

GLINT R4[®]

Programmer's Guide - Volume II

DRAFT ONLY

**PROPRIETARY AND CONFIDENTIAL
INFORMATION**



3D*labs*[®]

GLINT R4[®]

Programmer's Guide - Volume II

**PROPRIETARY AND CONFIDENTIAL
INFORMATION**

Issue 3

Proprietary Notice

The material in this document is the intellectual property of **3Dlabs**. 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, **3Dlabs** 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. **3Dlabs** may not produce printed versions of each issue of this document. The latest version will be available from the **3Dlabs** web site.

3Dlabs 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.

3Dlabs is the worldwide trading name of **3Dlabs** Inc. Ltd.

3Dlabs, GLINT R4 and GLINT R4 are registered trademarks of **3Dlabs** 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 **3Dlabs** Inc. Ltd. 1999. All rights reserved worldwide.

Email: info@3dlabs.com

Web: <http://www.3dlabs.com>

3Dlabs Ltd.

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

3Dlabs K.K.

Shiroyama JT Mori Bldg 16F
40301 Toranomom
Minato-ku, Tokyo, 105, Japan
Tel: +81-3-5403-4653
Fax: +91-3-5403-4646

3Dlabs Inc.

480 Potrero Avenue
Sunnyvale, CA 94086,
United States
Tel: (408) 530-4700
Fax: (408) 530-4701

Change History

Document	Issue	Date	Change
160.4.2	1	1 Dec 99	First DRAFT Issue.
160.4.2	2	25 January 2000	Extensive updates
160.4.2	3	18 June 2001	Clarified OpaqueSpan bit; fixed Initialization example values for stencil position and width, GID; removed index entries and traces of FBReadMode , deleted Windowbase references, corrected GID test control (no longer in Windows reg), Stencil source data field.

Contents

Proprietary Notice.....	i
Change History	ii
Contents	iii
GRAPHICS PROGRAMMING	7-1
7.1 The Graphics Pipeline	7-1
7.1.1 Router.....	7-3
7.1.2 Initialization.....	7-3
7.1.3 Dominant and Subordinate Sides of a Triangle	7-3
7.1.4 Register Set Up for Depth Testing	7-4
7.1.5 Subpixel Correction	7-4
7.2 Pipeline Overviews.....	7-5
7.2.1 A day in the life of a 3D triangle.....	7-5
7.2.2 A day in the life of a 2D primitive.....	7-10
RASTERIZER AND 2D SETUP	8-1
8.1 Description	8-1
8.1.1 Trapezoids.....	8-2
8.2 Antialiasing.....	8-4
8.2.1 Antialias Application	8-4
8.2.2 Polygon Antialiasing Considerations	8-5
8.2.3 Registers.....	8-6
8.2.4 Antialias Example.....	8-6
8.2.5 Antialiasing Primitive Types	8-6
8.2.6 Points	8-9
8.2.7 Lines	8-12
8.2.8 Polygons.....	8-15
8.3 Span Operations	8-18
8.3.1 Mode changes in Span Operations	8-19
8.3.2 Alpha Filtering.....	8-20
8.3.3 Span Mask Processing.....	8-20
8.3.4 Block Write.....	8-21
8.3.5 Pixel Sizes.....	8-21
8.3.6 Bitmaps, Spans and Images	8-22
8.4 Rasterizer Mode	8-32

8.4.2	<i>Multi-rasterizer Operation</i>	8-33
8.4.3	<i>Rasterizer Unit Registers</i>	8-33
8.4.4	<i>Render Command</i>	8-36
8.5	2D Setup	8-42
8.5.1	<i>Glyph rendering</i>	8-42

SCISSOR, STIPPLE AND COLOR DDA UNITS 9-1

9.1	Scissor Unit.....	9-1
9.1.1	<i>User Scissor Test</i>	9-1
9.1.2	<i>Screen Scissor Tests</i>	9-1
9.1.3	<i>Scissor Registers</i>	9-2
9.1.4	<i>Span Operations and the Scissor Unit</i>	9-3
9.1.5	<i>Scissor Example</i>	9-3
9.2	Stipple Unit.....	9-3
9.2.1	<i>Area Stippling</i>	9-4
9.2.2	<i>Line Stippling</i>	9-5
9.2.3	<i>Span Operations and Stippling</i>	9-5
9.2.4	<i>Registers</i>	9-5
9.2.5	<i>Examples</i>	9-7
9.2.6	<i>Line Stipple Example</i>	9-8
9.2.7	<i>Area Stipple Pattern Example</i>	9-8
9.3	Color DDA Unit.....	9-9
9.3.1	<i>RGBA and Color-Index(CI) Modes</i>	9-10
9.3.2	<i>Gouraud Shading</i>	9-11
9.3.3	<i>Flat Shading Example</i>	9-12
9.3.4	<i>Gouraud Shaded Trapezoid Example</i>	9-12
9.3.5	<i>Gouraud Shaded Line Example</i>	9-13

LOCALBUFFER READ/WRITE..... 10-1

10.1.1	<i>Mode Registers</i>	10-2
10.2	Window register.....	10-4
10.3	Pixel Ownership (GID) Test Unit.....	10-4
10.3.1	<i>Pixel Ownership Test</i>	10-4
10.4	Stencil Test.....	10-7
10.4.1	<i>Registers</i>	10-8
10.4.2	<i>Stencil Example</i>	10-10
10.5	Depth Test.....	10-11

10.5.1	<i>Registers</i>	10-13
10.5.2	<i>Depth Example</i>	10-15
	TEXTURE MAPPING	11-1
11.1.1	<i>Compatibility with Earlier Chipsets</i>	11-1
11.2	Texture Co-ordinate Generation	11-2
11.2.1	<i>Calculate texture coordinates</i>	11-2
11.2.2	<i>Level of Detail calculation</i>	11-3
11.2.3	<i>Texture Read</i>	11-6
11.2.4	<i>Filter Modes</i>	11-8
11.2.5	<i>Texel Formatting</i>	11-12
11.2.6	<i>Lookup Table (LUT)</i>	11-16
11.2.7	<i>Texture Filtering and Alpha Mapping</i>	11-17
11.2.8	<i>Texture Color Compositing</i>	11-18
11.2.9	<i>Implementation</i>	11-29
	VOLUME II INDEX	33

7

Graphics Programming

GLINT R4 provides a rich variety of operations for 2D and 3D graphics supported by its pipeline architecture. Primarily intended as a rasterizer, the R4 also has an onboard Delta setup unit for standalone operation.

In this chapter, section §7.1 shows the basic units in the Pipeline. The following chapters describe a typical rendering process for a Gouraud shaded triangle and looks at each of the units in detail.

7.1 The Graphics Pipeline

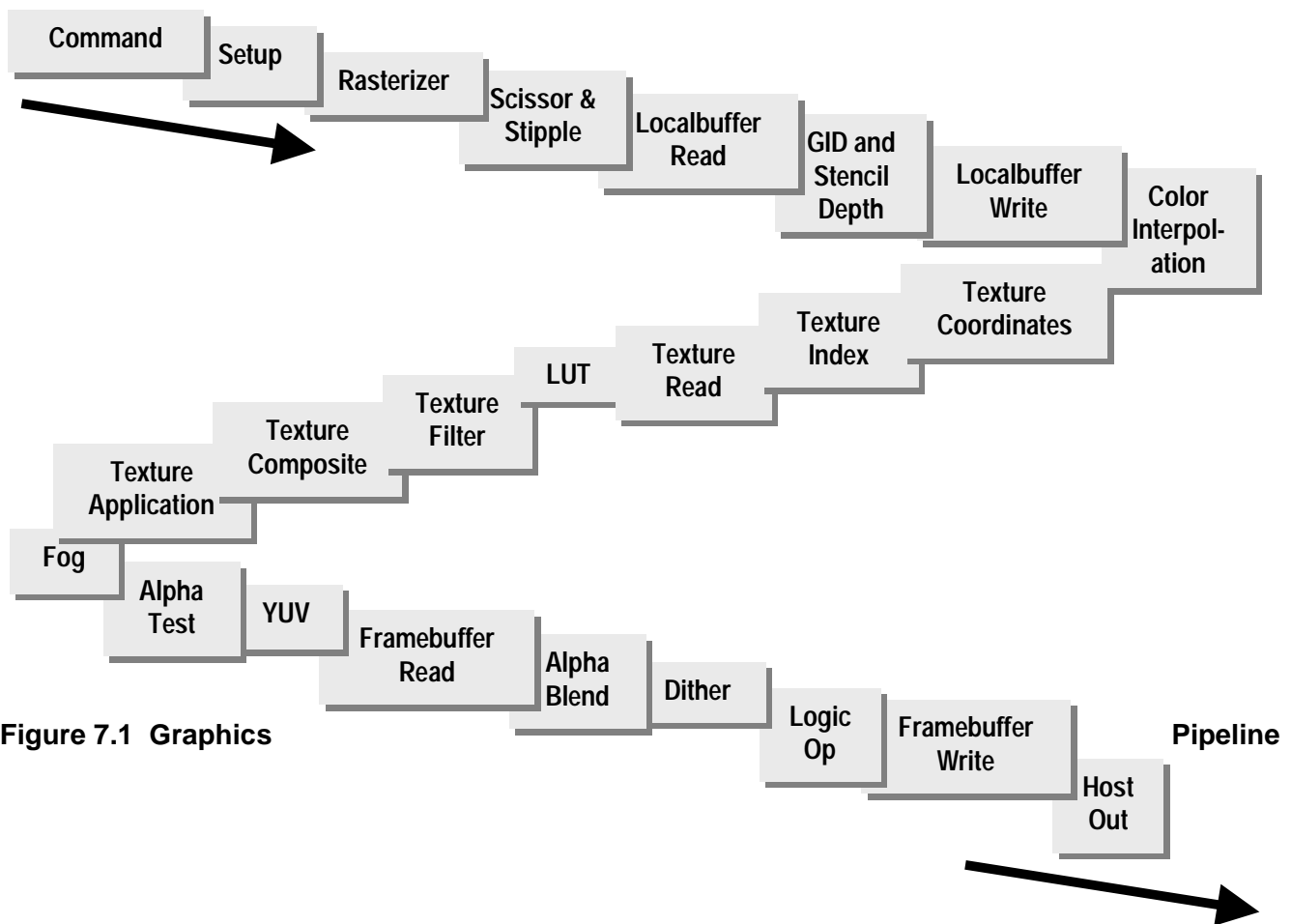


Figure 7.1 Graphics

Figure 7.1 shows a schematic of the pipeline. In this diagram, the localbuffer contains¹ the pixel ownership values (known as Graphic IDs), the Depth (Z) and Stencil buffer. The framebuffer contains the Red, Green, Blue and Alpha bitplanes. The units in the HyperPipeline are:

- Rasterizer scan converts the given primitive into a series of fragments for processing by the rest of the pipeline.
- Scissor Test clips out fragments that lie outside the bounds of a user defined scissor rectangle and also performs screen clipping to stop illegal accesses outside the screen memory.
- Stipple Test masks out certain fragments according to a specified pattern. Line and area stipples are available.
- GID (Pixel Ownership) is concerned with ensuring that the location in the framebuffer for the current fragment is owned by the current visual. Comparison occurs between the given fragment and the Graphic ID value in the localbuffer, at the corresponding location, to determine whether the fragment should be discarded.
- Stencil Test conditionally discards a fragment based on the outcome of a test between the given fragment and the value in the stencil buffer at the corresponding location. The stencil buffer is updated dependent on the result of the stencil test and the depth test.
- Depth Test conditionally discards a fragment based on the outcome of a test between the depth value for the given fragment and the value in the depth buffer at the corresponding location. The result of the depth test can be used to control the updating of the stencil buffer.
- Color DDA is responsible for generating the color information (RGBA or Color Index(CI)) associated with a fragment.
- Texture is concerned with mapping a portion of a specified image (texture) onto a fragment. The process involves interpolating to determine the texel coordinates including perspective division, reading the texels, filtering to calculate the texture color, and application which applies the texture color to the fragment color.
- Fog blends a fog color with a fragment's color according to a given fog factor. Fogging is used for depth cueing images and to simulate atmospheric fogging.
- Antialias Application combines the incoming fragment's alpha value with its coverage value when antialiasing is enabled.
- Alpha Test conditionally discards a fragment based on the outcome of a comparison between the fragments alpha value and a reference alpha value.
- Alpha Blending combines the incoming fragment's color with the color in the framebuffer at the corresponding location.
- Color Formatting converts the fragment's color into the format in which the color information is stored in the framebuffer. This may optionally involve dithering.
- Logical Op/Framebuffer Mask performs Logical Operations between the fragment and destination, and optionally applies a writemask.
- Host Out optionally gathers statistics for picking and extent checking, and returns data to the host for image uploads.

¹ That is, provides the access functionality to the areas of memory used for LB storage.

The Pipeline structure of GLINT R4 is very efficient at processing fragments. For example, texture mapping calculations are not actually performed on fragments that are clipped off by scissor testing. This approach saves substantial computational effort.

To obtain the best results when programming for pipelined processing, however, you need to be aware of what all the pipeline stages are doing at any time. For example, many operations require both a read and/or write to the localbuffer and framebuffer. Because these are at different points in the pipeline the programmer must enable and control data read/write from/to the framebuffer – simply setting a logical operation to XOR and enabling logical operations will not have the desired effect.

The R4 introduces additional optimization to help the programmer manage memory accesses. The **ReadMonitor** unit allows Local and Framebuffer writes without waiting for earlier memory operations to complete where different polygons are affected. The result is a significant reduction in rendering latency, particularly for small polygons.

7.1.1 Router

As discussed in Volume I, the register address space can be seen conceptually as either a message passing system or a flat address map. This allows some significant adaptive performance enhancements. One important performance feature of the pipeline is the Router. This is essentially a switch which allows the order of some of the units to be swapped, by setting or clearing the *Sequence* bit of the **RouterMode** register.

Textured primitives are typically more processor-intensive than non-textured primitives. When the *Sequence* bit is set, fragments are tested against the GID (Pixel Ownership), Stencil and Depth(Z) before the texture value is calculated. If the fragment fails any of these tests nothing is drawn so texture value calculations can be skipped - leading to higher performance.

OpenGL defines the order of operations on a fragment as texture, alpha test, stencil then depth(Z), which is the sequence used when the *Sequence* bit in the Router register is cleared. However, if the alpha test is disabled (or cannot reject fragments) then OpenGL compatible semantics are maintained even if the operation order is changed to the more efficient stencil, depth(Z), texture, and alpha test.

The order can be dynamically reconfigured at any time without any need to synchronize simply by writing to the Order bit.

7.1.2 Initialization

GLINT R4 requires many of its registers to be initialized in a particular way regardless of what is to be drawn. For instance, the screen size and appropriate clipping must be set up. Normally this only needs to be done once and for clarity this example assumes that all initialization has already been done. More details may be found later in this volume.

Other states (e.g. enabling Gouraud shading and depth buffering) change occasionally though rarely on a per primitive basis.

7.1.3 Dominant and Subordinate Sides of a Triangle

The dominant side of a triangle is that with the greatest range of Y values. The choice of dominant side is optional when the triangle is either flat bottomed or flat topped.

GLINT R4 always draws triangles from the dominant edge towards the subordinate edges. This simplifies the calculation of set up parameters as will be seen below.

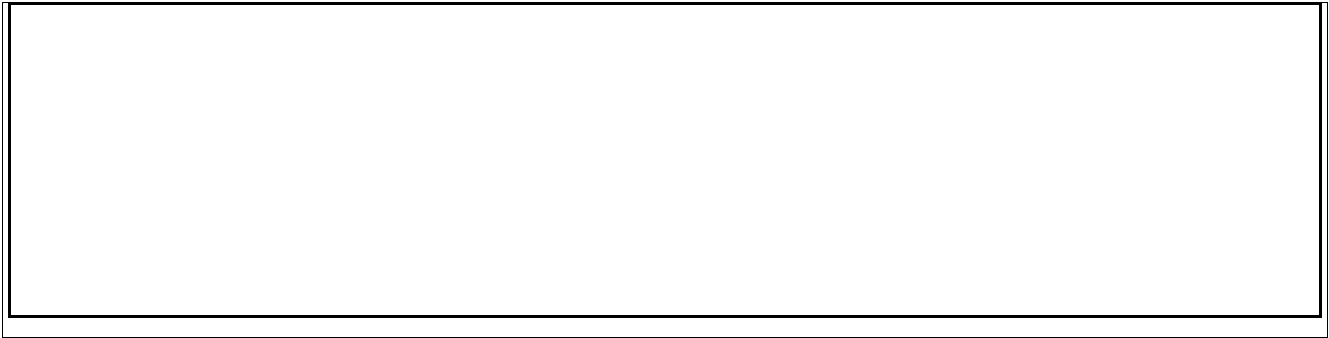


Figure 7.2 Dominant and Subordinate Sides of a Triangle

7.1.4 Register Set Up for Depth Testing

Internally GLINT R4 uses fixed point arithmetic. The formats for each register are described in the *Reference Guide*. Each depth value must be converted into a 2's complement 16.32 bit fixed point number and then loaded into the appropriate pair of 32 bit registers (**DZdxL**, **DZdxU**, **DZdyDomL**, **DZdyDomU**). The 'Upper' or 'U' registers store the integer portion, whilst the 'Lower' or 'L' registers store the 16 bit LSB, left justified and zero filled.

For the example triangle, R4 would need its registers set up as follows:

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

ZstartU (Z1_MS)
ZstartL (Z1_LS)
dZdyDomU (dZdy13_MS)
dZdyDomL (dZdy13_LS)
dZdxU (dZdx_MS)
dZdxL (dZdx_LS)
```

7.1.4.1 RasterizerMode

The R4 rasterizer has a number of mode bits which take effect until cleared and therefore tend to affect many primitives. These primarily involve bit mask operations described below. For details refer to the *Reference Guide*, **RasterizerMode** register. In the case of the Gouraud shaded triangle the default value for these modes is suitable.

7.1.5 Subpixel Correction

GLINT R4 supports subpixel correction of interpolated values when rendering aliased trapezoids (smooth shaded, textured, fogged or depth buffered). Subpixel correction ensures that all interpolated parameters associated with a fragment (color, depth, fog, texture) are correctly sampled at the fragment's center. This correction is required to ensure consistent shading of objects made from many primitives.

Subpixel correction is not applied to antialiased primitives².

² This applies only to antialiased primitives generated via the *Gamma* front end accelerator.

Control of subpixel correction is in the **Render** command register described below, and can be selected in bit settings for individual primitives (**DrawLine**, **DrawTriangle**). A full code example is given in the Appendices.

7.2 Pipeline Overviews

Before we review each unit in detail it is worth looking in general terms at how a graphic primitive passes through the pipeline, what messages are generated and what happens in each unit. Some simplifications have been made in the description to avoid detail which would otherwise complicate what is in fact a very simple process.

The descriptions concentrate on what happens as a fragment flows down the message stream. It is important to remember that at any instant in time there are many fragments flowing down the message stream and the further they get the more processing has occurred.

7.2.1 A day in the life of a 3D triangle

This section previews the render process for a typical 3D graphics primitive, the Gouraud shaded, depth buffered, dithered triangle.

For this example assume that the triangle is to be drawn into a window which has its colormap set for RGB as opposed to color index operation. This means that all three color components; red, green and blue, must be handled. Also, assume the coordinate origin is bottom left of the window and drawing will be from top to bottom. GLINT R4 can draw from top to bottom or bottom to top.

For clarity the equations are shown in full in the appendices, though in practice there are many common terms and factors which need only be computed once and normally the OGL driver performs all the necessary interpolations.

Consider a triangle with vertices, v_1 , v_2 and v_3 where each vertex comprises X, Y and Z coordinates, shown below. Each vertex has a different color made up of red, green and blue (R, G and B) components. The alpha component is omitted for this example.

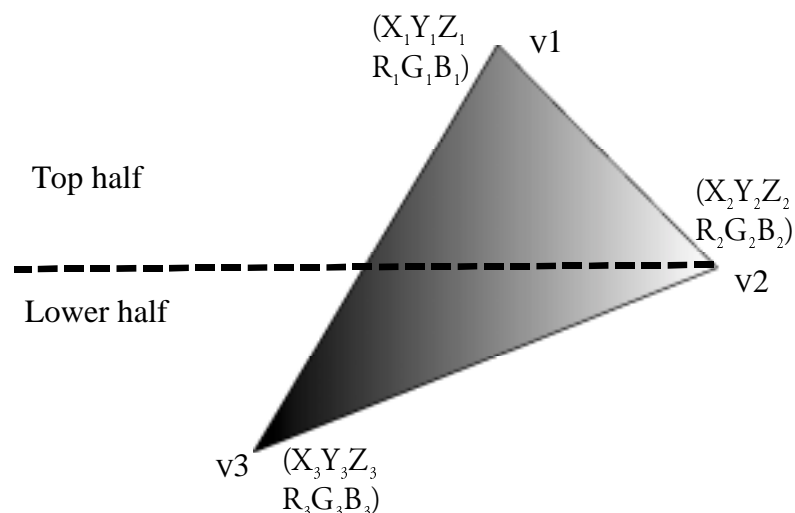


Figure 7.3 Example Triangle

The diagram makes a distinction between top and bottom halves because GLINT R4 is designed to rasterize (a) screen aligned trapezoids, and (b) flat-topped or -bottomed triangles; as shown below:



Figure 7.4 Screen aligned trapezoid and flat topped triangle

7.2.1.1 Delta Unit

The drawing process starts by generating and loading vertex data in the Delta Unit:

1. The application generates the triangle vertex information and makes the necessary OpenGL calls to draw it.
2. The OpenGL server/library gets the vertex information, transforms, clips and lights it. The vertex coordinates and color values are written into the vertex stores (in the Delta Unit) and the **DrawTriangle** command is issued.
3. The Delta Unit calculates the initial values and derivatives for the values to interpolate (X_{left} , X_{right} , red, green, blue and depth) for unit change in dx and dx_{left} . All these values are in fixed point integer and have unique message tags. Some of the values (the depth derivatives) have more than 32 bits to cope with the dynamic range and resolution so are sent in two halves. Finally, once the derivatives, start and end values have been sent the 'render triangle' message begins the rendering process.
4. The derivative, start and end parameter messages are received and filter down the message stream to the appropriate units. The depth parameters and derivatives to the Depth Unit; the RGB parameters and derivative to the Color DDA Unit; the edge values and derivatives to the Rasteriser Unit.

7.2.1.2 Rasterizer

The 'render triangle' message is received by the rasteriser unit and all subsequent messages (from the host) are blocked until the triangle has been rasterised (but not necessarily written to the framebuffer). A 'prepare to render' command is passed on so any other units can prepare themselves.

The Rasteriser Unit walks the left and right edges of the triangle and fills in the spans between. As the walk progresses messages are sent to indicate the direction of the next step: StepX or StepYDomEdge.

7.2.1.3 Rasterizer 'Edge walking' - Calculating the Slope for each Side

GLINT R4 draws filled shapes such as triangles as a series of spans with one span per scanline. Therefore it needs to know the start and end X coordinate of each span. These are determined by 'edge walking'. This process involves adding one delta value to the previous

span's start X coordinate and another delta value to the previous span's end x coordinate to determine the X coordinates of the new span. These delta values are in effect the slopes of the triangle sides. To draw from left to right and top to bottom, the slopes of the three sides are calculated as:

$$dX_{23} = \frac{X_3 - X_2}{Y_3 - Y_2}$$

$$dX_{13} = \frac{X_3 - X_1}{Y_3 - Y_1}$$

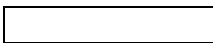
$$dX_{12} = \frac{X_2 - X_1}{Y_2 - Y_1}$$

This triangle will be drawn in two parts, top down to the 'knee' i.e. vertex 2 and then from there to the bottom. The dominant side is the left side so for the top half:



The start X,Y, the number of scanlines, and the deltas (above) give GLINT R4 enough information to edge walk the top half of the triangle. However, to indicate that this is not a flat topped triangle (GLINT R4 is designed to rasterize screen aligned trapezoids and flat topped triangles), the same start position in terms of X must be given twice as StartXDom and StartXSub.

To edge walk the lower half of the triangle, selected additional information is required. The slope of the dominant edge remains unchanged, but the subordinate edge slope needs to be set to:



Also the number of scanlines to be covered from Y2 to Y3 needs to be given. Finally to avoid any rounding errors accumulated in edge walking to X2 (which can lead to pixel errors), StartXSub must be set to X2.

The data field holds the current (x, y) coordinate. One message is sent per pixel within the triangle boundary. These messages, or fragments, are divided into two groups, active and passive. Fragments always start off in the active group but may be changed to the passive group if the pixel fails one of the tests (e.g. depth) on its path down the message stream. The two groups are distinguished by a single bit in the message tag.

The fragments (in either form) are always passed throughout the length of the message stream and are used by all the DDA units to keep their interpolation values in step. Any other messages pertaining to fragments always precede the fragment in the message stream.

The messages hold X, Y, color and coverage data for each fragment³. The data field expands between units to accommodate additional data when necessary.

³The coverage field is only used for antialiasing. For aliased primitives the coverage field holds a dErr value used for subpixel correction.

7.2.1.4 Rasterizing the Triangle

We are almost ready to draw the triangle. Setting up the registers as described here and sending the **Render** command draws the top half of the example triangle first.

To draw the example triangle, all the bit fields within the **Render** command should be set to 0 except the **PrimitiveType** which should be set to trapezoid and the *SubPixelCorrection Enable* bit which should be set to TRUE.

