, lines, trapezoids (including triangles as a subset), and bitmaps.

rasterization The act of converting a point, line, polygon, or bitmap, in device coordinates, into fragments.

rendering Conversion of primitives in object coordinates into an image.

scissor test A means of culling fragments which lie outside the defined scissor rectangle. The scissor rectangle is defined in device coordinates.

stencil buffer A buffer used to store information about a pixel which controls how subsequent stenciled fragments at the same location may be combined with its current value. Typically used to mask complex two-dimensional shapes.

stipple A one or two dimensional binary pattern which is used to cull fragments from being drawn.

task A process, or thread on the host which uses the GLINT R4 coprocessor. Typically tasks assume that they have sole use of GLINT R4 and rely on a device driver to save and restore their GLINT R4 context, when they are swapped out.

texel Texture element. An element of an image stored in texture memory which represents the color of the texture to be applied (fully or in part) to a corresponding fragment.

texture An image used to modify the color of fragments during processing. Often used for instance to achieve high realism in a scene, with relatively few primitives.

texture mapping The process of applying a two dimensional image to a primitive. For instance to apply a wood grain effect to a table.

window control buffer

A buffer containing control bits used by display hardware to select between multiple hardware LUTs or display buffers (such as overlay and underlay) on a per pixel basis. Usually a given value in the buffer corresponds to a single window on the screen.

writemask A bit pattern used to enable or inhibit the writing of the corresponding bits of a fragment's color into the framebuffer.

22 Volume III Index

Alpha Blend, 15-2, **15-4**
Alpha Blend Example, 15-11
Alpha Blend Unit, 15-1
Alpha Blending, 15-2, **15-4**
alpha buffer, 15-3, **16-5**, 21-34
Alpha test, 13-7
Alpha Test, 13-7
Alpha Test, 13-7
AlphaBlendMode, **15-3**, **15-4**, 15-5, 15-8
AlphaTestMode, **13-7**, **13-8**
antialiasing, **21-2**
Application Initialization, 19-8
block write, **21-2**
chroma, 15-9
ChromaLower, **15-9**
ChromaTestMode, **15-9**
ChromaUpper, **15-9**
CI Fogging Equation, 13-4
Color Format, 19-6
Color Format Example
  3:3:2, 16-6
  8:8:8:8, 16-6
Color Format Unit, 16-1
Color Index Format Example, 16-7
Color Interpolation, 21-20
command register, **21-2**
context, **21-2**
control register, **21-2**
delta, 21-20, 21-21
depth (Z) buffer, **21-2**
Depth Gradient, 21-21
depth-cueing, **21-2**
dFdx, **13-2**
dFdyDom, **13-2**
Disabling Specialized Modes, 19-6
Disabling units not in use, 20-3
Dither Example, 16-6
dithering, **21-2**
Dithering, 16-4
DitherMode, **16-4**, **16-5**, **16-6**
DMA
  Using the Bus Mastership, 20-3
dRdx, 21-20
dY, 20-4
Enabling Writing, 19-7
extent checking, 18-2
Extent Checking, 18-2
Fast double buffering in a window, 20-2
fast fill, **21-3**
FBColor, 14-1, **18-1**
FBData, 15-5
FBDestReadMode, 16-8
FBHardwareWriteMask, 17-1
FBReadMode, **14-1**, 14-2, 19-7
FBSoftwareWriteMask, **17-1**
FBWriteData, **16-7**
FBWriteMode, **15-5**
Filter Mode Example, 18-1
Filtering, 18-1
FilterMode, **18-1**, 18-2, 18-3, **18-4**
Fog, 13-1
Fog Example, 13-5
Fog Index Calculation - The Fog DDA, 13-1
fogging, 21-3
FogMode, **13-4**, **13-5**
framebuffer, **21-3**
Framebuffer, 19-4
  Bypass, 20-4
Framebuffer, 14-1
Framebuffer Depth, 19-3

Framebuffer Read Span Operations, 14-2
Graphic ID, **21-3**
Hardware Writemask Example, 17-2
Hardware Writemasks, 17-1
High Speed Flat Shaded Rendering, 16-7
Host, 18-1, **19-5**
Image Formatting, 15-4
Improving PCI bus bandwidth for Programmed I/O and DMA, 20-2
Initialization, 19-1
Initializing GLINT, 19-1
Interpolation
  Calculating Color values, 21-20
LBReadFormat, **19-4**
LBWriteFormat, **19-4**
Loading registers in unit order, 20-4
localbuffer, **21-3**
Localbuffer, 19-4
  Bypass, 20-4
Logical Op, 16-7
Logical Op and Software Writemask Example, 16-10
Logical Operations, 16-8
LogicalOpMode, **16-7**, 16-8
MaxHitRegion, 18-1, 18-3, **18-5**
MaxRegion, **18-2**, 18-3, **18-5**
Memory Configuration, 19-2
Merge-copy Span Operations, 14-2
MinHitRegion, 18-1, 18-3, **18-5**
MinRegion, **18-2**, 18-3, **18-5**
Miscellaneous Generic Graphics Tips, 20-5
origin
  window, 19-6
Origin
  Setting, 19-6
PCI burst transfers under Programmed I/O, 20-2
PCI bus, 19-2
PCI Disconnect Under Programmed I/O, 20-3
Performance Tips, 20-1
picking, 18-2
Picking Example, 18-5
PickResult, 18-1, **18-2**, **18-5**
PixelSize, **19-3**
primitive, **21-3**
pseudocode, 21-1
Rapid clear of the localbuffer & framebuffer, 20-3
Register Updates
  Avoiding Unnecessary, 20-4
ResetPickResult, **18-2**, **18-5**
RGBA Fogging Equation, 13-3
scissor test, 18-2, 18-3
Screen Clipping Region, 19-4
Screen Width, 19-4
SGRAM Block Writes, 20-1
Software Writemask Example, 17-1
Software Writemasks, 17-1
Standard Framebuffer Read Operation, 14-1
StartXDom, 21-2
Statistic Operations, 18-2
StatisticMode, **18-2**, **18-4**, **18-5**
stencil buffer, **21-4**
stipple, **21-4**
Sync, **18-3**, 18-6, **19-5**
Sync Interrupt Example, 18-6
Synchronization, 18-3
System Initialization, 19-2
texture, **21-4**
texture mapping, **21-4**
UseConstantFBWriteData, *16-7*
Video Timing, 19-3
Window Address
  Setting, 19-6
window control, **21-4**
Window Initialization, 19-6
Write Masks, 17-1
writemask, **21-4**
Writemasks, 19-7
XOR Example, 16-9