```
// Draw triangle with knee
// Set deltas
StartXDom (X1<<16) //Converted to 16.16 fixed point
dXDom (((X3 - X1)<<16)/(Y3 - Y1))
StartXSub (X1<<16)
dXSub (((X2 - X1)<<16)/(Y2 - Y1))
StartY (Y1<<16)
dY (-1<<16)
Count (Y1 - Y2)

// Set the render command mode
render.PrimitiveType = GLINT R4_TRAPEZOID_PRIMITIVE
render.SubPixelCorrectionEnable = TRUE
// Draw the top half of the triangle

Render(render)
```

After the **Render** command has been issued the registers in R4 can immediately be altered to draw the lower half of the triangle. Only two registers need be loaded and the **ContinueNewSub** command sent. Once R4 has received **ContinueNewSub** it starts drawing the sub-triangle.

```
// Set-up the delta and start for the new edge
StartXSub (X2<<16)
dXSub (((X3 - X2)<<16)/(Y3 - Y2))

// Draw sub-triangle
ContinueNewSub (Y2 - Y3) // Draw lower half
```

7.2.1.5 Scissor and Stipple

This unit does 4 tests on the fragment (as embodied by the active step message). The screen scissor test takes the coordinates associated with the fragment, converts them to be screen

relative (if necessary) and compares them against the screen boundaries. (The other three tests - user scissor, line stipple and area stipple - are disabled in this example.) If the enabled tests pass then the active fragment is forwarded to the next unit, otherwise it is changed into a passive step and then forwarded.

7.2.1.6 Router

In this example the Router is set up so the Depth test occurs before the texture operations (i.e. **RouterMode Sequence** bit = 1).

7.2.1.7 Local Buffer Read

In general terms the Local Buffer Read Unit reads the Graphic ID, Stencil and Depth information from the Local Buffer and passes it to the next unit. This includes performing the GID test on fragments, checking cache and local buffer for data, and data formatting. See volume I - Localbuffer - for more information

7.2.1.8 Stencil and Depth

When an active fragment is received the internal stencil and depth values are compared with the fragment's as specified in the **StencilMode** and **DepthMode** registers. If the enabled tests pass then the new local buffer data is written back to the fragment, which is forwarded to the next unit.

If any of the enabled tests fail then equivalent passive step message is forwarded instead (a local buffer write may still be done). The Depth DDA is stepped to update the local depth value.

7.2.1.9 Local Buffer Write

The Local Buffer Write Unit calculates the address, formats the GID, Stencil and Depth data and (if writes are enabled) passes the formatted data and address to the Memory Controller.

The memory is much wider than the pixel data so any writes are first done into a write combine buffer which is flushed to memory as required. See volume I - Localbuffer - for more information.

The fragment is forwarded to the next unit.

7.2.1.10 Color DDA

The Color DDA unit responds to an active fragment by updating the fragment's color field and sending this to the next unit. The color field holds the *current* RGBA value from the DDA. After the step message is sent the DDA is incremented in the correct direction, ready for the next pixel.

7.2.1.11 Texturing, Fog and Alpha Tests

In this example, Texturing, Fog and Alpha Tests are disabled so the fragments are forwarded unchanged.

7.2.1.12 Framebuffer Read

In general terms Framebuffer Read reads the color information from the framebuffer and passes it onto the next unit. It is functionally similar to the Localbuffer but handles color data

rather than GID, depth and stencil data, write-combined operations and Patch2 and Patch32_2 formats. See volume I - Framebuffer - for more information.

7.2.1.13 Alpha Blend

The formatting of the Framebuffer data is deferred until the Alpha Blend Unit as it is the only unit which needs to match buffer formats with the internal formats.

In this example no alpha blending or logical ops are taking place so reads are disabled and fragments pass through unaltered.

7.2.1.14 Dither

The Dither Unit uses the least significant bits of the (X, Y) coordinate information from the step message to dither the color field. Part of the dithering process is to convert from the internal color format into the format of the framebuffer. The new color is inserted back into the color field and the fragment forwarded.

7.2.1.15 Logical Ops

In this example Logical Ops are disabled so the fragments pass through.

7.2.1.16 Framebuffer Write

The Framebuffer Write Unit calculates the address and (if writes are enabled) passes the formatted data and address to the Memory Controller.

The memory is much wider than the pixel data so any writes are first done into a write combine buffer and only when this needs to be flushed is the Memory Controller given the write command - see volume I - Framebuffer - for more information.

7.2.1.17 Host Out

The Host Out Unit deals with host synchronisation and statistics. In this example it simply consumes any fragments which reach this point in the message stream.

7.2.2 A day in the life of a 2D primitive

GLINT R4 introduces an alternative method for rendering which is particularly suited to 2D operations. These are pure 2D without any 3D functionality such as depth or stencil testing or alpha blending.

2D drawing works on spans of pixels, where a span is always 64 pixels sequentially along a scanline. The core now works on 64 pixels in parallel (in addition to processing multiple spans along the length of the message stream) and a pixel can be 8, 16 or 32 bits in size. Spans can be read, written, copied, uploaded or downloaded. A span can have a constant color or a variable color per pixel in the span.

The primitive we are going to look at is a fill with a constant color through a bit mask held in the texture memory. The zero bits in the bit mask do not cause the corresponding pixels in the framebuffer to be written to. The fill shape is a rectangle for simplicity, but could be any shape (with suitable decomposition into primitives GLINT R4 understands). As usual, we refer to "units" along the message stream but these are more accurately considered as functional groupings than physical entities.

7.2.2.1 Initialization

The application generates the rectangle information and makes the necessary Windows API calls to draw it.

7.2.2.2 2D Set Up Unit

The NT device driver gets the rectangle information and uses the **Render2D** command to set up and rasterise the rectangle. Other state data and information is also set up, as discussed below.

7.2.2.3 Rasterizer

The 'render trapezoid' message is received by the rasteriser unit and all subsequent messages (from the host) are blocked until the trapezoid has been rasterised (but not necessarily written to the framebuffer). The **Render** message has the *FastFillEnable* bit set. A 'prepare to render' message is also passed on internally so any other units can prepare themselves.

The Rasteriser Unit walks the left and right edges of the trapezoid (a rectangle in this case) and fills in the spans between the left and right hand edges. As the walk progresses messages are sent to indicate the direction of the next step. These internal SpanStep commands control the subsequent processing of the span fragment.

7.2.2.4 Scissor and Stipple Unit

Scissor and Stipple Unit. This unit does 3 tests on the span (as embodied by the SpanStep message). The screen scissor test takes the coordinates associated with the SpanStep message, converts them to be screen relative (if necessary) and compares the pixel mask against the screen boundaries and clears the bits for pixels which lie outside the screen boundary. The pixel mask is potentially further reduced using the scissor tests (applied in a similar way). The area stipple test is disabled for this example but, if it was enabled would potentially remove further pixels after suitable alignment. The modified SpanStep message is forwarded to the next unit.

7.2.2.5 Color DDA

The Color DDA unit does not respond to the SpanStep messages so they just pass through.

7.2.2.6 Texture Coordinate and Index

The Texturing Coordinate Unit responds to the SpanStep message and appends the u, v coordinates of the texel where the bit mask data for this span is held. The S and T DDAs are set up to step through the bit mask pattern. The SpanStep is forwarded on to the next unit.

The Texture Index Unit converts the uv coordinate in the SpanStep message into an ij coordinate of the texel where the bit mask data for this span is held. The SpanStep is forwarded on to the next unit.

7.2.2.7 Texturing, Fog and Alpha

The Texture Read Unit converts the ij coordinate into a physical address where the texel data is held. The texel data is read (maybe sourced from the secondary cache) and zero extended up to 64 bits if the bit mask was held as 8, 16, or 32 bits. After being optionally inverted or mirrored it is ANDed with the pixel mask field in the SpanStep message and forwarded to the next unit.

The remaining texture units, Fog and Alpha Tests Units do not respond to the SpanStep messages so they just pass through.

7.2.2.8 Localbuffer Read, Stencil/Depth and Localbuffer Write

The LB Read, Stencil/Depth, LB Write Units do not respond to the SpanStep messages (in this example) so they just pass through.

7.2.2.9 Framebuffer Read

In general terms the Framebuffer Read Unit reads the color information from the framebuffer and forwards it to the next unit. More specifically for spans it calculates the linear address in the framebuffer of the required data. This is done using the (X, Y) position recorded internally and locally stored information on the 'screen width' and window base address. The span is decomposed into a series of memory aligned reads.

In this example no logical ops are taking place so reads are disabled and hence no read address is sent to the Memory Controller. The span tags just pass through.

7.2.2.10 Alpha Blend and Dither

The Alpha Blend and Dither Units do not respond and the span data simply passes through.

7.2.2.11 Logical Ops

The Logical Ops are disabled so the Span data passes through.

7.2.2.12 Framebuffer Write

The Framebuffer Write Unit calculates the address, aligns the pixel mask to the memory block write boundaries and passes these to the Memory Controller. The pixel data previously set up in the **FBColor** Register can be written, ideally using the block fill capability of the SGRAM. The Span data is passed on to the next unit.

7.2.2.13 Host Out

The Host Out Unit is concerned with synchronisation with the host - for this example it simply consumes any messages which reach this point in the message stream.

8

Rasterizer and 2D Setup

The rasterizer decomposes a primitive into a series of fragments for processing by the rest of the HyperPipeline.

GLINT R4 can directly rasterize:

- aliased screen aligned trapezoids
- aliased single pixel wide lines
- aliased single pixel points
- antialiased screen aligned trapezoids
- antialiased circular points

All other primitives are treated as one or more of the above, for example an antialiased line is drawn as a series of antialiased trapezoids.

2D Operations can be largely implemented using the **Render2D** and **Render2Dglyph** registers. These, together with the **GlyphData** and **GlyphPosition** registers constitute a functional subunit of the Rasterizer and are discussed below.

8.1 Description

The rasterizer unit scan converts the given primitive into a list of pixel coordinates which meet the rasterisation rules of OpenGL, X and NT. In addition to generating the coordinates, the order in which pixels are visited is also defined (by the **Render** command) so the local DDA units in the Texture, Color, Fog and Depth units can incrementally keep in step.

When a primitive is antialiased the percentage coverage of the primitive within the scan converted pixels is calculated for later use in the alpha blend unit. The same method of antialiasing is used for all primitives⁴.

The primitive is scan converted to a higher resolution (e.g. 4x4 sub samples per Render command) and the number of sub pixel sample points covered is counted. The ratio of covered sample points to total number of sample points gives the coverage weighting by which to adjust the color.

The rasterisation process steps through along the Y axis and calculates the two intersection points for this scanline. For normal rasterisation the pixels between these two intersection points are filled in. During antialiasing a step of Y/4 (for example) is used and within each scan line four pairs of intersections are calculated per scanline. The coverage for each of the four sub pixel scanline makes in a pixel (on this scanline) are calculated and summed.

The coordinates passed to the rasterizer can be window relative or screen relative. The rasterizer treats both the same. Conversion to memory addresses does not happen until they reach the Local Buffer and Framebuffer Units.

⁴ Applies to *Gamma* interface only.

The Rasterizer is not concerned whether the origin is the bottom left or top left and again it is the Local Buffer and Framebuffer Units which take this into account when calculating the memory address. Obviously if the direction of scan conversion is important then the parameters must match up with the origin definition to give the desired effect.

*Note: Long term mode information is held in the **RasterizerMode** command and short term mode information (which only applies to the primitive being rasterised) is passed with the **Render** command.*

8.1.1 Trapezoids

GLINT R4's basic area primitive is the screen aligned trapezoid, discussed in the previous chapter. This is characterized by having top and bottom edges parallel to the X axis. The side edges may be vertical (a rectangle), but are usually diagonal. The top or bottom edges can degenerate into points in which case we are left with either flat topped or flat bottomed triangles.

Any polygon can be decomposed into screen aligned trapezoids or triangles. Usually, polygons are decomposed into triangles because the interpolation of values over non-triangular polygons is ill defined. The rasterizer does handle flat topped and flat bottomed 'bow tie' polygons which are a special case of screen aligned trapezoids.

X's definition of a polygon is more complex than OpenGL's. It can be concave and self intersecting. In the non convex case the best thing is for X to do is to decompose the polygon into a series of spans and render them as 1 pixel high rectangles. For any convex polygons X can decompose them into screen aligned trapezoids as a further optimisation over just using spans. X does not support antialiased polygons.

8.1.1.1 Edge Matching for Meshes, Fans etc.

Adjacent triangles or polygons which share an edge or vertex must be drawn so that pixels which make up the edge or vertex are drawn once only. This may be achieved by omitting the pixels down the left or the right sides and the pixels along the top or lower sides. GLINT R4 follows the convention of omitting the pixels down the right hand edge. Control of whether pixels along the top or lower sides are omitted depends on the start Y value and the number of scanlines to be covered. With the example, if StartY = Y1 and the number of scanlines is set to Y1toY2, the lower edge of the top half of the triangle will be excluded. This excluded edge is drawn as the top of the lower half of the triangle.

To minimize delta calculations, triangles may be scan converted from left to right or from right to left. The direction depends on the dominant edge, that is *the edge which has the maximum range of Y values*. Rendering always proceeds from the dominant edge towards the relevant subordinate edge. In the example above, the edge with the greatest Y range (dominant) is on the right so rendering will be from right to left.

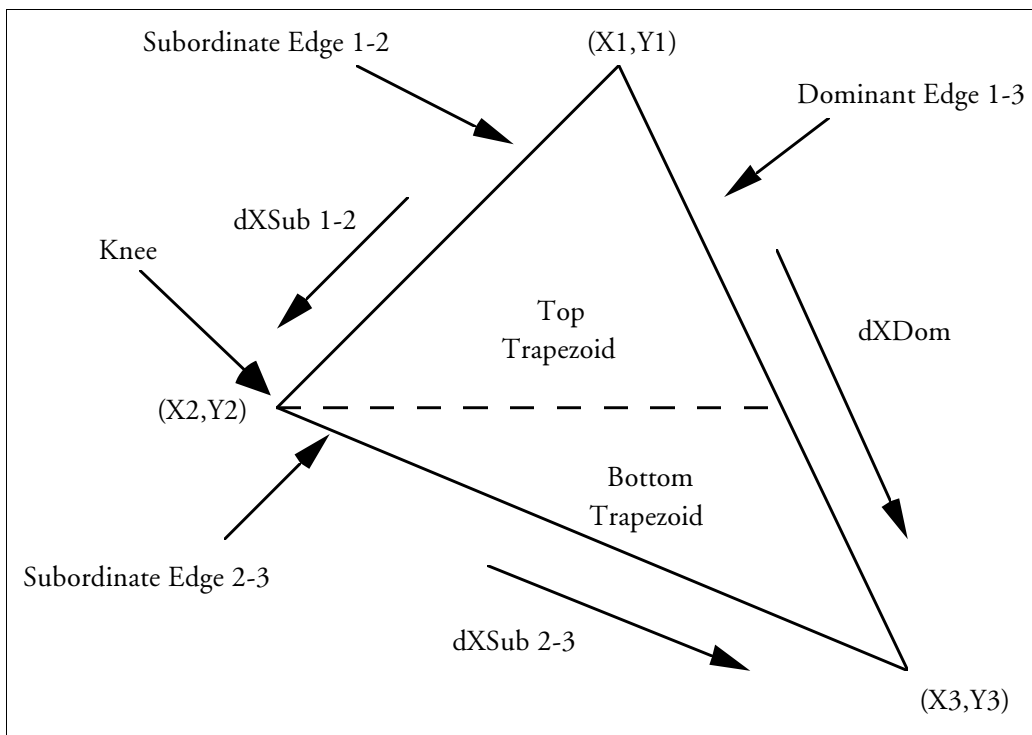


Figure 8.1 Rasterizing a triangle

The sequence of actions required to render a triangle (with a 'knee') are:

- Load the edge parameters and derivatives for the dominant edge and the first subordinate edges in the first triangle.
- Send the **Render** command. This starts the scan conversion of the first triangle, working from the dominant edge. This means that for triangles where the knee is on the left we are scanning right to left, and vice versa for triangles where the knee is on the right.
- Load the edge parameters and derivatives for the remaining subordinate edge in the second triangle.
- Send the **ContinueNewSub** command. This starts the scan conversion of the second triangle.

Render Data Field					
AreaStippleEnable	1	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiasEnable	0	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	1

StartXDom (X_1)

$dX_{Dom} ((X_3 - X_1) / (Y_3 - Y_1))$

```

StartXSub (X1)
dXSub ((X2- X1)/(Y2 - Y1))
StartY (Y1)
dY (-1.0)
Count (Y1 - Y2)
Render
StartXSub (X2)
dXSub ((X3- X2)/(Y3 - Y2))
ContinueNewSub (Y2 - Y3) // Bottom half

```

Note: If both edges need to be reloaded to continue on with the bottom half of the polygon then issue `ContinueNewSub (0)` and then `ContinueNewDom (count)`. The `ContinueNewSub (0)` will just update the DDA with the new start value but not draw any scanlines. Alternatively, if the accuracy of the DDA end values is good enough and can be used as the start values for the next trapezoid then the delta values can be updated and the `Continue` message used.

The sub pixel correction is only needed if color, depth, fog or texture interpolation is being used.

After the **Render** command has been sent the registers can be updated immediately to draw the second half of the triangle. Only two registers need to be loaded to do this, followed by the **ContinueNewSub** command. When the first triangle has been drawn and the **ContinueNewSub** command issued, GLINT R4 starts drawing the sub-triangle and the **ContinueNewSub** command register is loaded with the remaining number of scanlines to be rendered.

8.2 Antialiasing

Antialias application controls the way the coverage value generated by the rasterizer combines with the color generated in the color DDA units. The application depends on the color mode - RGBA or Color Index (CI).

Note: The following comments apply to rasterization of polygons set up in the front end Gamma chip. R4 does not directly support antialiasing in standalone mode.

8.2.1 Antialias Application

When antialiasing is enabled by setting the **AntialiasMode Enable** bit and the **Render** register's `CoverageEnable` bit, the fragment's color and alpha is weighted by the percentage area of the pixel covered by the fragment. The coverage weighting is determined by the Rasterizer and varies from 0 to 100% "saturation".

If antialiasing is not enabled the fragment is forwarded for alpha testing.

The mode (RGBA or CI) is set using the `ColorMode` bit in the **AntialiasMode** register. In RGBA mode the color value is multiplied by the coverage value calculated in the rasterizer (its range is 0% to 100%). The RGB values remain unchanged unless the `ScaleColor` bit is also set. Color scaling is not required by OGL and may reduce performance.

In CI mode the coverage value is placed in the lower 4 bits of the color field. The Color Look Up Table is assumed to be set up so that each color has 16 intensities associated with it, one per coverage entry.

8.2.2 Polygon Antialiasing Considerations

A number of issues should be considered when using GLINT R4 to render antialiased polygons. Depth buffering cannot be used with GLINT R4 antialiasing. This is because the order the fragments are combined in is critical in producing the correct final color. Polygons must therefore be depth sorted, and rendered front to back, using the alpha blend modes: *SourceAlphaSaturate* for the source blend function and *One* for the destination blend function. In this way the alpha component of a fragment represents the percentage pixel coverage, and the blend function accumulates coverage until the value in the alpha buffer equals one, at which point no further contributions can be made to a pixel.

Although this technique works well in many cases, it is an approximation. Consider the case below which shows three polygons of equal depth which intersect a single pixel. In this case there would ideally be a contribution from each of the polygons. However, if the rendering order is polygon A followed by polygon B, each of which contributes approximately 50% pixel coverage, then polygon C will make no contribution to the pixel as the alpha value is saturated ($50\%+50\%=100\%$).

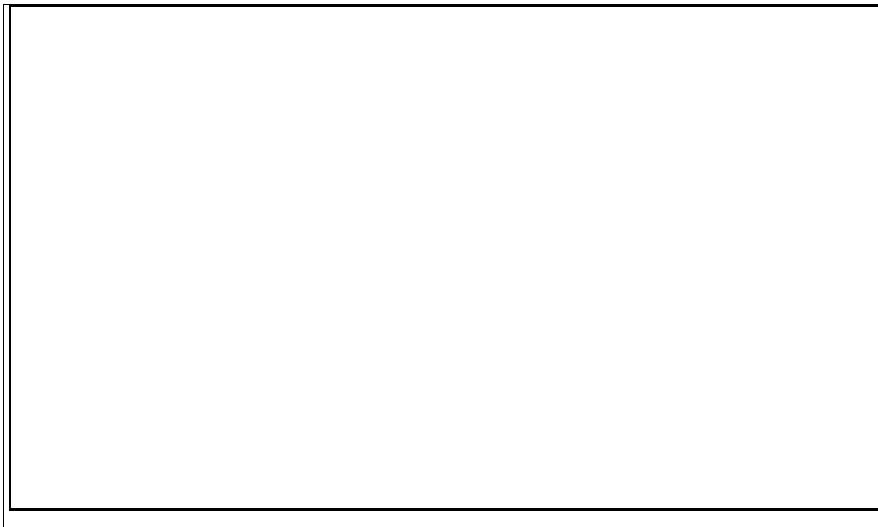


Figure 8-2 Polygon Antialiasing

When antialiasing general scenes with no restrictions on rendering order, the accumulation buffer is the preferred choice. This is indirectly supported on GLINT R4 via image uploading and downloading, with the accumulation buffer residing on the host.

When antialiasing, interpolated parameters which are sampled within a fragment (color, fog and texture), sometimes are not representative of a continuous sampling of a surface so care should be taken when rendering smooth shaded antialiased primitives. This problem does not occur in aliased rendering, as the sample point is consistently at the center of a pixel. See *The OpenGL Programming Guide* for more details of antialiasing.

8.2.3 Registers

The **AntialiasMode** register provides the enables described earlier:

Name	Type	Offset	Format
AntialiasMode	Alpha Test	0x8808	Bitfield
AntialiasModeAnd	Alpha Test	0x ABF0	Bitfield Logic Mask
AntialiasModeOr	Alpha Test	0x ABF8	Bitfield Logic Mask

Control register

Bits	Name	Read	Write	Reset	Description
0	Enable	✓	✓	X	When set, scales the fragment's alpha value under control of the remaining bits in this register and the coverage value. When clear, the fragment's alpha value is not changed. 0 = Disable 1 = Enable
1	Color Mode	✓	✓	X	This bit defines the fragment's color format: 0 = RGBA 1 = CI
2	Scale Color	✓	✓	X	This bit, when set allows the coverage value to scale
3...31	Unused	0	0	X	

Notes: For the coverage application to take place the enable in the **AntialiasMode** register must be qualified by the *CoverageEnable* bit in the **Render** command register.

Figure 8-3 AntialiasMode Register

8.2.4 Antialias Example

Enable antialiasing for an RGBA primitive:

```
// Set AA application for RGBA primitive
begin.AntialiasEnable = GLINT_R4_TRUE
antialiasMode.AntialiasEnable = GLINT_R4_TRUE
antialiasMode.ColorMode = GLINT_R4_TRUE
AntialiasMode(antialiasMode)

// Set the blend mode to an appropriate value if
// blending is required. Not shown.
// When issuing a Render command the CoverageEnable
// bit should be set in addition to the antialias
// unit being enabled:
// render.CoverageEnable = GLINT_R4_TRUE
```

8.2.5 Antialiasing Primitive Types

R4 uses a subpixel point sampling algorithm to antialias primitives. R4 can directly rasterize antialiased trapezoids and points. Other primitives are composed from these base primitives.

The rasterizer associates a coverage value with each fragment produced when antialiasing. This value represents the percentage coverage of the pixel by the fragment. GLINT R4 supports two levels of antialiasing quality:

- normal, which represents 4x4 pixel subsampling
- high, which represents 8x8 pixel subsampling

Selection between these two is made by the *AntialiasingQuality* bit in the **Render**, **PointMode**, **TriangleMode** or **LineMode** registers.

Use the **FlushSpan** command to terminate rendering an antialiased primitive. This is necessary because of the way R4 maintains antialiasing continuity.

When rendering a primitive which does not complete on a scanline boundary, R4 retains antialiasing information about the last sub-scanline(s) it has processed but does not generate fragments for them unless a **FlushSpan** command is received. The commands **ContinueNewSub**, **ContinueNewDom** or **Continue** can then be used to maintain continuity between adjacent trapezoids, which allows complex antialiased primitives to be built up from simple trapezoids or points.

To illustrate this consider using screen aligned trapezoids to render an antialiased line. The line will in general consist of three screen aligned trapezoids as shown in the diagram below.



Figure 8-4 Antialiased Line

The procedure to render the line is as follows:

```
// Set-up the blend and coverage application units
// as appropriate – not shown
// In this example only the edge deltas are shown
// loaded into registers for clarity. In reality
// start X and Y values are required. This example
// uses 4x4 antialiasing.
```

```

// Render Trapezoid A

dY(1<<14)
dXDom(dXDom1<<14)
dXSub(dXSub1<<14)
Count(count1<<2)
render.PrimitiveType = GLINT R4_TRAPEZOID
render.AntialiasEnable = GLINT R4_TRUE
render.AntialiasQuality = GLINT R4_MIN_ANTIALIAS
render.CoverageEnable = GLINT R4_TRUE
Render(render)

// Render Trapezoid B

dXSub(dXSub2<<14)
ContinueNewSub(count2<<2)

// Render Trapezoid C

dXDom(dXDom2<<14)
ContinueNewDom(count3<<2)

// Now we have finished the primitive flush out
// the last scanline
FlushSpan()

```

Note: When rendering antialiased primitives, any count values should be given in subscanlines. For example if the quality is 4x4 then the count will be 4 times the number of scanlines completely covered by the primitive plus the number of subscanlines contained in the remaining partially covered scanlines. Also, if using 4x4 quality then any delta value must be divided by 4. If using 8x8 quality then the multiply/divide factor is 8.

When rendering, *AntialiasEnable* must be set in the **AntialiasMode** register to scale the fragment's color by the coverage value. An appropriate blending function should also be enabled. See the Antialias Application and Alpha Blend sections for more details.

Note: When rendering antialiased bow-ties the coverage value on the cross-over scanline may be incorrect.

8.2.5.1 Antialiased Polygons

As for other primitives, the following comments apply to rasterization of polygons set up in the front end Gamma chip. R4 does not directly support antialiasing in standalone mode.

Antialiased polygons (or more precisely, screen aligned trapezoids) are scan converted by walking the trapezoid's edges to a higher resolution (x4, say). The coverage for a specific pixel is calculated by summing the coverage each of the sub scanlines contributes. More specific details are given in the implementation section.

Care needs to be taken when trapezoids (from the same polygon) meet part way through a scan line. The span of pixels cannot be generated until the second trapezoid is available as it will contribute to the coverage in this scanline. If, on the last trapezoid, the scan line is only part covered then a 'flush' command is needed to generate the coverage for these pixels as there is no follow-on trapezoid.

8.2.5.2 Stippling during Rasterizing

Normally, stipple processing is accommodated in the Stipple Unit. This covers all stipple requirements for OpenGL (e.g. aliased lines, polygons) and most other platforms, e.g. X. Details are given in the Stipple Unit section.

The Rasterizer does provide additional stipple functionality, for example stippling requirements for X which cannot be met by the Stipple Unit:

- Arbitrary stipple on lines.
- Arbitrary stipple on polygons, especially rectangles.

The bit mask unit in the rasterizer (normally used for characters) can give an arbitrary stipple to any primitive. The stipple pattern required is loaded into the **BitMaskPattern** register 32 bits at a time, in the order in which the pixels in the primitive are generated. The state of each bit in the bit mask determines if an active pixel is generated or a passive one. One bit in the stipple sequence is required for each pixel in the primitive.

This stippling method is independent of the Stipple Unit and can replace its function or be used as a second level of stippling.

8.2.5.3 Stipple Lines (X)

The standard OpenGL method of stippling lines can be used in X for the more restricted case where the mark/space ratio of the stipple is the same. X allows an arbitrary stipple pattern to be defined using the Bitmap facility. Here the host provides a number of bit mask words where each bit corresponds to one pixel in the line. The state of this bit determines whether the associated pixel is generated or skipped.

8.2.6 Points

Points are the easiest of all primitives to scan convert but there are a number of special cases. The main questions are whether the point is antialiased or not, and its size.

All the DDA related parameters are held constant over a point (a point may cover many pixels), and between points in a Begin/End set. Before any point rasterisation is done the host must have set up the Texture, Color, Fog and Depth units so they maintain a constant value and don't increment between pixels in a point.

In OpenGL no stipple operations are defined for points so stippling must be disabled. This can be done by changing the stipple mode (see Stipple Unit) or by setting the stipple operation in the **Render** command to 'none'. This later method is much easier for the software to use.

R4 does not support a **DrawPoint** command because points do not require setup calculations. Instead points can be individually loaded as below, or set up as line segments and rendered as points.

8.2.6.1 Aliased Points (OpenGL)

For points larger than one pixel, trapezoids should be used. The fields in the **Render** command register are described in detail later, however, in this case the *PrimitiveType* field in the **Render** command should be set to equal GLINT R4_POINT_PRIMITIVE.

8.2.6.2 Worked example – one-pixel points

A series of one pixel points $P(X_1, Y_1)$, $P(X_2, Y_2) \dots P(X_n, Y_n)$ are required. The **Render** command is set up as shown:

Render Data Field					
AreaStippleEnable	0	LineStippleEnable	0	PrimitiveType	2
FastFillEnable	0	FastFillIncrement	X	UsePointTable	X
AntialiasEnable	X	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	X	SyncOnHostData	X	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	0

StartXDom (X_1)

StartY (Y_1)

Render

StartXDom (X_2)

StartY (Y_2)

Render

... ..

... ..

StartXDom (X_n)

StartY (Y_n)

Render

8.2.6.3 Aliased Points (X)

X only has single pixel sized points so these are rendered by sending any of the step commands with the (X, Y) position encoded in the data field for each point to render.

8.2.6.4 Antialiased Points (OpenGL)

R4 can render antialiased points using the *Gamma* front end to define OpenGL points. Antialiased points are treated as circles, with the coverage of the boundary fragments ranging from 0% to 100%. For information about available sizes and the Points Table, see the **AAPointSize** register in the R4 Reference Guide, volume III.

R4 supports:

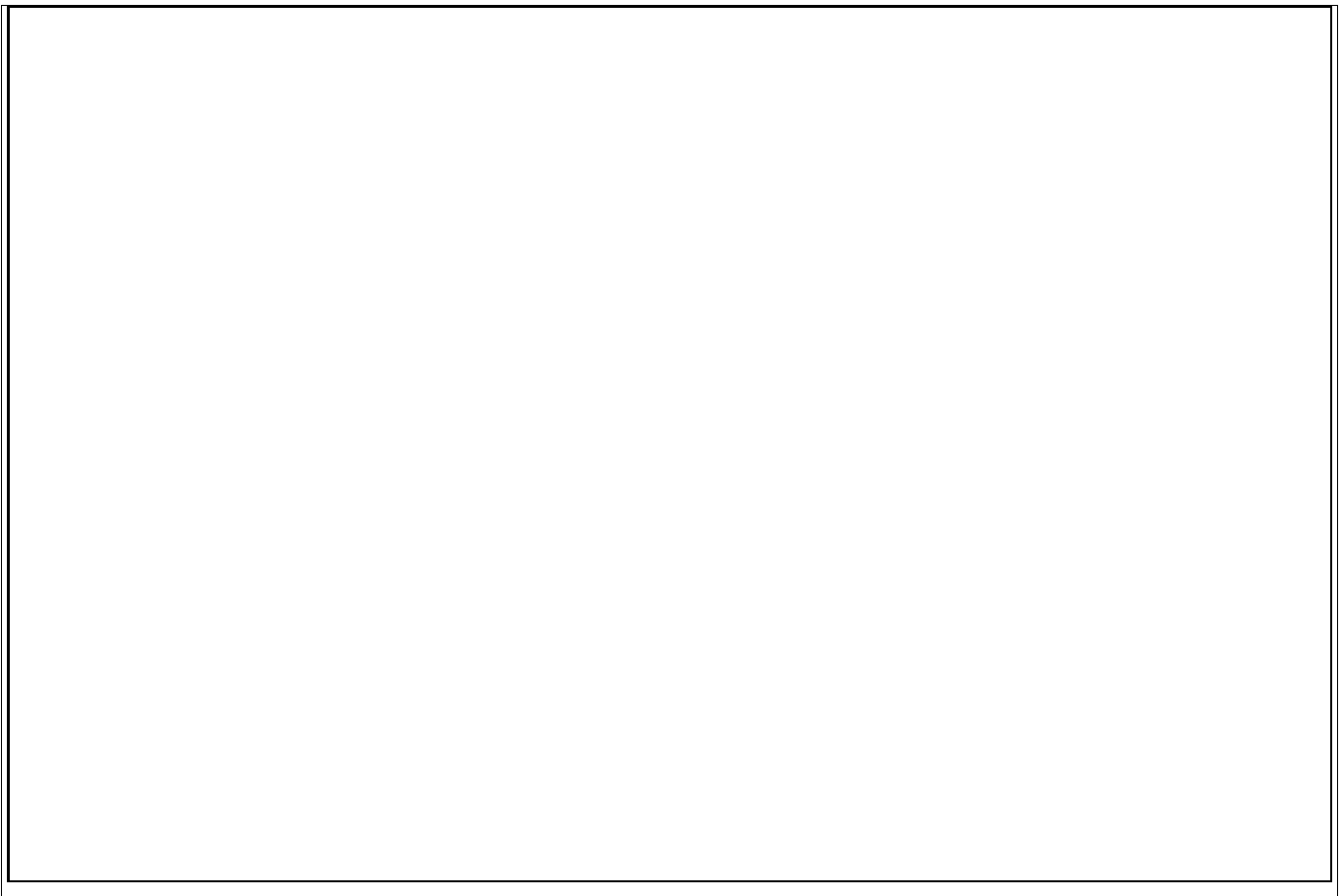
- point diameters of 0.5 to 16.0 in steps of 0.25 for 4x4 antialiasing
 - point diameters of 0.25 to 8.0 in steps of 0.125 for 8x8 antialiasing
- ...using the *AntiAliasQuality* bit in the **Render** or **PointMode** register

To scan convert an antialiased point as a circle R4 traverses the boundary in sub scanline steps to calculate the coverage value. For this, the sub scanline intersections are calculated incrementally using a small table. The table holds the change in X for a step in Y. Symmetry is used so the table only holds the delta values for one quadrant.

The pattern of table accesses, additions and subtractions are shown in Figure 2.3 below for an odd diameter point. On the diagram the symbol $\pm = \text{Table}[n]$ by an arrow indicates the contents of the **PointTable** at address n are added/subtracted to move along the arrow.

StartXDom, **StartXSub** and **StartY** are set to the top or bottom of the circle and **dY** set to the subscanline step. In this example the point table will have three entries.

Figure 2-3 Antialiasing an odd-diameter point.

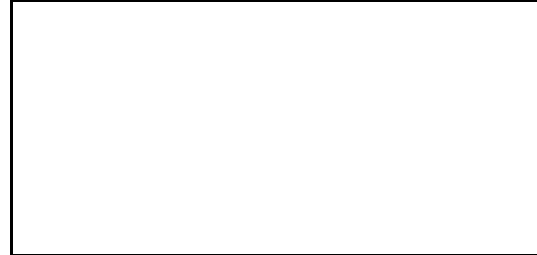


Note: in the case of an even diameter, the last of the required entries in the table is set to zero.

The *GLINT R4 Reference Guide* gives full details of how the point table is laid out.

Note that the table is configurable and point shapes other than circles can be rendered. Also if the **StartXDom** and **StartXSub** values are not coincident then horizontal thick lines with rounded ends, can be rendered.

The point looks like this and we will render from bottom to top. The origin is assumed to be bottom left and we are using 4x4 antialiasing quality. The point's diameter is 3 pixels, or 12 sub scanlines. The point table is assumed to be set up already.



Render Data Field					
AreaStippleEnable	0	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	1
AntialiasEnable	1	AntialiasingQuality	0	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	X	TextureEnable	1
FogEnable	1	CoverageEnable	1	SubPixelCorrectionEnable	0

- StartXDom (X)
- StartXSub (X)
- StartY (Y)
- dY (1.0/ 4.0)
- Count (12)
- Render
- FlushSpan ()

The SubPixelCorrection bit may be turned on to enable correction of the color, depth, fog and texture values at the start of a scanline.

8.2.7 Lines

There are two accepted way of drawing lines: using a DDA, or Bresenham's algorithm. Bresenham's algorithm has an advantage over DDAs because no divide is necessary. This has some benefits, particularly for 2D. For OpenGL we use the DDA method because the cost of the divide is acceptable and is needed to calculate the gradient of any color or depth change.

Lines are specified by their end points (accurate to 4 bits of sub pixel position) and rate of change in X and Y per step along the major axis of the line.

8.2.7.1 Aliased Lines (OpenGL and X)

Single pixel wide aliased lines are drawn using a DDA algorithm so all it needs by way of input data is StartX, StartY, dX, dY and length. The algorithm just calculates:

```
while (length-->0)
{
  X = X + dx
  Y = Y + dy
  plot ((int)X, (int)Y)
}
```

The variables X, Y, dx and dy are all fixed point numbers. The conversion to memory address using the X, Y coordinate is done in the memory read units.

8.2.7.2 Worked example - Aliased PolyLine (OpenGL)

A two segment polyline from (X_1, Y_1) to (X_2, Y_2) to (X_3, Y_3) is required. Both segments are X major, so:

$$\text{abs}(X_{n+1} - X_n) > \text{abs}(Y_{n+1} - Y_n)$$



Note: For individual line segments or the first line segment in a polyline the line stipple is reset (as shown).

Render Data Field					
AreaStipple Enable	0	LineStippleEnable	1	PrimitiveType	0
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiasEnable	0	AntialiasingQuality	X	ResetLineStipple	1
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	0

StartXDom (X_1)

dXDom (± 1.0)

StartY (Y_1)

dY ($(Y_2 - Y_1)/(X_2 - X_1)$)

Count ($\text{abs}(X_2 - X_1)$)

Render

dXDom (± 1.0)

dY ($(Y_3 - Y_2)/(X_3 - X_2)$)

ContinueNewLine ($\text{abs}(X_3 - X_2)$)

Note: The use of ContinueNewLine is not recommended for OpenGL because the DDA units will start with a slight error as compared with the value they would have been loaded with for the second and subsequent segments. The fractional bits of the DDA can be forced to zero or half on the ContinueNewLine action.

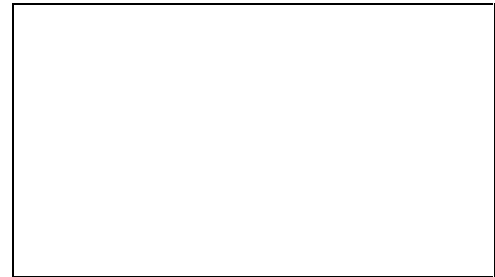
8.2.7.3 Aliased Wide Lines (OpenGL)

There is no direct support for wide lines. The OpenGL server has two options:

1. Wide lines can be drawn by repeating a single pixel wide line, but offset by one pixel in X for X major lines or one pixel in Y for Y major lines. Any values interpolated along the line (e.g. color) will need to be re-initialised at the start of each individual line. This is easily done with the Render command.
2. Wide lines can be converted to parallelograms (the ends of a wide line are parallel to the edge of the screen in OpenGL) and then rendered as polygons.

As you might expect neither method is the best in all cases. For vertical or near vertical lines method 2 will cause fewer page breaks in memory so should be faster. However if there is any stippling then method 1 is likely to be much faster. Method 1 is the simpler method and is the preferred implementation.

A single wide line from (X_1, Y_1) to (X_2, Y_2) is required. The line is 3 pixels wide. The line is X major so $\text{abs}(X_2 - X_1) > \text{abs}(Y_2 - Y_1)$.



Render Data Field					
AreaStippleEnable	0	LineStippleEnable	1	PrimitiveType	0
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	1
SyncOnBitMask	0	SyncOnHostData	X	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	0

StartXDom $(X_1 - 1)$
 dXDom (± 1.0)
 StartY (Y_1)
 dY $((Y_2 - Y_1)/(X_2 - X_1))$
 Count $(\text{abs}(X_2 - X_1))$
 Render
 StartXDom (X_1)

Render

StartXDom ($X_1 + 1$)

Render

8.2.7.4 Aliased Wide Lines (X)

Individual wide lines in X have square ends and multiple connected wide lines have a range of joint styles. X needs to convert the wide lines either to polygons, or to a series of spans, to achieve the desired effect.

8.2.7.5 Antialiased Lines (OpenGL)

Antialiased lines on R4 are supported via the *Gamma* OpenGL interface only - the onboard Delta unit does not implement antialiased lines.

Antialiased lines of any width are drawn as antialiased polygons (see below). If stipple is enabled then the line is drawn as a series of polygons to match up with the stipple parameters, using the *RepeatFactor*, *StippleMask* and *Mirror* fields in the **LineMode** or **LineStippleMode** registers. The line width is defined in the *AALineWidth* register as a floating point number.

8.2.8 Polygons

The only non-triangle polygons the rasteriser handles are screen aligned trapezoids. These are characterised by having the top and bottom edges parallel to the X axis. The side edges may be vertical but are more usually diagonal. The top or bottom edges can degenerate into points in which case we are left with flat topped or flat bottomed triangles. Any polygon can be decomposed into this shape, however the sample OpenGL server always decomposes polygons⁵ into triangles because the interpolation of values over non-triangular polygons is ill defined.

The rasteriser does handle vertical 'bow tie' polygons.



As part of the rasterisation process a number of parameters (color, depth, fog and texture) are calculated for each fragment generated. These are calculated in the DDA unit down stream under the guidance of the rasteriser step messages. The ideal way to calculate these values is to use the fragments XY coordinate and substitute this into the plane equation for each parameter in turn. This technique gives the best result, however it is computationally expensive so it is normal to use an incremental method such as a DDA to approximate to it. The DDA method introduces some errors of its own:

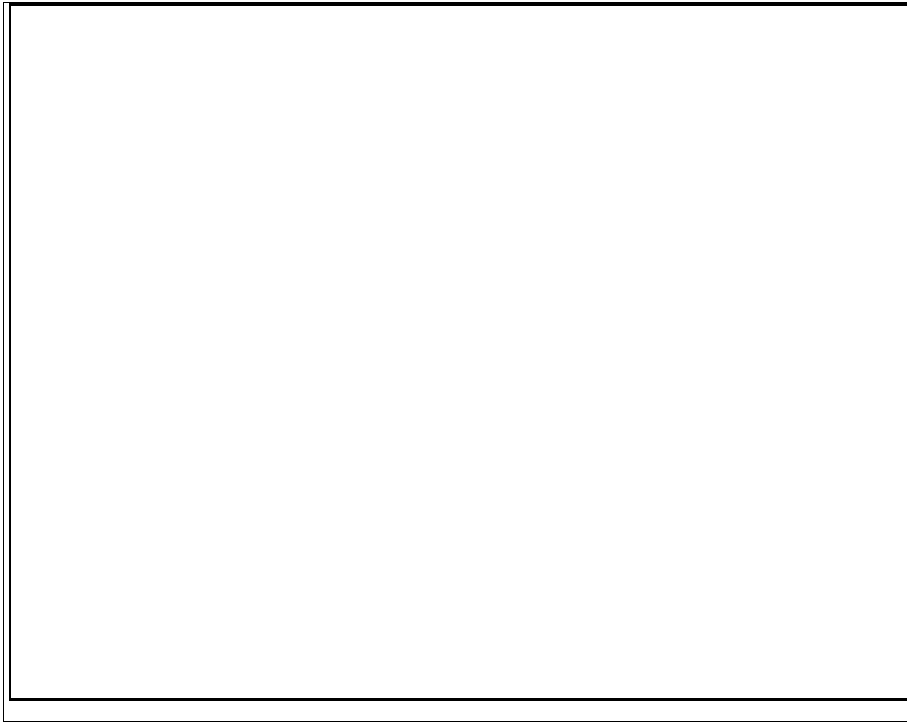
- An incremental error due to the finite precision of the delta values. To overcome this source of errors enough fractional bits are used so that the error cannot propagate into the actual bit range of the DDA where the parameter value is extracted from.
- The start value for a parameter, P, can be nearly dPdx (one step in the X direction) out because the value calculated as a result of a Y step (shown as a circle in the following

⁵Excluding the special case of screen aligned rectangles.

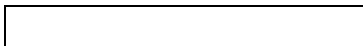
diagram) corresponds to the value of the sample on the edge and not at the center of the first fragment to be drawn.

It is necessary to correct for this error to eliminate bright edge artefacts and achieve high quality rendering.

This correction is needed for every scanline. A similar correction is needed at the start of the primitive because the parameter value at the start vertex is unlikely to lie on the horizontal center of a pixel so needs adjustment in Y. This correction is handled by software.



If $dErr$ is the distance the edge is away from the pixel's center (must be < 1) and $dPdx$ is the change in P for unit change in x then the correct value at the first sample point is:



The distance $dErr$ is sent internally by the rasteriser in `PrepareToRender` and `Step` messages. The multiplication is done in the DDA units whenever these messages are received, but only update P_x on the **SubPixelCorrection** register if the LS bit of the data field is set. The correction $dErr$ is sent as a 7 bit 2's complement 1.6 fixed point format. The $dErr$ value sent in the messages is the $dErr$ needed for the next scanline (of the first one in the case of a `PrepareToRender`).

8.2.8.1 Sub Pixel Correction not Supported for Antialiased Primitives

Sub pixel correction must be enabled by the `SubPixelCorrectionEnable` bit in the **Render** command if it is required. See the *GLINT R4 Reference Guide* for more information.

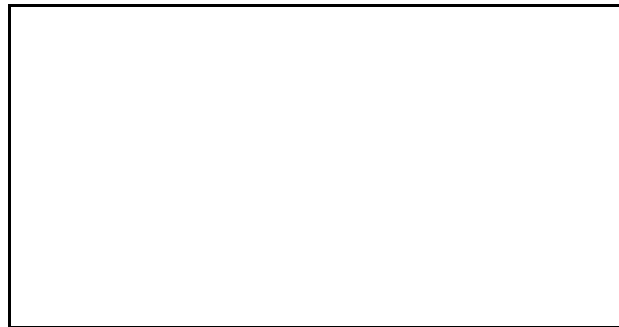
Antialiasing presents a much more complex problem to solve in that the sample point for the parameters must be inside the boundary of the fragment, but this may not be the center of the pixel anymore. Near horizontal edges can give rise to a $dErr$ value which approaches the width of the screen (or window). Two methods can be used to overcome this:

- The sample point can be moved to be within the boundary by 'micro nudging' the DDAs in X and Y.
- The parameter being interpolated can be integrated over the interior sub pixel sample points and then divided by the number of interior points (this is the method in the OpenGL spec).

In both these cases the changes to the DDA units are too extensive given the other problems antialiasing presents (the coverage calculation doesn't take into account sub pixel visibility and doesn't work well with a depth buffer). No sub pixel corrections are done for antialiased primitives.

8.2.8.2 Antialiased Triangle

The triangle looks like this and is rendered from top to bottom. The origin is assumed to be bottom left. Antialias quality is 4x4:



Render Data Field					
AreaStippleEnable	1	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiasEnable	1	AntialiasingQuality	0	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	1
FogEnable	1	CoverageEnable	1	SubPixelCorrectionEnable	1

```

StartXDom (X1)
dXDom ((X3- X1)/(4 * (Y3 - Y1)))
StartXSub (X1)
dXSub ((X2- X1)/(4 * (Y2 - Y1)))
StartY (Y1)
dY (-1.0/ 4.0)
Count ((Y1 - Y2) * 4)

Render
StartXSub (X2)
dXSub ((X3- X2)/(4 * (Y3 - Y2)))
ContinueNewSub ((Y2 - Y3) * 4)    // Bottom half
FlushSpan ()

```

Note: The DDA units need to have their sample point biased from the center of the pixel to the lower edge of the pixel so the DDA units can be tracked properly with the walk messages. This can be done by calculating the start values for integer Y values rather than at Y+0.5 as would normally be done.

The sub pixel correction is only needed if color, depth, fog or texture interpolation is being used.

8.3 Span Operations

Many 2D rendering operations can be implemented more efficiently using span operations, enabled with the *FastFillEnable* bit in **Render** and **Render2D**. For both 2D textures and rasterizer bit mask operations the improvement can be from about 40 Mpixels/s to 400 Mpixels/s.

GLINT R4's span filling implementation can be used for image upload, image download, filling with constant color, filling with a pattern, characters (i.e. bit masks), copies, and copies with logical ops. Any trapezoid can be used and the scanning direction can be left-to-right or right-to-left. Benefits of span fill for 2D operations include:

- Better utilization of SGRAM block fill (where memory devices permit) for solid, stippled and patterned fills and character bitmaps.
- Span mechanism is independent of pixel size – makes maximum use of framebuffer bandwidth for 8, 16 and 32 bit pixels.
- Multiple pixels processed in parallel
- No alignment restrictions – any span operation may be performed to any pixel alignment for all pixel sizes.
- Page break overheads are spread over many more read/write operations during a BitBlit operation – performance of BitBlits is much closer to peak memory bandwidth
- Both window- and screen-relative operations supported
- Scissor clipping can be used in conjunction with span operations

If any reads are enabled, span operations are converted into a series of normal memory reads. The memory data returned is aligned and sent on in 64 bit words for further processing.

Note: This differs from earlier chips where the memory interface was responsible for decoding the span mask and returning the appropriate aligned data.

Span reads are only supported when the pixel data is laid out in the Linear or Patch64 formats. 32_2 and patch_2 formats do not support spans (but packed support is available for non-span rasterization - see **Packed8Pixels** and **Packed16Pixels** in the *GLINT R4 Reference Guide*.)

If source and destination reads are enabled then the source data is read first and stored in the scratch pad ram. Then the destination data is read and packed into 64 bit words and sent on. After each destination word is sent the corresponding source word from the scratch pad ram is read and sent on. The destination buffers are read in increasing numerical order.

Page breaks are kept to a minimum by reading all the data from a buffer for a span before moving onto the next buffer (for the same span mask).

The span operation does have some restrictions:

- Stencil and Depth tests are not available. These units just ignore the commands associated with fast span fills.

- Gouraud shading, alpha tests, alpha blend and dither operations are not available.
- If GIDs are being used for window clipping then spans cannot be used at full speed as they normally ignore GID information and write to all pixels in the span. However the result writes 4 pixels per cycle.

When span operation is enabled the rasterizer divides the pixels between the left- and right-hand edges of the polygon or rectangle into a succession of spans, each 64 pixels wide.

Each span is described by a 64 bit wide span mask and each pixel in the span has a corresponding bit in the span mask. If a bit in the span mask is set, then the corresponding pixel will be read and/or written. The least significant bit in the span mask (bit 0) corresponds to the leftmost pixel on the screen for the span.

The span mask does not have any fixed alignment with the pixels stored in the framebuffer, i.e. the first pixel in the span may correspond to any pixel in the framebuffer. Any masking or shifting to align the span data being read or written to the 64 bit framebuffer architecture is performed automatically.

Span filling may be performed left-to-right or right-to-left, but the pixels within an individual span are always read and/or written left to right. Hence if bitmask or image download data is provided, the data within individual spans must be ordered left to right. Normally if any data is provided span filling should be left-to-right.

The use of spans for image handling is shown later (Bitmaps, Spans and Images).

Spans operate in both the LB and FB functional groups. In the Localbuffer the data written is constant for the span and is held in the **LBClearDataU** and **LBClearDataL** registers which together provide 40 bits of data. This is replicated automatically to the four pixels in a memory word. For Packed16 mode where there are 8 pixels in a memory word software must replicate the 16 bits of clear data into the 32 bit **LBClearDataL** register. The **LBClearData** registers hold the depth, stencil and GID data in the format it is in the local buffer - i.e. no formatting is done on the clear data before it is written.

The byte enables (in LBWriteMode) can be used to protect bytes from being updated. If, however, the field to clear is not byte aligned and a multiple of bytes in width (i.e. a 3 bit stencil field), then clearing this field while leaving the others intact can only be done via a read-modify-write operation so will run at one quarter the speed.

8.3.1 Mode changes in Span Operations

GLINT R4 supports major mode changes during native display list operations. This is described in greater detail in the *Framebuffer* chapter. However to ensure that the effect of mode changes during display lists can be software controlled, two registers (**FBDestReadEnables** and **FBSourceReadEnables**) are set up to provide monitoring and readback for software-specified modes e.g. AlphaBlend or LogicalOps.

The Boolean equation for a span read in buffer n is:

$$\begin{aligned} \text{destRead} = & (\text{mode.ReadEnable} \& \text{mode.Enable}[n] \& \sim\text{mode.UseReadEnables}) | \\ & (\text{mode.ReadEnable} \& \text{mode.Enable}[n] \& \text{mode.UseReadEnables} \& \\ & (\text{E4} \& \text{R4} | \text{E5} \& \text{R5} | \text{E6} \& \text{R6} | \text{E7} \& \text{R7})) \end{aligned}$$

where "mode" is shorthand for **FBDestReadMode** and E* and R* are taken from **FBDestReadEnables**. The logical operations versions of the registers

(**FBDestReadEnablesAnd** and **FBDestReadEnablesOr**) can be used to change individual bits.

8.3.2 Alpha Filtering

One use of the mode monitoring feature is an alpha filtering enhancement. In many cases when doing alpha blending the blend mode is set such that if the fragment's alpha is a specific value (typically 0 or 255) then the framebuffer color (from a destination read) is effectively ignored as it doesn't contribute to the final alpha blended color. In this case there is no point in reading the destination pixel value and we can save memory bandwidth by avoiding it.

Alpha filtering is enabled by the *AlphaFiltering* bit in **FBDestReadMode** and the reference alpha value to compare against can be found in **FBDestReadEnables**.

8.3.3 Span Mask Processing

Span fills are enabled by setting the *FastFillEnable* bit in the **Render** command. The *SpanOperation* bit when clear indicates writes are to use the constant color found in the previous **FBBlockColor** register. When this bit is set write data is variable and is either provided by the host (i.e. **SyncOnHostData** is set) or is read from the framebuffer. All other trapezoid parameters are the same.

The span mask can also be used to grow the extent region or perform picking as part of HostOut statistics gathering.

The span mask undergoes several processing steps before it is used by the Framebuffer Unit to determine which pixel to read and/or write:

- The Rasterizer generates the mask using the left and right hand edge information. Note that the edges may be vertical or sloped.
- If *SyncOnBitMask* is enabled in the **Render** command, then the span mask is ANDed with the bit mask data provided by the host. If no bit mask data is present the Rasterizer wait for it to arrive before proceeding.
- The bit mask data may be optionally inverted, byte swapped, word swapped or mirrored (in any combination) before the ANDing is performed. The inversion can be used to enable drawing of the background bits. The byte and word swapping allows bit mask data from different endian hosts to be accommodated. The mirror operation swaps bits 0 and 31, bits 1 and 30, etc. which changes the left most pixel in a span from being controlled by the least significant bit to the most significant bit in the bit mask.
- If Screen Scissor testing is enabled then pixels falling outside the left and right edges of the screen scissor region have their corresponding bits in the span mask cleared.
- If the User Scissor test is enabled, then pixels falling outside the left and right edges of the user scissor region have their corresponding bits in the span mask cleared.
- If Area Stippling is enabled, then the stipple mask is extracted from the area stipple table for the appropriate scan line and expanded, if necessary, to 32 bits by replication. The normal offset, select and mirror controls in X and in Y may be used as for non-span rendering. The stipple mask is ANDed with the span mask.
- If Texture Mapping is enabled, then a texel is read from the texture logical or physical address under the control of the **TextureCoordMode**, **TextureOperation**, **LogicalTexturePage**, **TextureReadMode** and the S, T and Q DDA parameters. If the

texel is to be used as a bit mask, then any specified texel formatting is performed and the final 64 bit texel value is optionally inverted, byte swapped and mirrored before being ANDed with the span mask.

- The span mask is now used to read/write the framestore pixel data

8.3.4 Block Write

The *FastFillIncrement* and *BlockWidth* parameters in **Render** and **FBWriteMode** are no longer required or supported. For more information on block write see Volume I, section 4.2.5, Clearing the Localbuffer using **FBWrite**.

8.3.5 Pixel Sizes

The local buffer holds up to four fields of information: Depth, Stencil, GID and fast clear planes (FCP). FCPs are not implemented in GLINT R4 and the bitfields are reserved for historical reasons.

R4 allows pixel sizing on a unit-by-unit basis, which can be desirable for texturing. When using span operations it is important to maximize the number of pixels per 32 or 64 bits processed, The Rasterizer unit **PixelSize** register can have the following values on either a global or unit-tailored basis:

:

- Depth: 15, 16, 24 and 31.
- Stencil: 0, 1, 2, 3, 4, 5, 6, 7 and 8.
- GID: 0, 1, 2, 3 and 4.

The depth plane always starts at bit 0. The Stencil and GID fields can start on any bit position from 16 to 39 inclusive. It is the user's responsibility to ensure that they don't overlay or reference bits outside the pixel width.

Selecting a depth width of 15 bits forces the stencil and GID fields to be set from bit 15 of the pixel and ignores the normal stencil and GID settings. If the specified width of a field is less than its internal width then the field is zero extended at the Least Significant end to its internal width.

Since **PixelSize** is a core register it can be modified at any time without affecting in-progress rendering. It is not necessary to synchronize with the chip before changing pixel depth.

Pixel size is also definable in the **DMARectangleRead** and **DMARectangleWrite** registers.

8.3.5.1 Sub Pixel Precision

The rasterizer has 16 bits of fraction precision and the screen width used is typically less than 2^{16} wide, so a number of bits (called subpixel precision bits) are available.

Consider a screen width of 4096 pixels. This figure gives a subpixel precision of 4 bits ($4096=2^{12}$). The extra bits are required for a number of reasons:

- antialiasing (where vertex start positions can be supplied to subpixel precision)
- when using an accumulation buffer (where scans are rendered multiple times with jittered input vertices)
- for correct interpolation of parameters to give high quality shading as described below

8.3.6 Bitmaps, Spans and Images

The GLINT R4 is not software-compatible with earlier MX chips. Specific changes affecting bitmaps, spans and images include separate control of source and destination FB and LB reads using new registers, automatic span read alignment, pattern RAM data held in localbuffer, and texture units now generate source offsets but not addresses.

8.3.6.1 Bitmaps

A Bitmap primitive is a trapezoid or line of ones and zeros which controls which fragments are generated by the rasterizer. The bitmap operates on any fragments produced by the rasterizer, including spans and characters.

Bitmaps may be implemented as Rasterizer bitmasks or 2D Textures with or without span fill enabled. Span Fills are described in the next section. Span fills are generally an order of magnitude faster but do not normally support LB test functions (Depth, GID, Stencil) or Alpha Test, Logical Ops, Texturing or Dither. (But see Volume I, Section 1.1.6 - GID Field - for GID testing of LB spans.)

Bitmaps are controlled using the **BitMaskPattern** register and parameters enabled in the **RasterizerMode** command: *ByteSwapBitMask*; *MirrorBitMask*; *InvertBitMask*; *BitMaskPacking* and *BitMaskOffset*. In addition to its raw data, each bitmap is characterised by its origin coordinates (bottom left or top left); width and height.

When *SyncOnHost* is enabled in the **Render** command only fragments where the corresponding Bitmap bit is set are submitted for drawing. The normal use for this is in drawing characters, although the mechanism is available for all primitives. Bitmap data unless otherwise formatted is by default packed contiguously into 32 bit words so that rows are packed adjacent to each other. Bits in the mask word are by default used from the least significant end towards the most significant end and are applied to pixels in the order they are generated in. The relationship between bits in the mask and the scanning order is shown in Figure 8-5.

The rasterizer scans through the bits in each word of the Bitmap data and increments the X,Y coordinates to trace out the rectangle of the given width and height. By default, any set bits (1) in the Bitmap cause a fragment to be generated, any reset bits (0) cause the fragment to be rejected.


```

Render.PrimitiveType = GLINT R4_TRAPEZOID_PRIMITIVE
render.SyncOnBitMask = PERMDIA3_TRUE
// Issue render command. First fragment will be
// generated on receipt of the BitMaskPattern
Render (render)
// 8x5 pixel bitmap requires 40 bits, and so 2
// 32 bit words.
BitMaskPattern (patternWord0)
BitMaskPattern (patternWord1)

```

Rendering starts as soon as the first `patternWord` is loaded into the **BitMaskPattern** register.

R4 provides the ability to start a scanline at an arbitrary offset into the first bitmask that is downloaded for each scanline, and to discard unused bits at the end of a scanline. This lets the host download data directly from a host bitmap without having to shift and pack the bits. This functionality is controlled by the *BitMaskPacking* and the five *BitMaskOffset* bits in the **RasterizerMode** register.

8.3.6.2 Bitmaps with Spans

The fastest way to render downloaded bitmap data is to use a span operation (described in §2.1.6, Span Operations, above). The rasterizer is set up as normal and the *FastFillEnable* bit in the Render command is enabled. The *SpanOperation* bit determines if the span writes use constant color data or variable color data. All other trapezoid parameters are the same.

A span is always 64 pixels long and any combination of pixels within the span can be read and/or written. Pixels with a width of 8 or 16 bits are processed 8 or 4 pixels at a time respectively and all read and write alignment is handled in hardware. The span mechanism can be used for image upload, image download, filling with constant color, filling with a pattern, characters (i.e. bit masks), copies and copies with logical ops. Any trapezoid can be used and the scanning direction can be left-to-right or right-to-left⁶.

If the span is being written with a constant color value⁷ and the SGRAM supports block fills (where a number of pixels can be written simultaneously) then span filling automatically uses this mode of operation to give a very much faster filling rate.

The Memory Controller takes care of mapping this logical configuration on to the actual SGRAM configuration where the SGRAM chips may have fewer pixels in a block, the framebuffer may be interleaved and/or hold packed pixels.

When the bitmap data is downloaded it is ANDed with the span mask generated by the rasterizer. The resulting mask is passed through the core to be used as the block fill mask. Thus a single memory access can be used to process up to 32 pixels.

⁶The pixels within a span are always read and/or written in a left to right order so if the host is providing any bitmask or image download data then it needs to take this into account. The simplest thing is for the host to always scan left to right when supplying data.

⁷This is not strictly true as the framebuffer may be in packed pixel format so adjacent pixels within a 32/64 bit word could have different colors.

Since the downloaded bitmask data will be ANDed with masks generated by the Rasterizer without any re-alignment being performed, the host software must ensure that the masks match up. This can be achieved in either of two ways:

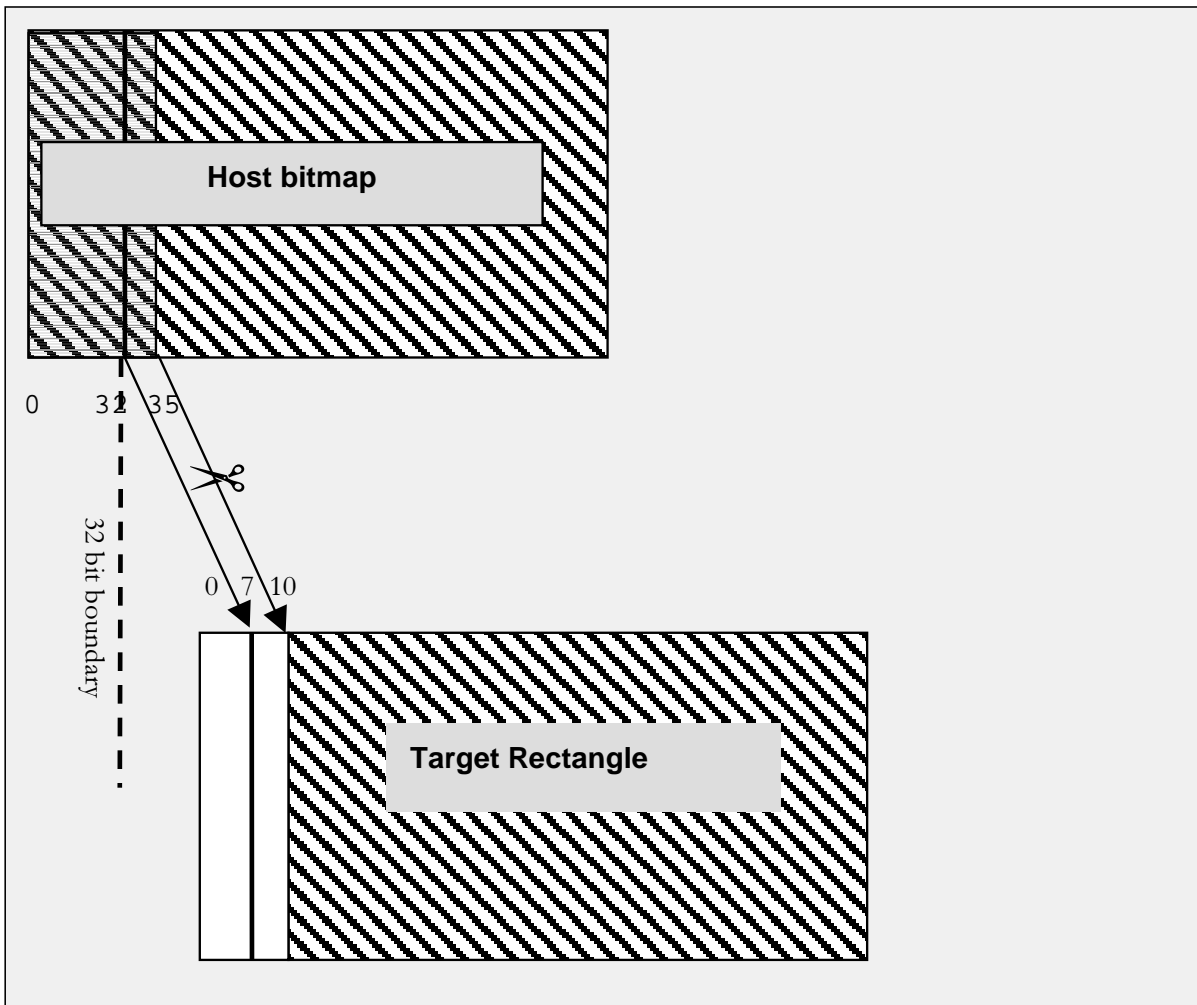
1. the host software can align the bits that it downloads to match the alignment of the Rasterizer.
2. use the User Scissor (generally faster and recommended).

Note: this is a general algorithm. In the special case where the data to be downloaded is already aligned to 32 bits on both the left and right edges the scissor need not be used.

For example, suppose we want to download data to fill a rectangle with left edge at 10 and right edge at 200. Assume that the host bitmap data is to be loaded from an offset of 35 within the bitmap. Our goal is to match the bit at offset 35 with the pixel at offset 10.

Since we want to avoid shifting the data and incurring a host processing overhead, we download the host bitmap data at the previous 32-bit boundary. This means that we must set GLINT R4 up to discard the first 3 bits of data.

We achieve this by rasterizing a rectangle whose left edge is 3 pixels less than that required, in



this case we would rasterize the left edge to start at pixel 7. This aligns the source bitmap data with the mask data produced by the rasterizer. But, in order to protect the 3 pixels that we would otherwise overwrite, we use the scissor clip and set its bounds to be those of the original rectangle.

When using a span operation like this the rasterizer waits for new bitmask data to be downloaded at the start of each scanline. So we do not have to perform the alignment operation on the right hand edge.

The following gives the outline for this algorithm:

```

leftalign = bitmapxleft & 31
width = Xright - Xleft + leftalign
StartXDom ((Xleft - leftalign)<<16)
dXDom (0)
StartXSub (Xright<<16)
    
```



```

StartY (Y<<16)
dY (1<<16)
Count (height)

// protect the edge pixels with the scissor

minXY.X = Xleft
minXY.Y = Y
maxXY.X = Xright
maxXY.Y = Y + height
ScissorMinXY(minXY)          // Load the registers
ScissorMaxXY(maxXY)

// Enable the unit
scissorMode.UserScissorEnable = GLINT R4_ENABLE
scissorMode.ScreenScissorEnable = GLINT R4_ENABLE

// At least the following bits require setting for
// the Render command.

Render.PrimitiveType = GLINT R4_TRAPEZOID_PRIMITIVE
render.SyncOnBitMask = GLINT R4_TRUE
render.FastFillEnable = GLINT R4_TRUE

// Issue render command. First fragment will be
// generated on receipt of the BitMaskPattern.

Render (render)

// download the bits from the source bitmap 32 bits
// at a time aligning the bitmap pointer at the
// start of each scanline

BitmapBase += bitmapyorg * bitmapwidth
bitmapxleft &= ~31
for (h = 0; h < height; ++h) {
    pulBitmap = BitmapBase + bitmapxleft/8;

```

```

    for (c = 0; c < width; c += 32) {
        BitMaskPattern(pulBitmap)
        pulBitmap += sizeof(ULONG)
    }
    BitmapBase += bitmapwidth
}

```

8.3.6.3 Glyphs

A byte stream of glyph data (packed four to a word) can be downloaded and automatically chopped up and padded to the necessary width for the texture units to use as a bitmap. For example a glyph with a width between 17 and 24 pixels will be sent down as a stream of bytes and each triplet of bytes will be padded with zero and sent to be written into memory. If the input words have their bytes labelled:

First word: DCBA (A is the least significant byte)

Second word: HGFE

Then the output words sent on to the rasterizer are:

First word: 0CBA

Second word: 0FED

8.3.6.4 Image Copy/Upload/Download

R4 supports three “pixel rectangle” operations – Copy, Upload and Download.

Image operations involve rectangular regions with pixel coordinates rather than the usual 3D coordinates. The image regions can be moved among host memory and any GLINT R4 buffer(s).

8.3.6.5 Copy

Image Copy moves raw blocks of data around buffers. To zoom or re-format data external software must upload the data, process and return it.

To copy a rectangular area the rasterizer would be configured to render the destination rectangle, thus generating fragments for the area to be copied.

Note: Care must be taken when the source and destination overlap to choose the source scanning direction so that the overlapping area is not overwritten before it has been moved. This may be done by swapping the values written to the StartXDom and StartXSub, or by changing the sign of dY and setting StartY to be the opposite side of the rectangle.

If the source and destination rectangles overlap then the direction of the scan conversion is important and must be set up correctly by the host. Localbuffer copy operations are tested for pixel ownership (GID). Note that this implies two reads of the localbuffer, one to collect the source data, and one to get the destination GID for the pixel ownership test.

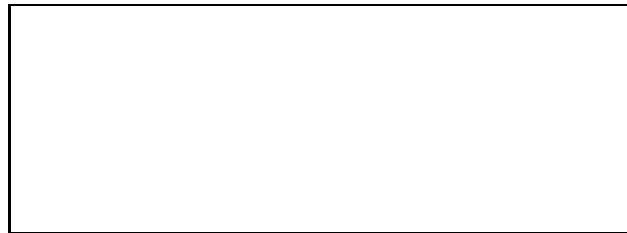
8.3.6.6 Upload/Download

The host places a pixel image in a windows-relative rectangle, in any buffer (depth, stencil or color) using the Rasterizer.

The host could control the process directly, but the rasterizer also manages clipping, fragment processing and window coordinate tracking.

During download, for example, the rasterizer scans the image so the host does not need to provide X,Y coordinates, waits for a depth, stencil or color command from the host, then processes the next pixel. In other words, the process is synchronous with host processing. To maintain synchronization enable the *SyncOnHost* bit in the **Render** command.

The image download rectangle looks like this and the origin is assumed to be bottom left. The host provides the data in top to bottom, left to right order. Color data will be provided. There are *n* pixels in the rectangle.



Render Data Field					
AreaStippleEnable	0	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiasEnable	0	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	1	TextureEnable	0
FogEnable	0	CoverageEnable	0	SubPixelCorrectionEnable	0

In OpenGL the AreaStippleEnable would always be 0, but in X may be enabled or disabled.

```

StartXDom (X1)
dXDom (0)
StartXSub (X2)
dXSub (0)
StartY (Y2)
dY (-1.0)
Count (Y2 - Y1 + 1)      // Width of image
Render
Color (P0)    // Pixel 0
Color (P1)
Color (P2)
...
...
Color (Pn)
    
```

Note: the rasteriser overscans the rectangle because the right hand edge is not plotted and the downloaded image doesn't include these pixels

Any functions which can generate fragment values, the color DDA for example, should generally be disabled for any copy, upload or download operations.

Warning: During image upload, all the returned fragments must be read from the Host Out FIFO, otherwise the GLINT R4 pipeline will stall. In addition it is strongly recommended that any units which can discard fragments (for instance the following tests: bitmask, alpha, user scissor, screen scissor, stipple, pixel ownership, depth, stencil), are disabled otherwise a shortfall in pixels returned may occur, also leading to deadlock.

Bit mask processing can be used in conjunction with image operations to allow arbitrary stipples, for example. Use the **BitMaskPattern** command to load the bit mask.

Unlike conventional bit mask functionality, during image loading the **Bitmaskpattern** command must be interleaved accurately with the image data to ensure that the new mask is available immediately the old mask is consumed. Pixels arriving without mask bits are considered passive until the new mask arrives.

If the host fails to supply a required color, depth or stencil tag the chip waits until one arrives, or (to avoid unnecessary hangs) terminates the image operation when any tag other than color, depth, stencil, FBData or BitMaskPattern are received.

During image uploads the host can read back a window-relative rectangle from any buffer. The buffer read must be set up using the **FBSourceReadAddress**, Offset and Operations registers.

8.3.6.7 Image Copy/Upload/Download with Spans

2D image operations to and from the framebuffer can be optimized by using a span operation. The benefits are greatest at lower pixel depths since packed pixel data is transferred through the core.

Copy

Using span operations when copying pixel data within the framebuffer is straightforward. Simply set the *FastFillEnable* and *SpanOperation* bits in the **Render** command.

Note: This works both with and without logical op processing.

Download

Download facilities ("Write Pixels") allow the host to transfer image data to local memory. The rasteriser supports this function by scan converting the rectangle (so the host doesn't need to generate X, Y coordinates). The rasteriser is constrained by the *SyncOnHostData* bit in the **Render** command to wait for Depth, Stencil or Color data from the host (in the **Depth**, **Stencil** or **Color** registers) before moving on to the next pixel. In other words it runs synchronously to the host for the duration of this primitive.

The bit mask mode can also be enabled during this function so arbitrary stippling can be done on the image being downloaded (useful in X). The bit mask register is loaded whenever the **BitMaskPattern** register is received. This is slightly different⁸ to the way it works when the rasteriser is not in Image download mode. The **BitMaskPattern** data must be interleaved

⁸This change is necessary to prevent a deadlock situation arising if too many **Color** messages (for example) are sent before the next **BitMask** message is due.

correctly with the image data to ensure the new mask is available immediately after the last bit in the current mask has been used. If this sequence is not correct then all subsequent fragments until the new mask is received will be passive.

There is the potential for the host to send too few Color (**Depth**, **Stencil** or **FBData**) messages for the size of primitive it has defined. Rather than have GLINT R4 hang because it is waiting for messages which will never arrive, any message other than Color, **Depth**, **Stencil**, **FBData** or **BitMaskPattern** stop primitive generation.

The *SyncOnHost* functionality is in fact available for any primitive, although usually used in conjunction with downloads.

Image downloads are also supported by DMA - see **DMARectangleRead** in the *GLINT R4 Reference Manual*

Upload

Image upload ("ReadPixels"). This function provides the host with a method of reading back a window-relative rectangular region of any of the buffers (depth, stencil, color). The rasteriser supports this function by scan converting the rectangle and sending the active walk messages. The Local Buffer Read Unit or the Framebuffer Read Unit will have already been **set** up to do the read and generate the appropriate **LBDepth**, **LBStencil** or **FBColor** message, which will be collected by the Host Out Unit and passed back to the host.

Upload can also be run via DMA using the **DMARectangleWrite** command. The image data may be a sub image of a larger image and have any natural alignment or pixel size. Information regarding the rectangle transfer is held in registers loaded from the input FIFO or a DMA buffer.

Note: failure to supply an EOF may have unpredictable results.

The pixel data written to host memory is always packed, however when read from the Host Out FIFO it can be in packed or unpacked format (packed when Reset). It can also, optionally, be aligned on 64 byte boundaries. The minimum number of PCI writes are used to align and pack the image data.

GLINT R4 is set up to rasterize the source area for the pixel data (depth, stencil, color, etc.) enabled in the **Render** command. This is done before the Rectangular DMA is started.

8.4 Rasterizer Mode

The RasterizerMode register sets long-term modes, particularly these:

- *MirrorBitMask*: This is a single bit flag which specifies the direction that bits are checked in the **BitMaskPattern** register. If the bit is reset, the direction is from least significant to most significant (bit 0 to bit 31), if the bit is set, it is from most significant to least significant (from bit 31 to bit 0). Using a value of 3 is very useful in conjunction with the *MirrorBitMask* bit for handling Microsoft Windows bitmaps since this causes a complete byte swap of the downloaded data.
- *InvertBitMask*: This is a single bit which controls the sense of the accept/reject test when using a Bitmask. If the bit is reset then when the *BitMask* bit is set the fragment is accepted and when it is reset the fragment is rejected. When the bit is set the sense of the test is reversed.
- *BitMaskPacking*: This is a single bit which controls the packing of bits which are downloaded as part of a *SyncOnBitMask* operation. If this bit is reset then any spare bits at the end of a scanline are used to start the next scanline. If this bit is set then extra bits at the end of a scanline are discarded. This is not available for use with span fills.
- *BitMaskOffset*: This is a 5 bit field which specifies the first bit to be used in the first bitmask word of every scanline downloaded as part of a *SyncOnBitMask* operation. This is not available for use with span fills.
- *Fraction Adjust*: These 2 bits control the action taken by the rasterizer on receiving a **ContinueNewLine** command. As GLINT R4 uses a DDA algorithm to render lines, an error accumulates in the DDA value. GLINT R4 provides for greater control of the error by:
 1. leaving the DDA running, which means errors will be propagated along a line, or
 2. setting the fraction bits to either zero, a half or almost a half (0x7FFF).
- *Bias Coordinates* is a 2-bit field with the following actions:
 - 0 – Add 0 to the coordinates (Effectively do nothing)
 - 1 – Add exactly one half to the coordinates
 - 2 – Add nearly one half (0x7FFF) to the coordinates
- *Host Data Byte Swap Mode*: The data downloaded by the host when using *SyncOnHostData* can have its bytes re-ordered. If the downloaded data has a byte ordering of ABCD then, this 2 bit field specifies re-ordering as follows:
 - 0: ABCD (no swap)
 - 1: BADC (swap within halfwords)
 - 2: CDAB (halfword swap)
 - 3: DCBA (full byte swap)
- *Y Limits Clipping*: When set, this bit enables Y Limits clipping. When reset Y Limits clipping is disabled. This is described in the next section.
- *Multi Rasterizer*: If set this bit causes the rasterizer to work in multi-Rasterizer mode. If reset the rasterizer works in single Rasterizer mode.

8.4.1.1 Y Limits Clipping

The rasterizer normally rasterizes all pixels on every scanline, generating a fragment per pixel. If large numbers of scanlines are subsequently clipped out by, for example, one of the scissor units, then a lot of time can be wasted. The **Ylimits** register has been added to provide a way of quickly eliminating whole scanlines for a given primitive. This is effectively a Y scissor clip in the Rasterizer.

If Y limits testing has been enabled in the **RaserizerMode** register, and if a scanline being rasterized falls outside the Y limits bounds, then the rasterizer will move directly onto the next scanline without rasterizing in X.

- Y Limits clipping is automatically disabled when *SyncOnHostData* or *SyncOnBitMask* is used.

8.4.2 Multi-rasterizer Operation

R4 is specifically designed for multi-rasterizer operation behind a geometry accelerator. Typically, two R4s are used to rasterize the output from one *Gamma* chip.

To support multi-rasterizer operation, R4 implements:

- a display striping bus to carry video data from one R4 to the next
- video and RAMDAC enhancements
- pixel-accurate genlock.

The setup procedure is described in Volume I, section 5.2.4, *Multi-rasterizer Setup*

8.4.3 Rasterizer Unit Registers

Real coordinates with fractional parts are provided to the rasterizer in 2's complement 16 bit integer, 16 bit fraction format, as illustrated below for a typical register in this unit:

Name	Type	Offset	Format		
ContinueNewDom	Rasterizer Command	0x8048	Integer		
Bits	Name	Read	Write	Reset	Description
0...15	Scanlines	✓	✓	x	16 bit unsigned integer
16...31	Reserved	0	0	x	Reserved for future use, mask to 0

Table 1.1 Typical register description – ContinueNewDom

8.4.3.1 Command registers

The following table lists the command registers which control the rasterizer unit. The control registers are shown separately below.

Register Name	Data Field	Description
Render	Bitfield	Starts the rasterization process
ContinueNewDom	16 bit integer	Allows the rasterization to continue with a new dominant edge. The dominant edge DDA in the rasterizer is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex polygon to be broken down into a collection of trapezoids, with continuity maintained across boundaries. Note: other DDAs are not reloaded with new start values until the next Render command. Thus it is not possible to use this command, for example, to Gouraud shade a triangle from left to right which has a knee on the left hand side. To avoid this, 3D rendering should always start from the side without the knee. The data field holds the number of scanlines (or sub scanlines) to fill. This count is not loaded into the <i>Count</i> register.
ContinueNewSub	16 bit integer	Allows the rasterization to continue with a new subordinate edge. The subordinate DDA is reloaded with the new parameters. The dominant edge is carried on from the previous trapezoid. This is useful when scan converting triangles with a 'knee' (i.e. two subordinate edges). The data field holds the number of scanlines (or sub scanlines) to fill. This count is not loaded into the Count register.
Continue	16 bit integer	Allows the rasterization to continue after new delta value(s) have been loaded, but does not cause either of the trapezoid's edge DDAs to be reloaded. The data field holds the number of scanlines (or sub scanlines) to fill. This count is not loaded into the Count register.
ContinueNewLine	16 bit integer	Allows rasterization to continue for the next segment in a polyline. The XY position is carried on from the previous line, but the fraction bits in the DDAs can be: kept, set to zero, half, or nearly one half, under control of the RasterizerMode. The data field holds the number of pixels or subpixels in a line. This count is not loaded into the Count register. The use of ContinueNewLine is not recommended in OpenGL as for the second and subsequent segments the DDA units will start with a slight error compared with the value they would have been loaded with.
FlushSpan	Not used	Used when antialiasing to force the last span out when not all sub spans may be defined.
PixelSize	0 = 32 bits 1 = 16 bits 2 = 8 bits	Configures the Rasterizer (and other core units) with the size of pixel to process when spans are used. It also informs the framebuffer interface Unit, but in this case all reads and writes are affected and not just spans. This replaces the pixel size field in the PCI FBModeSel register and works the same way for single pixel reads and writes (i.e. the framebuffer can be set to 32 bit pixels even though it is displaying 8 bit pixels to process 4 pixels at a time).

Register Name	Data Field	Description
WaitFor Completion	Not used	This is used to suspend the core until all outstanding reads and writes in both the localbuffer and framebuffer memory units have completed. This is intended to prevent a new primitive from starting to be rasterized before the previous primitive is completely finished. It would be used, for example, to separate texture downloads from the surrounding primitives. The same functionality can be achieved using the Sync register and waiting for it in the Host Out FIFO; however, this method doesn't involve the host and can be inserted into a DMA buffer.

Table 8.1 Command Register Descriptions

Register Name	Data Field	Description
RasterizerMode	See below	Defines the long term mode of operation of the rasterizer.
StartXDom	Fixed point 16.16 format	Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing.
dXDom	Fixed point 16.16 format	Value added when moving from one scanline (or sub scanline) to the next for the dominant edge in trapezoid filling. Also holds the change in X when plotting lines so for Y major lines this will be some fraction (dx/dy), otherwise it is normally ± 1.0 , depending on the required scanning direction.
StartXSub	Fixed point 16.16 format	Initial X value for the subordinate edge.
dXSub	Fixed point 16.16 format	Value added when moving from one scanline (or sub scanline) to the next for the subordinate edge in trapezoid filling.
StartY	Fixed point 16.16 format	Initial scanline (or sub scanline) in trapezoid filling, or initial Y position for line drawing.
dY	Fixed point 16.16 format	Value added to Y to move from one scanline to the next. For X major lines this will be some fraction (dy/dx), otherwise it is normally ± 1.0 , depending on the required scanning direction.
Count	16 bit integer	Number of pixels in a line. Number of scanlines in a trapezoid. Number of sub scanlines in an antialiased trapezoid. Diameter of a point in sub scanlines.
BitMaskPattern	32 bits defined earlier	Value used to control the BitMask stipple operation (if enabled).
PointTable0 PointTable1 PointTable2 PointTable3	Packed dx point data.	Antialias point data table. There are 4 words in the table and the register tag is decoded to select a word.
ScanLine Ownership	See Multi-Rasterizer chapter	Defines which scanlines are owned when in multi-rasterizer mode.
Ylimits	Ymax: 2's complement 16 bit value in the upper word. Ymin: 2's complement 16 bit value in the lower word.	Defines the Y extents the rasterizer should fill between. A scanline is filled if its Y value satisfies $Y_{min} \leq Y < Y_{max}$

Table 8.2 Rasterizer Registers

8.4.4 Render Command

For efficiency, the Render command register has a number of bit fields that can be set or cleared per render operation and which qualify other state information. These bits are:

- *AreaStippleEnable*
- *LineStippleEnable*

- *ResetLineStipple*
- *TextureEnable*
- *FogEnable*
- *CoverageEnable*
- *SubpixelCorrection*.

This feature enables units to be set or cleared in one step as part of a specific render operation. For example, to clear a window to a background color when stippling and fog have already been enabled for 3D operations it is not necessary to clear the enable bits in **FogMode**, **AreaStippleMode** and **LineStippleMode** individually. They can be left enabled but overridden for the window clear operation simply by adjusting the **Render** command bitfield settings, shown below:

Render

Name	Type	Offset	Format		
Render	Global	0x8038	Bitfield		
<i>Command</i>					
Bits	Name	Read	Write	Reset	Description
0	AreaStipple Enable	✗	✓	x	This bit, when set, enables area stippling of the fragments produced during rasterisation in the Stipple Unit. Note that area stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no area stippling occurs irrespective of the setting of the area stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no area stippling for this primitive.
1	LineStipple Enable	✗	✓	x	This bit, when set, enables line stippling of the fragments produced during rasterisation in the Stipple Unit. Note that line stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no line stippling occurs irrespective of the setting of the line stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no line stippling for this primitive.
2	ResetLine Stipple	✗	✓	x	This bit, when set, causes the line stipple counters in the Stipple Unit to be reset to zero, and would typically be used for the first segment in a polyline. This action is also qualified by the LineStippleEnable bit and also the stipple enable bits in the Stipple Unit. When this bit is reset the stipple counters carry on from where they left off (if line stippling is enabled)
3	FastFillEnable	✗	✓	x	This bit, when set, causes the span fill mechanisms to be used for the rasterisation process. The type of span filling is specified in the SpanOperation field. When this bit is reset the normal rasterisation process occurs.

4, 5	Unused	0	0	x	
6, 7	Primitive Type	✗	✓		This two bit field selects the primitive type to rasterise. The primitives are: 0 = Line 1 = Trapezoid 2 = Point
8	Antialiase Enable	✗	✓		This bit, when set, causes the generation of sub scanline data and the coverage value to be calculated for each fragment. The number of sub pixel samples to use is controlled by the AntialiasingQuality bit. When this bit is reset normal rasterisation occurs.
9	Antialiasing Quality	✗	✓		This bit, when set, sets the sub pixel resolution to be 8x8 When this bit is reset the sub pixel resolution is 4x4.
10	UsePoint Table	✗	✓		When this bit and the AntialiasingEnable are set, the dx values used to move from one scanline to the next are derived from the Point Table.
11	SyncOnBit Mask	✗	✓		This bit, when set, causes a number of actions: The least significant bit or most significant bit (depending on the MirrorBitMask bit) in the Bit Mask register is extracted and optionally inverted (controlled by the InvertBitMask bit). If this bit is 0 then any fragments are skipped. After every fragment the BitMask register is rotated by one bit. If all the bits in the BitMask register have been used then rasterisation is suspended until a new BitMaskPattern tag is received. If any other tag is received while the rasterisation is suspended then the rasterisation is aborted. The message which caused the abort is then processed as normal. Note the behaviour is slightly different when the SyncOnHostData bit is set to prevent a deadlock from occurring. In this case the rasterisation doesn't suspend when all the bits have been used and if new BitMaskPattern tags are not received in a timely manner then the subsequent fragments will just reuse the bit mask.
12	SyncOnHost Data	✗	✓		When this bit is set a fragment is produced only when one of the following tags have been received from the host: Depth, Stencil, Color or FBData, FBSourceData. If SyncOnBitMask is reset then any tag other than one of these three is received then the rasterisation is aborted. If SyncOnBitMask is set then any tag other than one of these five or BitMaskPattern is received then the rasterisation is aborted. The tag which caused the abort is then processed as normal for that register type. The <i>BitMaskPattern</i> register doesn't cause any fragments to be generated, but just updates the BitMask register.

13	TextureEnable	✘	✓	x	This bit, when set, enables texturing of the fragments produced during rasterisation. Note that the Texture Units must be suitably enabled as well for any texturing to occur. When this bit is reset no texturing occurs irrespective of the setting of the Texture Unit controls. This bit is useful to temporarily force no texturing for this primitive.
14	FogEnable	✘	✓	x	This bit, when set, enables fogging of the fragments produced during rasterisation. Note that the Fog Unit must be suitably enabled as well for any fogging to occur. When this bit is reset no fogging occurs irrespective of the setting of the Fog Unit controls. This bit is useful to temporarily force no fogging for this primitive.
15	Coverage Enable	✘	✓	x	This bit, when set, enables the coverage value produced as part of the antialiasing to weight the alpha value in the alpha test unit. Note that this unit must be suitably enabled as well. When this bit is reset no coverage application occurs irrespective of the setting of the AntialiasMode.
16	SubPixel Correction Enable	✘	✓	x	This bit, when set enables the sub pixel correction of the color, depth, fog and texture values at the start of a scanline. When this bit is reset no correction is done at the start of a scanline. Sub pixel corrections are only applied to aliased trapezoids.
17	Reserved	0	0	x	
18	SpanOperation	✘	✓	x	This bit, when clear, indicates the writes are to use the constant color found in the previous <i>FBBlockColor</i> register. When this bit is set write data is variable and is either provided by the host (i.e. SyncOnHostData is set) or is read from the framebuffer.
19	Unused	0	0	x	
20...26	Reserved	✘	✓	x	
27	FBSourceRead Enable	✘	✓	x	This bit, when set enables source buffer reads to be done in the Framebuffer Read Unit. Note that the Framebuffer Read Unit must be suitably enabled as well for the source read to occur. When this bit is reset no source reads occur irrespective of the setting of the Framebuffer Read Unit controls.
28...31	Unused	0	0	x	

RasterizerMode

Name	Type	Offset	Format
RaasterizerMode	Rasterizer	0x80A0	Bitfield
RaasterizerModeAnd	Rasterizer	0xABAA0	Bitfield
RaasterizerModeOr	Rasterizer	0xABAA8	Bitfield

Control register

Bits	Name	Read ⁹	Write	Reset	Description
0	MirrorBit Mask	✓	✓	x	<ul style="list-style-type: none"> When set the bit mask bits are consumed from the most significant end towards the least significant end. When reset the bit mask bits are consumed from the least significant end towards the most significant end.
1	InvertBit Mask	✓	✓	x	When this bit is set the bit mask is inverted first before being tested.
2,3	Fraction Adjust	✓	✓	x	These bits control the action of a ContinueNewLine command and specify how the fraction bits in the Y and XDom DDAs are adjusted. <ul style="list-style-type: none"> 0: No adjustment is done, 1: Set the fraction bits to zero, 2: Set the fraction bits to half. 3: Set the fraction to <i>nearly half</i>, i.e. 0x7fff
4,5	Bias Coordinates	✓	✓	x	These bits control how much is added onto the SartXDom, StartXSub and StartY values when they are loaded into the DDA units. The original registers are not affected. <ul style="list-style-type: none"> 0: Zero is added, 1: Half is added, 2: <i>Nearly half</i>, i.e. 0x7fff is added
6	Reserved	✓	✓	x	Reserved
7,8	BitMask ByteSwap Mode	✓	✓	x	These bit controls the byte swapping of the BitMask data before it is used. If the bytes are labelled ABCD on input then they are swapped as follows: <ul style="list-style-type: none"> 0: ABCD (i.e. no swap) 1: BADC 2: CDAB 3: DCBA
9	BitMask Packing	✓	✓	x	This bit controls whether the bitMask data is packed or if a new BitMask data is required on every scanline. <ul style="list-style-type: none"> 0: BitMask data is packed, 1: BitMask data is provided for each scanline.

⁹ Logic Op register readback is via the main register only

10-14	BitMaskOffset	✓	✓	x	These bits hold the bit position in the BitMask data where the first bit is taken from for the bit mask test for the first BitMask data on a new scanline. Subsequent BitMask data starts from bit 0 until the next scanline. Successive bits are taken from increasing bit positions until the bit mask is consumed (i.e. bit 31 is reached). The least significant bit is bit zero.
15,16	HostDataByteSwapMode	✓	✓	x	These bits controls the byte swapping of the BitMask data before it is used. If the bytes are labelled ABCD on input then they are swapped as follows: 0: ABCD (i.e. no swap) 1: BADC 2: CDAB 3: DCBA
17	MultiRasterizer	✓	✓	x	This bit selects whether the rasterizer is to work in single rasterizer mode or in multi-Rasterizer mode. In multi-rasterizer mode it only processes the scanlines allocated to it. 0: Single Rasterizer mode 1: Multi-Rasterizer mode
18	YLimitsEnable	✓	✓	x	This bit, when set, enables the Y limits testing to be done between the minimum and maximum Y values given by the YLimits register.
19	Reserved	✓	✓	x	
20...22	StripeHeight	✓	✓	x	This field specifies the number of scanlines in a stripe. The options are: 0 = 1 3 = 8 1 = 2 4 = 16 2 = 4
23	WordPacking	✓	✓	x	This bit controls how the two host words sent during , a span operation are packed into the 64 bit internal span data. 0 = first word in bits 0...31, second word in 32...63 1 = first word in bits 32...63, second word in 0...31
24	OpaqueSpans	✓	✓	x	This bit, when set allows the color of each pixel in the span to be either foreground or background as set by the supplied bit masks. If this bit is 0 then any supplied bit masks are anded with the pixel mask to delete pixels from the span.
25	Reserved	/	/	x	
26	D3DRules	✓	✓	x	This bit, if set, uses D3D rules for subpixel correction calculations, otherwise OpenGL rules are used.
27...31	Reserved	0	0	x	Reserved for future use, mask to 0

-
- Notes:
- Defines the long term mode of operation of the rasterizer.
 - The `OpaqueSpan` field determines how constant color spans are written (recall the `Render` command selects between constant color or variable color spans). Transparent spans just use one color for the foreground pixels and the background pixels are not written. Opaque spans write to foreground and background pixels using `FBBlockColor` for the foreground pixels and `FBBlockColorBack` for the background pixels. This bit should be set to 0 for performance reasons when foreground/background processing is not required.
 - 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.
-

8.5 2D Setup

This unit performs a number of functions to improve the throughput of 2D rendering. There are two new registers - **Render2D** and **Render2DGlyph** - which allow:

- Rectangle setup using only two messages
- Glyph rendering from texture memory in two messages
- Glyph data can be handled (downloaded, chopped and padded) scanline by scanline compatibly with bitmap textures
- Packed pixel downloads are converted from 4- to 8-bit format
- Run Length Encoded (RLE) data downloads are automatically expanded

The `Render2D` command incidentally flushes the write combine buffers to ensure memory is updated (and therefore visible to bypass or video reads) after the rectangle is rendered.

8.5.1 Glyph rendering

Once the position is established (**GlyphPosition**) subsequent glyphs can be rendered by writing the address of the texture bitmap containing the glyph to the **TextureBaseAddr(0)** register followed by the **Render2DGlyph** Command. The glyph position is updated automatically from the *Width* bitfield. Because glyphs are rendered as a span, the direction is always increasing X and Y.

9

Scissor, Stipple and Color DDA Units

9.1 Scissor Unit

Two scissor tests are provided in GLINT R4, the User Scissor test and the Screen Scissor test.

The user scissor checks each fragment or span against a user supplied scissor region; the screen scissor converts the fragment to screen-relative coordinates and checks that the fragment or span lies within the screen.

The scissor unit operates both on active fragments and spans. In span processing the pixel mask bits corresponding to a failed fragment are reset.

9.1.1 User Scissor Test

The user scissor test checks each fragment as follows:

$$X_{Min} \leq X < X_{Max}$$

$$Y_{Min} \leq Y < Y_{Max}$$

Where X and Y are the coordinates for the fragments, and XMin, XMax, YMin and YMax define the user supplied scissor region. If a fragment fails the test it is discarded. The test may be screen- or window- relative.

9.1.2 Screen Scissor Tests

This test ensures that a fragment lies within the screen boundaries. For each fragment the XY origin stored in the **WindowOrigin** register is added to the fragment coordinates and this is tested against the screen boundaries stored in the **ScreenSize** register. Since the X and Y coordinates are held as 2's complement numbers, the window origin can be moved off the edges of the screen.

Note that the **WindowOrigin** register only affects the origin for clipping, it does not affect the base address for rendering. The *Windows Initialization* chapter gives further details on how to set the base address of a window for rendering.

The Screen Scissor test is:

$$0 \leq (X + WX) < SW$$

$$0 \leq (Y + WY) < SH$$

Where:

X = Fragment X coordinate

WX = Window origin X coordinate

Y = Fragment Y coordinate

WY = Window origin Y coordinate

SW = Screen Width

SH = Screen Height

The diagram below shows a simple case of a screen with a single window which has a user defined scissor region. The shaded area shows the region where fragments pass the

user and screen scissor tests and so can progress in the pipeline. Fragments outside this region are culled from the pipeline.

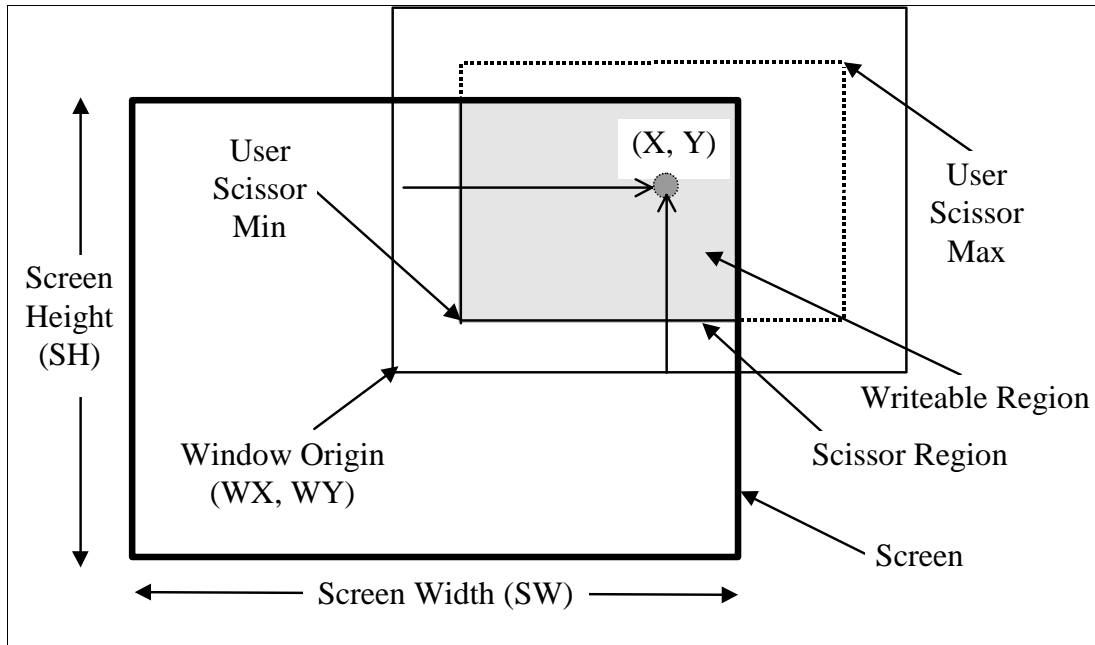


Figure 9-1 Screen Scissor and User Scissor Tests

This test may reject fragments if some part of a window has been moved off the screen. It does not reject fragments if part of a window is simply overlapped by another window (GID testing can be used to detect this).

9.1.3 Scissor Registers

The unit is controlled by the **ScissorMode** register:

Name	Type	Offset	Format		
ScissorMode	Scissor	0x8180	Bitfield		
ScissorModeAnd	Scissor	0xABB0	Bitfield Logic Mask		
ScissorModeOr	Scissor	0xABB8	Bitfield Logic Mask		
<i>Control registers</i>					
Bits	Name	Read 10	Write	Reset	Description
0	UserScissor Enable	✓	✓	x	enables the user scissor clipping
1	ScreenScissor Enable	✓	✓	x	enables the screen scissor clipping
2...31	Unused	0	0	x	

¹⁰ Logic Op register readback is via the main register only

Figure 9-2 ScissorMode Register

The screen scissor test would normally be left enabled by default. The most common exception is during image upload.

The user scissor region is specified by two registers **ScissorMinXY** and **ScissorMaxXY** the X values are stored in the least significant 16 bits of the register, the Y values in the most significant 16 bits of the register.

The **WindowOrigin** register has the X coordinate of the origin stored in the least significant 16 bits of the register, and the Y coordinate in the most significant 16 bits of the register. As each fragment is generated by the rasterization unit this origin is added to the coordinates of the fragment to generate its screen coordinates.

The **ScreenSize** register specifies the screen width and height, with the width in the least significant 16 bits and the height in the most significant 16 bits.

9.1.4 Span Operations and the Scissor Unit

If a span mask is presented to the scissor unit, the pixel mask (and potentially the color mask) is modified to zero out bits corresponding to pixels which lie outside the scissor region. This is true for both the user scissor and the screen scissor. The screen scissor first converts the span's coordinates to screen-relative.

9.1.5 Scissor Example

To enable screen scissor for a region: $10 \leq X < 500$, $100 \leq Y < 200$ with a screen size of 1280x1024 and the window origin at (100,100).

```
// Set the screen size
    screenSize.Width = 1280
    screenSize.Height = 1024
    ScreenSize(screenSize)

// Set the window origin
    ScissorMode(scissorMode)
    WindowOrigin(windowOrigin)

// Render primitives
```

9.2 Stipple Unit

Stippling is a process which checks each fragment against a bit in a defined pattern. The fragment can either be rejected or accepted depending on the result of the stipple test. If it is rejected, then it undergoes no further processing, otherwise it proceeds down the pipeline. GLINT R4 supports line and area stippling.

9.2.1 Area Stippling

Both the *AreaStippleEnable* in the **PrepareToRender** message and *enable* in the **AreaStippleMode** message must be set to enable the area stipple test. If the stipple test is disabled then the area span mask is set to 0xffffffffffff, otherwise it is calculated as follows.

The address of the stipple pattern row to use in the test is calculated as follows:

- Add the Y offset to the bottom five bits of Y coordinates of the span coordinate. If the corresponding mirror bits are set then invert the Y address.
- Extract the bottom *m* bits of the resulting Y value where *m* is determined by the Y Sel fields. The extracted Y address is zero extended to 5 bits where necessary and is now called Y'.
- Add the *YTableOffset* to Y' to move the test to the required sub stipple pattern row.

The Y' value selects the row in the stipple RAM (row zero is at **AreaStipplePattern[0]**) and this is the first value of the area stipple mask which is processed by each of the following stages and passed on to the next:

- The mask is rotated right by the *XTableOffset* amount to select the sub stipple pattern to replicate, mirror, etc.
- The least significant 2, 4, 8, 16 or 32 bits are extracted from the area stipple mask and replicated to fill all 32 bits of the mask. The Xsel field determines the number of bits to replicate (0 = 2 bit to replicate, etc.).
- Next the area stipple mask is mirrored if the *MirrorX* bit is set. The mirroring is done by swapping bits (0, 63), (1, 62), (2, 61), etc..
- The area span mask is inverted under control of the *InvertStipplePattern* bit.
- The area span mask is rotated right by (*Xoffset* + *X*) bits.

The area stipple pattern is always 32x32 and is window relative. However the *XtableOffset* and *YtableOffset* fields in **AreaStippleMode** allow the 32x32 bit table to hold several smaller area stipple patterns. The least significant 5 bits of the fragment's (X,Y) coordinates, index into the controlling bit of the 2D stipple pattern. If the selected bit in the pattern is set, then the fragment passes the test, otherwise it is rejected as described above.

The mask is defined in the **AreaStipplePattern** registers. Area stippling is enabled and controlled using the **AreaStippleMode** register and must be qualified by the *AreaStippleEnable* bit in the **Render** command register. This allows temporary disable stippling when Bitmaps or OGL pixel rectangles are being rendered.

The address selection can be controlled independently in the X and Y directions. In addition the bitpattern can be inverted or mirrored using *InvertStipplePattern* or *MirrorX*. Inverting the bit pattern has the effect of changing the sense of the accept/reject test. If the mirror bit is set the most significant bit of the pattern is towards the left of the window, the default is the converse.

In some situations window relative stippling is required but coordinates are only available screen relative. To allow windows relative stippling, an offset can be added to the coordinates before indexing the stipple table. X and Y offsets can be controlled independently.

9.2.2 Line Stippling

Line stippling applies normally to aliased lines. Antialiased lines can be stippled by applying the stipple pattern to the rectangles which constitute the antialiased line.

In this test, fragments are conditionally rejected on the outcome of testing a linear stipple mask. If the bit is zero then the test fails, otherwise it passes. The line stipple pattern is 16 bits in length and is scaled by a repeat factor, r, (in the range 1 to 512). The stipple mask bit, b, which controls the acceptance or rejection of a fragment is determined using:

$$b = (\text{floor}(s / r)) \bmod 16$$

where s is the stipple counter which is incremented for every fragment (normally along the line). This counter may be reset at the start of a polyline, but between segments it continues as if there were no break.

The stipple pattern can be optionally mirrored, that is the bit pattern is traversed from most significant to least significant bits, rather than the default, from least significant to most significant.

The **UpdateLineStippleCounters** register controls initialization of the line stipple counters, which can be reset or loaded from a previously saved value. The **UpdateLineStippleCounters** register can be reset by writing 0 to bit 0 (earlier chips required resetting all 32 bits in the register).

The **SaveLineStippleCounters** register is used to save the current line stipple counters. The combination of **UpdateLineStippleCounters** and **SaveLineStippleCounters** is useful to implement stippling of wide polylines.

Line stippling is enabled using the **LineStippleMode** register and must be qualified by the *LineStippleEnable* bit in the **Render** command register.

9.2.3 Span Operations and Stippling

If the Area Stipple unit is enabled it modifies span masks generated by the rasterizer. (Line stipple has no effect on the span mask.) The mask can be rotated or inverted before being ANDed with the pixel mask for transparent spans, or the color mask for spans using the *OpaqueSpan* bit in the **AreaStippleMode** register.

9.2.4 Registers

The **LineStippleMode** register controls line stipple:

Name	Type	Offset	Format
LineStippleMode	Stipple	0x81A8	Bitfield
LineStippleModeAnd	Stipple	0xABC0	Bitfield Logic Mask
LineStippleModeOr	Stipple	0xABC8	Bitfield Logic Mask

Control register

Bits	Name	Read	Write	Reset	Description
0	StippleEnable	✓	✓	x	This field, when set, enables the stippling of lines. The <i>LineStippleEnable</i> bit in the <i>Render</i> command must also be set.
1...9	RepeatFactor	✓	✓	x	This field holds the positive repeat factor for stippled lines. The repeat factor stored here is one less than the desired repeat factor.

10...25	StippleMask	✓	✓	x	This field holds the stipple pattern.
26	Mirror	✓	✓	x	This field, when set, will mirror the StippleMask before it is used.
27...31	Unused	0	0	x	

Figure 9-3 LineStippleMode Register

The least significant bit of the **UpdateLineStippleCounters** register controls loading the line stipple counters. If set the line stipple counters are loaded with the previously saved values. If reset, the counters are cleared to zero. The counters can also be reset by means of the *ResetLineStipple* bit in the **Render** command.

The **AreaStippleMode** register controls area stipple operation:

Name	Type	Offset	Format
AreaStippleMode	Stipple	0x81A0	Bitfield
AreaStippleModeAnd	Stipple	0xABD0	Bitfield Logic Mask
AreaStippleModeOr	Stipple	0xABD8	Bitfield Logic Mask

Control registers

Bits	Name	Read ¹¹	Write	Reset	Description
0	Enable	✓	✓	x	This field, when set, enables area stippling. The <i>AreaStippleEnable</i> bit in <i>Render</i> must also be set for this to have an effect.
1..3	X address select:	✓	✓	x	0 = 1 bit 2 = 3 bit 4 = 5 bit
4..6	Y address select:	✓	✓	x	0 = 1 bit 2 = 3 bit 3 = 4 bit 4 = 5 bit
7..11	X Offset	✓	✓	x	This field holds the offset to add to the X value before it is used to index into the stipple bit. This allows a window relative stipple pattern to be selected when the coordinates are given in screen relative format.
12..16	Y Offset	✓	✓	x	This field holds the offset to add to the Y value before it is used to index into the area stipple pattern table. This allows a window relative stipple pattern to be selected when the coordinates are given in screen relative format.
17	Invert Stipple Pattern	✓	✓	x	0 = No Invert 1 = Invert
18	Mirror X	✓	✓	x	0 = No Mirror 1 = Mirror
19	Mirror Y	✓	✓	x	0 = No Mirror 1 = Mirror
20	OpaqueSpan	✓	✓	x	This bit, when set, allows the area stipple pattern to modify the color mask, otherwise the pixel mask is modified.

¹¹ Logic Op register readback is via the main register only

21...25	XTableOffset	✓	✓	x	This field allows a sub area stipple pattern to be extracted from the area stipple table, i.e. the area stipple table is treated as a cache of smaller stipple patterns.
26...30	YTableOffset	✓	✓	x	This field allows a sub area stipple pattern to be extracted from the area stipple table, i.e. the area stipple table is treated as a cache of smaller stipple patterns.
31	Unused	0	0	x	

Figure 9-4 AreaStippleMode Register

The *EnableUnit* bit in the **LineStippleMode** and **AreaStippleMode** registers are qualified by the *LineStippleEnable* and *AreaStippleEnable* bits in the **Render** command register.

The **SaveLineStippleCounters** register (which has no data field) saves the line stipple counters internally.

The area stipple is set up in the **AreaStipplePattern** register, where n represents an integer between 0 and 31.

The **LoadLineStippleCounters** register is shown in the *GLINT R4 Reference Guide*

Name	Type	Offset	Format
LoadLineStippleCounters	Global	0x81B0	Bitfield

Command

Bits	Name	Read	Write	Reset	Description
0...3	LiveBit Counter	✗	✓	x	
4...12	LiveRepeat Counter	✗	✓	x	
13...16	SegmentBit Counter	✗	✓	x	
17...25	SegmentRepeat Counter	✗	✓	x	
26...31	Unused	0	0	x	

Figure 9-5 LoadLineStippleCounters register

9.2.5 Examples

A repeating area stipple pattern of 2x2 pixels producing a 50% grey area:

```
// Use only the first two table entries
AreaStipplePattern0(0x1)
AreaStipplePattern1(0x2)

// Set-up mode register
areaStippleMode.UnitEnable = GLINT R4_ENABLE
areaStippleMode.XSel = 0 // Address index based on
areaStippleMode.YSel = 0 // LSB of address, repeats
```

```

// every 2nd pixel in X & Y
areaStippleMode.XOffset = 0
areaStippleMode.YOffset = 0
areaStippleMode.Invert = 0
areaStippleMode.MirrorY = 0
areaStippleMode.MirrorX = 0
// Load mode register
    AreaStippleMode(areaStippleMode)
// When the Render command is sent the
// AreaStippleEnable
// bit should be set in addition to the area stipple
// test being enabled:
// render.AreaStippleEnable = GLINT R4_TRUE

```

9.2.6 Line Stipple Example

A line stipple which rejects alternate fragments:

```

// Set counters to zero
UpdateLineStippleCounters(0x0)
// Set the stipple mode
lineStippleMode.UnitEnable = GLINT R4_ENABLE
lineStippleMode.RepeatFactor = 0 // Repeat factor 1
lineStippleMode.StippleMask = 0xAAAA
LineStippleMode(lineStippleMode)
// When issuing a Render command the
// LineStippleEnable bit should be set in addition
// to the line stipple test being enabled:
// render.LineStippleEnable = GLINT R4_TRUE

```

9.2.7 Area Stipple Pattern Example

Another repeating area stipple pattern of 2x2 pixels producing a 50% grey area:

AreaStiPPlEPattern0	(0xAAAAAAAA)
AreaStipplePattern1	(0x55555555)
AreaStipplePattern2	(0xAAAAAAAA)
AreaStipplePattern3	(0x55555555)
AreaStipplePattern4	(0xAAAAAAAA)
AreaStipplePattern5	(0x55555555)
AreaStipplePattern6	(0xAAAAAAAA)


```

AreaStipplePattern7      (0x55555555)
AreaStipplePattern31    (0x55555555)
// Set-up mode register
areaStippleMode.UnitEnable = GLINT R4_ENABLE
areaStippleMode.Xselect = 0
areaStippleMode.Yselect = 0
areaStippleMode.Xoffset = 0
areaStippleMode.Yoffset = 0
areaStippleMode.Invert = 0
areaStippleMode.MirrorY = 0
areaStippleMode.MirrorX = 0
// Load mode register
AreaStippleModeareaStippleMode)

// When issuing a Render command, the
// AreaStippleEnable bit should be set to enabled:
// Arender.AreaStippleEnable = GLINT R4_TRUE

```

9.3 Color DDA Unit

The color DDA unit is used to associate a color with a fragment produced by the rasterizer. This unit should be enabled for rendering operations and disabled for pixel rectangle operations (i.e. copies, uploads and downloads). Color DDA functionality is controlled by the ColorDDA register:

ColorDDAMode ColorDDAModeAnd ColorDDAModeOr

Name	Type	Offset	Format
ColorDDAMode	Color	0x87E0	Bitfield
ColorDDAModeAnd	Color	0xABE0	Bitfield Logic Mask
ColorDDAModeOr	Color	0xABE8	Bitfield Logic Mask

Control registers

Bits	Name	Read ¹²	Write	Reset	Description
1	Enable	✓	✓	x	This bit, when set, causes the current color to be generated.

¹² Logic Op register readback is via the main register only

2	Shading	✓	✓	x	Selects the shading mode. The two options are: 0 = Flat – the color is taken from the Constant Color register. 1 = Gouraud – the color is taken from the DDAs.
3...31	Unused	0	0	x	

Notes: The ColorDDAMode register controls the operation of the Color DDA unit using the Enable and Shading bits. 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.

9.3.1 RGBA and Color-Index(CI) Modes

Two color modes are supported by GLINT R4, RGBA and color index (CI). GLINT R4's internal color representation is RGBA with 8 bits per component: A typical register layout is **ConstantColor**:

Constant Color

Name	Type	Offset	Format
ConstantColor	Delta	0x87E8	Bitfield

Control register

Bits	Name	Read	Write	Reset	Description
0...7	Red	✓	✓	x	
8...15	Green	✓	✓	x	
16...23	Blue	✓	✓	x	
24...31	Alpha	✓	✓	x	

Notes: This register holds the constant color in packed format. This is a legacy register maintained for backwards compatibility which has been superseded by the *ConstantColorDDA* register.

The *ConstantColorDDA* register, as well as loading up the constant color register, also loads the DDA start register from the corresponding color byte and sets the dx and dyDom gradients to zero. This allows a constant color to be set up irrespective of the shading mode.

This format is the same for all the different framebuffer configurations supported. If the number of bits in the framebuffer for a color component is less than 8 then the color value is left shifted into the most significant bits of that component's field. The unused least significant bits should be set to zero.

In CI mode the color index is placed in the lower byte of the 32 bit register (i.e., the red component). If less than 8 bits are used the index is left justified to be in the most significant end of the red component. The unused least significant bits should be set to zero.

For further information on Color modes see chapter 9 - Color Format and Logical Ops.

9.3.2 Gouraud Shading

Shading may be flat or Gouraud. For flat shading, the color value is taken from the **ConstantColor** register, not from the DDA. When in Gouraud shading mode, the color DDA unit performs linear interpolation given a set of start and increment values. Interpolated values are clamped to avoid overflow or underflow. For details of color interpolation calculation see Appendix 13-2 - Calculating Depth Gradient Values.

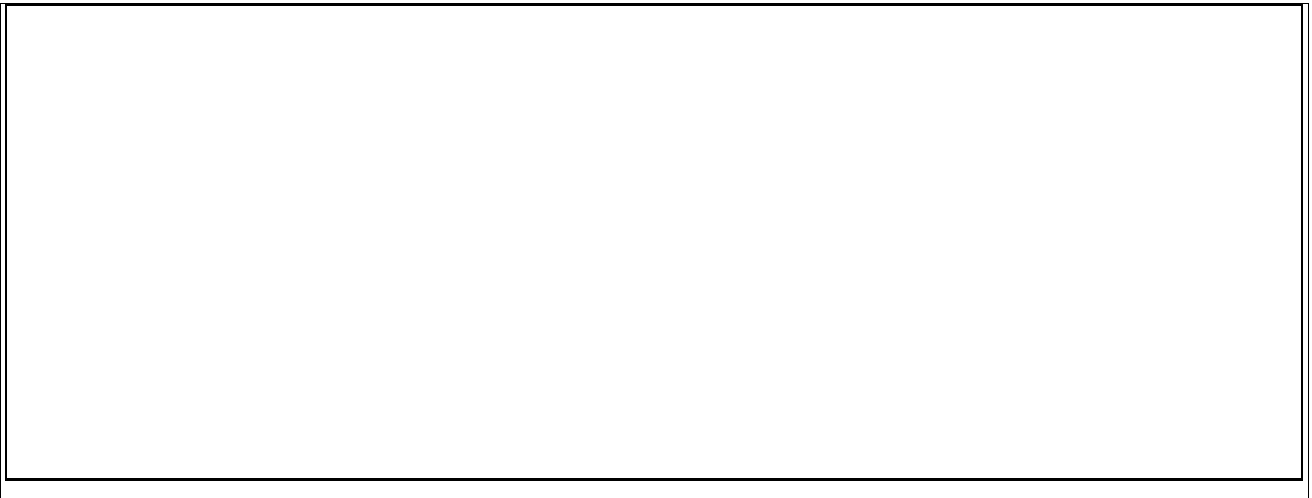


Figure 9-6 Color Interpolation

Color interpolates from the dominant edge of the trapezoid to the subordinate edges. This means that two increment values are required per color component, one to move along the dominant edge and one to move across the span to the subordinate edge. This is illustrated in Figure 9-6, where C represents a color component (red, green, blue, alpha or color index). The control registers are shown in table 3.3, below.

For Gouraud shaded lines, each line is treated as the dominant edge of a trapezoid so no **dCdx** increment is required.

To allow accurate interpolation, the increment values are specified in a 24bit fixed point format. The format is 2's complement with 9 bits of integer and 15 bits of fraction. A typical register layout is shown below:

Name	Type	Offset	Format
dAdyDom	Color DDA	0x87D8	Fixed point

Control register

Bits	Name	Read	Write	Reset	Description
0...14	Fraction	✓	✓	x	2's complement 9.15 fixed point fraction
15...23	Integer	✓	✓	x	2's complement 9.15 fixed point integer
24...31	Unused	0	0	x	

Figure 9-7 Fixed Point Color Format

Note that if you are rendering to multiple buffers and have initialized the start and increment values in the color DDA unit, then any subsequent **Render** command will reload the start values.

If subpixel correction has been enabled for a primitive, then any correction required will be applied to the color components.

The registers to set up Gouraud shading in the color DDA unit are:

Register	Data Field	Description
RStart	Fixed point 9.15 format	Red start value
dRdx	Fixed point 9.15 format	Red derivative per unit X
dRdyDom	Fixed point 9.15 format	Red derivative per unit Y, dominant edge
GStart	Fixed point 9.15 format	Green start value
dGdx	Fixed point 9.15 format	Green derivative per unit X
dGdyDom	Fixed point 9.15 format	Green derivative per unit Y, dominant edge
BStart	Fixed point 9.15 format	Blue start value
dBdx	Fixed point 9.15 format	Blue derivative per unit X
dBdyDom	Fixed point 9.15 format	Blue derivative per unit Y, dominant edge
AStart	Fixed point 9.15 format	Alpha start value
dAdx	Fixed point 9.15 format	Alpha derivative per unit X
dAdyDom	Fixed point 9.15 format	Alpha derivative per unit Y, dominant edge

Table 9.3 Color Interpolation Registers

9.3.3 Flat Shading Example

A flat shaded primitive:

```
// Set DDA to flat shade mode
    colorDDAMode.UnitEnable = GLINT R4_ENABLE
    colorDDAMode.Shade = GLINT R4_FLAT_SHADE_MODE
    ColorDDAMode(colorDDAMode)
ConstantColor(0xFFFFFFFF) // Load the flat color
```

9.3.4 Gouraud Shaded Trapezoid Example

```
// Enable unit in Gouraud shading mode
    colorDDAMode.UnitEnable = GLINT R4_ENABLE
    colorDDAMode.Shade = GLINT R4_GOURAUD_SHADE_MODE
    ColorDDAMode(colorDDAMode)

// 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
dBdyDom ()

```

9.3.5 Gouraud Shaded Line Example

```

// Set DDA for Gouraud shaded mode
    colorDDAMode.UnitEnable = GLINT R4_ENABLE
    colorDDAMode.Shade = GLINT R4_GOURAUD_SHADE_MODE
    ColorDDAMode(colorDDAMode)

// For lines we need only start values and
// dominant edge deltas
RStart() // Set-up the red component start value
dRdyDom() // Set-up the red component increment
GStart() // Set-up the green component start value
dGdyDom() // Set-up the green component increment
BStart() // Set-up the blue component start value
dBdyDom() // Set-up the blue component increment

```


10

Localbuffer Read/Write

The localbuffer holds the Graphic ID, Stencil and Depth data associated with a fragment. The localbuffer address calculation uses the LocalBuffer mode, address and offset registers registers to set base addresses and screen-relative offsets, as well as positioning the Depth, Stencil and GID planes. For details see "Localbuffer and Framebufferonfiguration" in *Initialization* section 12.2.7 below.

The origin can be set in the relevant BufferMode register(s) to top left or bottom right using the *Origin* field.

Note: Enabling Patch addressing in the Layout field of the buffer mode registers introduces additional complexity into the address calculation which is beyond the scope of this manual. Localbuffer bypass accesses are not recommended when Patch mode addressing is enabled.

The localbuffer read format is controlled by the **LBDestReadFormat** register's definition of the positions of the Depth, Stencil and GID planes.

Selecting a depth width of 15 bits forces the stencil and GID fields to be set from bit 15 of the pixel and ignores the normal stencil and GID settings.

The natural internal width of the fields are depth (31), stencil (8), GID (4). If the specified width of a field is less than its internal width then the field is zero extended to its internal width.

Field	Width	Position
Depth	16, 24, 31, 15	Bit 0 to bit 3
Stencil	0 - 8	Starts at 16 to 39 (entered as 0 – 23)
GID	0 - 4	Starts at 16 to 39 (entered as 0 – 23) following Stencil

Table 10.4 Localbuffer Configurations

The enables for these are in the **GIDMode**, **StencilMode** and **DepthMode** registers. These tell Permedia4 which areas of the localbuffer are required for various operations. The operations are specified by the **LBWriteMode** Operation field in bits 29-31:

29...31	Operation	✓	✓	x	<p>This field defines where the data is to be taken from to do the write and what is to happen to it afterwards. This is only of interest during an upload or download operation. The options are:</p> <p>0 = No operation 1 = Download depth 2 = Download stencil 3 = Upload depth 4 = Upload stencil</p>
---------	-----------	---	---	---	--

Table 10.5 Localbuffer Read/Write Modes.

Note that the **LBReadFormat** and **LBWriteFormat** registers should not be written to while there are pending reads to the localbuffer. To avoid this a write to these registers should normally be preceded by a **WaitForCompletion** command.

10.1.1 Mode Registers

The **LBDestReadMode** register is as shown below:

LBDestReadMode LBDestReadModeAnd LBDestReadModeOr

Name	Type	Offset	Format
LBDestReadMode	Localbuffer	0xB500	Bitfield
LBDestReadModeAnd	Localbuffer	0xB580	Bitfield Logic Mask
LBDestReadModeOr	Localbuffer	0xB588	Bitfield Logic Mask

Control registers

Bits	Name	Read 13	Write	Reset	Description
0	Enable	✓	✓	x	This bit, when set, causes fragments or spans to read from the destination buffer
1	Reserved	✗	✗	x	
2...4	StripePitch	✓	✓	x	This field specifies the number of scanlines between the first scanline in a stripe and the first scanline in the next stripe. (It would normally be set to a number of RXs * StripeHeight). The options are: 0 = 1 1 = 2 2 = 4 3 = 8 4 = 16 5 = 32 6 = 64 7 = 128 This field will normally be set to zero for GLINT R4.
5...7	StripeHeight	✓	✓	x	This field specifies the number of scanlines in a stripe. The options are: 0 = 1 1 = 2 2 = 4 3 = 8 4 = 16 This field will normally be set to zero for GLINT R4.
8	Layout	✓	✓	x	This field selects the layout of the pixel data in memory for the destination buffer. The options are: 0 = Linear 1 = Patch64
9	Origin	✓	✓	x	This field selects where the window origin is for the destination buffer. The options are: 0 = Top Left. 1 = Bottom Left
10	UseRead Enables	✓	✓	x	When this bits is set the enables in the LBDestReadEnables register are used to determine if a destination read is required. The Enable bit must also be set as well for a read to occur.

¹³ Logic Op register readback is via the main register only

11	Packed16	✓	✓	x	When this bit is set the pixel size is 16 bits so a single memory word can hold 8 depth values.
12...23	Width	✓	✓	x	This field holds the width of the destination buffer. Its range is 0...4095.

Notes: Defines the localbuffer destination read operation. The destination address calculations are controlled by the *LBDestReadMode* register and the address is a function of X, Y, *LBDestReadBufferAddr*, *LBDestReadBufferOffset*, width and Packed16 parameters.

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.

Figure 10-1 LBDestReadMode Register

LBWriteFormat

Name	Type	Offset	Format
LBWriteFormat	Localbuffer <i>Control register</i>	0x88C8	Bitfield

Bits	Name	Read	Write	Reset	Description
0...1	DepthWidth	✓	✓	x	This field specifies the width of the depth field. The depth field always starts at bit position 0. The width options are: 0 = 16 bits 1 = 24 bits 2 = 31 bits 3 = 15 bits When the depth width is 15 the GID and Stencil fields are ignored and a one bit GID and Stencil are taken from bit 15. Only one of the GID or Stencil operation are enabled to select the desired field type.
2...5	StencilWidth	✓	✓	x	This field specifies the width of the stencil field. The legal range of values are 0...8. The stencil field always starts at bit position given in the next field.
6...10	StencilPosition	✓	✓	x	This field holds position of the least significant bit of the stencil field. The legal range of values are 0...23, representing bit positions 16...39 respectively.
11...19	Reserved	0	0	x	
20...22	GIDWidth	✓	✓	x	This field specifies the width of the Graphics ID field. The legal range of values are 0...4. The GID field always starts at the bit position given in the <i>GIDPosition</i> field.
23...27	GIDPosition	✓	✓	x	This field holds position of the least significant bit of the Graphics ID field. The legal range of values are 0...23, representing bit positions 16...39 respectively.
28...31	Reserved	0	0	x	

Notes: This register defines the position and width of the depth, stencil, GID (Graphics ID) in the data read back from the local buffer.

Figure 10-2 LBWriteFormat Register Layout

10.2 Window register

A number of Localbuffer operations, particularly Stencil, are conditioned by the **Window** register.

- The *ForceLBUpdate* bit is used to allow all the fields in the localbuffer to be updated simultaneously. *ForceLBUpdate* overrides all stencil and Depth testing. This is useful during initialization and copy operations.
- When the *LBUpdateSource* bit is set the source of the stencil and depth data is determined by the **StencilMode** and **DepthMode** registers respectively.
- The *OverrideWriteFiltering* control bit, when set causes the testing of $LBData = LBWriteData$ to always fail. This is mainly used when the GID field needs to be changed. It also allows the *LBReadFormat* to be different to the *LBWriteFormat* so the write data as seen by the memory is really different to the data that was read.
- *LBUpdateSource* is used in conjunction with the *ForceLBUpdate* bit to select whether the source data comes from: the localbuffer, or values held in local registers (**Depth**, **Window**, **Stencil**).
- The combination of *LBUpdateSource* being set to *LBSourceData*, and the *ForceLBUpdate* bit being enabled is particularly useful when copying a window from one location on the screen to another.
- The combination of *LBUpdateSource* being set to *Registers* and the force *LBUpdate* bit being enabled is particularly useful for initializing the contents of the various localbuffer fields in a window.
- Normally Permedia4 detects the case where the data to be written to the localbuffer is the same as the data read from the localbuffer, and avoids performing the write. Setting the *OverrideWriteFiltering* bit prevents these writes from being filtered out. This is of value when the localbuffer read format is different from the localbuffer write format since the comparison is done on the internal data format.

10.3 Pixel Ownership (GID) Test Unit

Any fragment generated by the rasterizer may undergo a pixel ownership test. This test establishes the current fragment's write permission to the localbuffer and framebuffer.

10.3.1 Pixel Ownership Test

The ownership of a pixel is established by testing the GID of the current window against the GID of a fragment's destination in the GID buffer. If the test passes, then a write can take place, otherwise the write is discarded.

The sense of the test can be set to one of: always pass, always fail, pass if equal, or pass if not equal. Pass if equal is the normal mode. In Permedia4 the GID planes, if present, are

4 bits deep allowing 16 possible Graphic ID's. If GIDMode is disabled fragments pass through undisturbed.

Pixel ownership is controlled by the relevant LB Format and **GIDMode** registers:

GIDMode

GIDModeAnd

GIDModeOr

Name	Type	Offset	Format
GIDMode	Localbuffer	0xB538	Bitfield
GIDMode And	Localbuffer	0x B5B0	Bitfield Logic Mask
GIDMode Or	Localbuffer	0x B5B8	Bitfield Logic Mask

Control registers

Bits	Name	Read 14	Write	Reset	Description
0	Fragment Enable	✓	✓	x	This bit, when set, causes GID testing to occur on fragments. If the test fails then the fragment is discarded
1	Span Enable	✓	✓	x	This bit, when set, allows the span pixel mask to be modified by GID testing each pixel. The mask is modified to disable those pixels which fail the test.
2...5	Compare Value	✓	✓	x	This field holds the 4 bit GID value to compare against. Unused bits (where the GID width in the local buffer format is less than 4 bits) should be set to zero.
6...7	Compare Mode	✓	✓	x	This field holds the comparison modes available for use during GID testing. The options are: 0 = Always pass 1 = Never pass (i.e. always fail) 2 = Pass when local buffer gid == CompareValue 3 = Pass when local buffer gid != CompareValue
8...9	Replace Mode	✓	✓	x	This field specifies the replacement mode. This is independent of the FragmentEnable bit (except when the replacement depends on the outcome of the GID test). The options are: 0 = Always replace 1 = Never replace 2 = Replace on GID test pass. 3 = Replace on GID test fails
10...13	Replace Value	✓	✓	x	This field holds the 4 bit GID value to replace the value read from the local buffer, if the replace mode is satisfied.
13...31	Reserved	0	0	x	Reserved

Figure 10-3 GIDMode Register

The *CompareMode* field will generally be set to 'Pass if Equal' for GID testing, with the current GID in the appropriate field.

¹⁴ Logic Op register readback is via the main register only

10.4 Stencil Test

The stencil test conditionally rejects fragments based on the outcome of a comparison between the value in the stencil buffer and a reference value. The stencil buffer is updated according to the current stencil update mode which depends on the result of the stencil test and the depth test.

This test only occurs if all the preceding tests (bitmask, scissor, stipple, alpha, pixel ownership) have passed. The stencil test is controlled by the stencil function and the stencil operation. The stencil function controls the test between the reference stencil value and the value held in the stencil buffer. If the test is LESS and the result is true then the fragment value is less than the source value. The stencil operation controls the updating of the stencil buffer, and is dependent on the result of the stencil and depth tests.

The table below shows the stencil functions available:

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

Table 10.6 Stencil Functions

If the stencil test is enabled then the stencil buffer will be updated depending on the outcome of both the stencil and the depth tests (if the depth test is disabled the depth result is set to pass). Refer to the tables below and the definition of the **StencilMode** register in section §10.4.1 to fully understand their relationship.

		Stencil Test	
		Pass	Fail
Depth Test	Pass	<i>dppass</i>	<i>sfail</i>
	Fail	<i>dpfail</i>	<i>sfail</i>

Table 10.7 Possible Update Operations for Stencil Planes

The entries dppass, dpfail and sfail are set to one of the update operations below. Source stencil is the value in the stencil buffer:

Update Method	Mode	Stencil Value
Keep	0	Source stencil
Zero	1	0
Replace	2	Reference stencil
Increment	3	Clamp (Source stencil + 1) to $2^{\text{stencil width}} - 1$
Decrement	4	Clamp (Source stencil -1) to 0
Invert	5	\sim Source stencil

Table 10.8 Stencil Operations

In addition a comparison bit mask is supplied in the **StencilData** register. This is used to establish which bits of the source and reference value are used in the stencil function test. It should normally be set to exclude the top four bits when the stencil width has been set to 4 bits in the **StencilMode** register.

The source stencil value can be from a number of places as controlled by bits 13-14 (*StencilSource*) in the **StencilMode** register:

Stencil Source	Mode	Use
Test logic	0	This is the normal mode.
Stencil register	1	This is used, for instance, in the OpenGL draw pixels function where the host supplies the stencil values in the Stencil register. This is used when a constant stencil value is needed, for example, when clearing the stencil buffer when fast clear planes are not available.
Source stencil value read from the localbuffer	2	This is used, for instance, in the OpenGL copy pixels function when the stencil planes in the destination are not to be updated. The stencil data comes from the localbuffer.
LBSourceData: (stencil value read from the localbuffer)	3	This is used, for instance, in the OpenGL copy pixels function when the stencil planes are to be copied to the destination. .

Table 10.9 Stencil Sources

See *The OpenGL Reference Manual* and *The OpenGL Programming Guide* from Addison-Wesley for more details of stencil operations and examples of its use.

10.4.1 Registers

Stencil test is controlled by the **StencilMode** register:

StencilMode StencilModeAnd StencilModeOr

Name	Type	Offset	Format
StencilMode	Stencil	0x8988	Bitfield
StencilModeAnd	Stencil	0xAC60	Bitfield Logic Mask
StencilModeOr	Stencil	0xAC68	Bitfield Logic Mask

Control registers

Bits	Name	Read	Write	Reset	Description
0	Unit enable	✓	✓	x	0 = Disable 1 = Enable
1...3	Update method	✓	✓	x	if Depth test passes and Stencil test passes (see table 1)
4...6	Update method	✓	✓	x	if Depth test fails and Stencil test passes (see table 1)
7...9	Update method	✓	✓	x	if Stencil test fails (see table 1)
10...12	Mode 0-7	✓	✓	x	Unsigned comparison function (see table 2)
13...14	Stencil source	✓	✓	x	0 = Test Logic 1 = Stencil Register 2 = LBData 3 = LBSourceData
15...16	Stencil widths	✓	✓	x	0 = 4 bits 1 = 8 bits 2 = 1 bit
17...31	Unused	0	0	x	

Figure 10-4 StencilMode Register

The **StencilData** register holds the other data associated with the test.

StencilData StencilDataAnd StencilDataOr

Name	Type	Offset	Format
StencilData	Stencil	0x8990	Bitfield
StencilDataAnd	Stencil	0xB3E0	Bitfield Logic Mask
StencilDataOr	Stencil	0xB3E8	Bitfield Logic Mask

Control registers

Bits	Name	Read	Write	Reset	Description
0...7	Stencil value	✓	✓	x	8 bit stencil test value
8...15	Compare mask	✓	✓	x	Determines which bits are significant in the test
16...23	Writemask	✓	✓	x	Determines which bits in localbuffer are updated
24...31	Reserved	0	0	x	

Figure 10-5 StencilData Register

The stencil writemask is used to control which stencil planes are updated as a result of the test.

The **Stencil** register holds an externally sourced stencil value. It is a 32 bit register of which only the least significant 8 bits are used. The unused most significant bits should be set to zero.

The **Stencil** register must be enabled to update the stencil buffer. If it is disabled then the stencil buffer will only be updated if *ForceLBUpdate* is set in the **Window** register.

10.4.2 Stencil Example

This example sets the stencil unit to use a supplied reference value (0x80) and to test fragments to be LESS than this value. It also sets the stencil planes update function to be Increment if the test passes and the depth test passes (or is not enabled), otherwise it sets the update function to Keep.

```
// Set the localbuffer read and write modes
```

```
// Set the stencil modes
```

```
stencilMode.UnitEnable = GLINT R4_ENABLE
```

```
stencilMode.DPPass = GLINT R4_STENCIL_METHOD_INCREMENT
```

```
stencilMode.DPFail = GLINT R4_STENCIL_METHOD_KEEP
```

```
stencilMode.SFail = GLINT R4_STENCIL_METHOD_KEEP
```

```
stencilMode.CompareFunction = GLINT R4_STENCIL_COMPARE_LESS
```

```
stencilMode.StencilSource = GLINT R4_SOURCE_TEST_LOGIC
```

```
stencilMode.Width = as appropriate
```

```
StencilMode(stencilMode)
```



```
// Set the reference stencil value and set the
// compare and writemasks to 0xFF
stencilData.ReferenceStencil = 0x80
stencilData.CompareMask = 0xFF
stencilData.StencilWriteMask = as appropriate for width of Stencil buffer
stencilData.FCStencil = don't care
StencilData(stencilData)

// Enable the depth test here if required, if not enabled the result of the depth test is set to pass.
```

10.5 Depth Test

The depth (Z) test, if enabled, compares a fragment's depth against the corresponding depth in the depth buffer. The result of the depth test can affect the stencil buffer update if stencil testing is enabled.

This test is only performed if all the preceding tests (bitmask, scissor, stipple, alpha, pixel ownership, stencil) have passed. The comparison tests available are:

Mode	Comparison Function
0	Never
1	Less
2	Equal
3	Less Than or Equal

Mode	Comparison Function
4	Greater
5	Not Equal
6	Greater Than or Equal
7	Always

Table 10.10 Depth Comparison Modes.

The test compares the fragment's depth against a source depth value. If the compare function is LESS and the result is true then the fragment value is less than the source value. The source value can be obtained from a number of places as controlled by a field in the DepthMode register.

Source	Use
DDA (see below)	This is used for normal Depth (Z) buffered 3D rendering.
Depth register	This is used, for instance, in the OpenGL draw pixels function where the host supplies the depth values through the Depth register. Alternatively this is used when a constant depth value is needed, for example, when clearing the depth buffer or 2D rendering where the depth is held constant.
LBSourcData:	Source depth value from the localbuffer: This is used, for instance, in the OpenGL copy pixels function when the depth planes are to be copied to the destination.
Source Depth	This is used by X during the a window copy operation where all the fields in the pixel are moved. This is used in the OpenGL CopyPixels function when the depth planes in the destination are not updated. The depth data will come either from the LBData message of the FCDepth register depending the state of the Fast Clear modes in operation.

Table 10.11 Depth Sources

When using the depth DDA for normal depth buffered rendering operations the depth values required are similar to those required for the color values in the color DDA unit:

Zstart = Start Z Value

dZdYDom = Increment along dominant edge.

dZdX = Increment along the scan line.

The dZdX value is not required for Z-buffered lines.

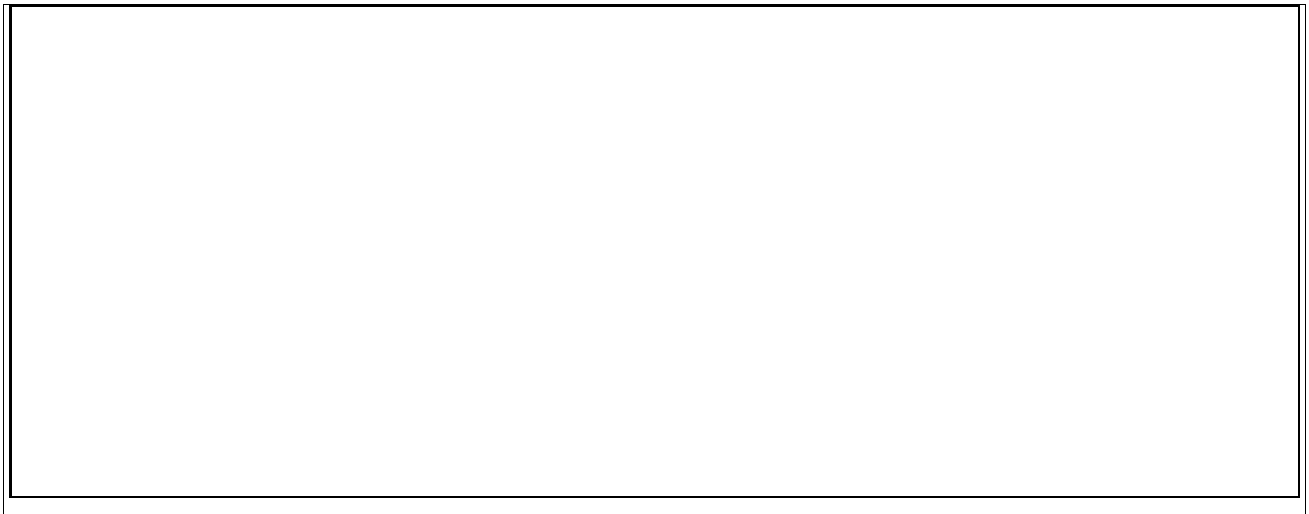


Figure 10-6 Depth Interpolation

The number format for the increment values is 2's complement fixed point integer: 32 bits integer and 16 bits fraction. All the start, derivative and internal data is in this format. This is mapped into the Upper and Lower registers (U and L) as shown below:

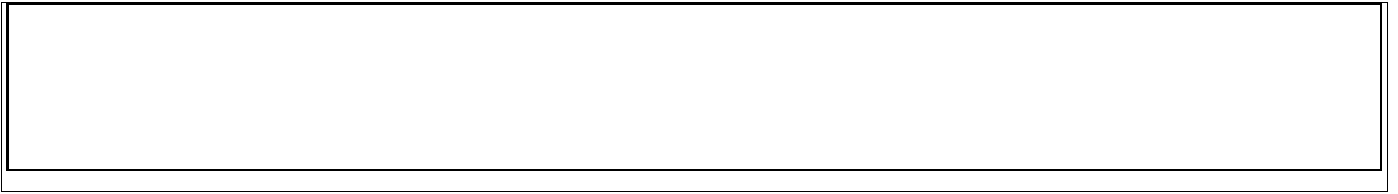


Figure 10-7 Depth Derivative Format.

The depth unit must be enabled to update the depth buffer. If it is disabled then the depth buffer will only be updated if ForceLBUpdate is set in the Window register.

10.5.1 Registers

Operation of the Depth unit is controlled by the **DepthMode** register:

DepthMode

DepthModeAnd

DepthModeOr

Name	Type	Offset	Format
DepthMode	Depth	0x89A0	Bitfield
DepthModeAnd	Depth	0xAC70	Bitfield Logic Mask
DepthModeOr	Depth	0xAC78	Bitfield Logic Mask

Control registers

Bits	Name	Read 15	Write	Reset	Description
0	Enable	✓	✓	x	This bit, when set, enables the depth test and the replacement depth value to depend on the outcome of the test. Otherwise the test always passes and the depth data in the local buffer is not changed.
1	WriteMask	✓	✓	x	This bit, when set enables the depth value in the local buffer to be updated when doing a read-modify-write operation. The byte enables (LB Write) can also be used when the Z value is 16 or 24 bits in size.
2...3	NewDepth Source	✓	✓	x	The depth value to write to the local buffer can come from several places. The options are: 0 = DDA. 1 = Source depth (i.e. read from Local Buffer) 2 = Depth register 3 = LBSourceData register. Only generated when source and destination reads are enabled.
4...6	Compare Function	✓	✓	x	This field selects the compare function to use. The options are: 0 = Never 1 = Less 2 = Equals 3 = Less Equals 4 = Greater 5 = Not Equal 6 = Greater Equal 7 = Always
7...8	Width	✓	✓	x	This field holds the width in bits of the depth field in local buffer. The options are: 0 = 16 bits wide 1 = 24 bits wide 2 = 31 bits wide 3 = 15 bits wide
9	Normalise	✓	✓	x	This bit, when set, will use all 50 bits of the DDA for Z interpolation, even for 24 or less bits of depth. The Width field must be set up to restrict the number of bits used in the comparison operation. When this bit is clear the depth test is compatible with GLINT MX. This bit should be 0 if NonLinearZ is set.
10	NonLinearZ	✓	✓	x	This bit, when set, enables the 32 bit DDA Z value to be encoded in 15, 16 or 24 bits using a non linear pseudo floating point representation. The non linear format is controlled by the following two fields.

¹⁵ Logic Op register readback is via the main register only

11...12	Exponent Scale	✓	✓	x	This field defines how much the exponent should be scaled by. The options are: 0 = scale by 1 1 = scale by 2 2 = scale by 4 3 = scale by 8
13...14	Exponent Width	✓	✓	x	This field defines the number of bits in the depth word to use as exponent bits. The options are: 0 = 1 bit wide exponent field 1 = 2 bits wide 2 = 3 bits wide 3 = 4 bits wide
15...31	Unused	0	0	x	

Notes: The register defines Depth operation. It controls the comparison of a fragment's depth value and updating of the depth buffer. (If the compare function is LESS and result = TRUE then the fragment value is less than the source value.)

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.

Figure 10-8 DepthMode Register.

The single bit writemask is used to control updating all the bits in the depth buffer. Depth values can come from the **Depth** register or Source or Destination Framebuffer reads, or the DDA.

The **Depth** register holds an externally sourced 32 bit depth value. If the depth buffer holds less than 32bits then the user supplied depth value is right justified to the least significant end of the register. The unused most significant bits should be set to zero.

The DDA and other registers are shown below (note the increment values are split into two registers):

Register	Description
ZStartU	Depth start value
ZStartL	
dZdxU	Depth derivative per unit X
dZdxL	
dZdyDomU	Depth derivative per unit Y, dominant edge, or along a line.
dZdyDomL	

Table 10.12 Depth Interpolation Registers.

10.5.2 Depth Example

Rendering a Gouraud shaded depth buffered trapezoid.

```
// Set the localbuffer read and write modes
// Set the depth mode

depthMode.UnitEnable = GLINT R4_ENABLE
```

```
depthMode.WriteMask = 1
depthMode.NewDepthSource = GLINT R4_NEW_DEPTH_SOURCE_DDA
depthMode.CompareMode = GLINT R4_DEPTH_COMPARE_MODE_LESS
DepthMode(depthMode)
// Load the depth start values and deltas for
// dominant edge and the body of the trapezoid

ZStartU() // Load upper and lower start values
ZStartL()
dZdxU() // Load upper and lower dZdX deltas
dZdxL()
dZdyDomU() // Load upper and lower dominant edge deltas
dZdyDomL()
// Enable unit in Gouraud shading mode
colorDDAMode.UnitEnable = GLINT R4_ENABLE
colorDDAMode.Shade = GLINT R4_GOURAUD_SHADE_MODE
ColorDDAMode(colorDDAMode)
// 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
dBdYDom()
// Render primitive
```

11

Texture Mapping

Texture Mapping memory management was introduced in Volume I, section 4.5 - Texture Mapping. The following pages describe the process from the graphics programming point of view. For a discussion of the theory and practice of texture mapping, see the *OpenGL Specification* and the *OpenGL Programming Guide*.

For each fragment within a primitive, texture mapping involves the following steps:

1. calculate the perspective correct texture coordinates for each fragment
2. calculate the level of detail for mipmapping
3. convert texture coordinates into memory indices
4. load texels into primary cache
5. format cache data into texels for filtering
6. check color values and optionally replace a range with alpha values to indicate transparency
7. filter texels from cache based on color components
8. composite the color and texel values with constant color values to produce a final texture value.

These fall into several different phases of operation:

1. Coordinate interpolation and perspective correction
2. Memory indexing
3. Cache loading
4. Alpha and texture filtering and border color
5. Texel compositing
6. Color value calculation and application including lighting effects and application modes

11.1.1 Compatibility with Earlier Chipsets

- Color interpolation is largely unchanged although **TextureAddressMode** is now named **TextureCoordMode** and *TextureLODBias* -S and -T need to be set to 0 to be compatible with GLINT MX.
- Level of Detail calculations now use **TextureFilterMode** instead of **TextureReadMode**. Supported texels must be 4, 8 or 16 bpp - 1, 2 and 4 bpp texels are not supported.
- **TextureReadMode** is not backward compatible with the MX chipset.
- LUT control registers have been consistently renamed (**LUT[0...15]**, **LUTAddress**, **LUTIndex**, **LUTData**, **LUTTransfer**, **LUTMode**)
- The **TextureColorMode** register has been renamed **TextureApplicationMode** and the Color and Alpha data are managed separately during compositing and application.

- **TextureFilterMode enable** must be set (=1) when texture mapping is enabled. The enable bit works in conjunction with the *TextureEnable* bit in the **Render Command**.

11.2 Texture Co-ordinate Generation

To generate the texture addresses, DDAs are used to interpolate the texture coordinates over a trapezoid or line primitive.

There are two general modes of operation: 2D and 3D. In 3D mode, the task divides into the following steps:

- interpolate the texture coordinates (S, T, Q) using the DDA units
- perspective correction of the coordinates by calculating S/Q and T/Q
- level of detail calculation
- wrap the corrected coordinates (s, t) using mirror, repeat or clamp operations to map the coordinates into the range 0.0 to 1.0 (u, v)
- pass the resulting coordinates (u, v) to the texture read unit.

For the 2D mode, the perspective correction stage is omitted, the wrap operation is always a repeat operation and no level of detail is performed.

In R4, per-pixel perspective correction is only available in texture 0 (see the **TextureIndexMode** register in the *GLINT R4 Reference Guide* for further information).

Note: This means that while per-poly mipmapping is available in both textures, per-pixel mipmapping is only available in one.

11.2.1 Calculate texture coordinates

Coordinate interpolation can be either 2D or 3D (set in the **TextureIndexMode** register):

For 2D operations the step or span messages trigger interpolation of the S and T coordinates (Q, S1, T1 and Q1 are not used). This is used for tiled fills, characters and icons, arbitrary large stipple patterns, color index dithering etc.

For 3D operations, **TextureCoordMode** interpolates two sets of texture coordinates (S, T and Q and S1, T1, Q1) and corrects them for perspective and range before they are used for Cache loading.

The coordinates can be used for (a) determining the Level of Detail for MIP mapping, or (b) calculating a 3D texture coordinate.

When used for LOD, the S, T and Q values are applied as a set of linked coordinates to the current fragment, while S1, T1 and Q1 are automatically offset in **dY** to track the coordinates in the adjacent fragment.

When used for 3D texturing, the Delta unit allocates S, T, Q and R as a set of linked coordinates to S, T, Q and S1. T1 is ignored and Q1 is a copy of Q.

The S, T and Q parameters are interpolated in DDA units in the same way as other interpolants: the 9 control registers: **SStart**, **dSdx**, **dSdyDom**, **TStart**, **dTdx**, **dTdyDom**, **QStart**, **dQdx** and **dQdyDom** hold the start, **dX** and **dYDom** parameters for S, T and Q. The values of S, T and Q at each vertex are used to calculate the gradient values in much the same way as the color gradients when Gouraud shading.

The fixed point format of these registers can be defined as you wish but must be internally consistent - the divide operation yields consistent internal results. One method of ensuring that the full range of accuracy available in the DDAs is used but not exceeded (the DDAs

clamp if the range is exceeded) is to normalize the S, T, Q values before calculating the gradient values. For example, for a triangle primitive this involves finding the maximum absolute value of the 9 register values defined at the vertices, and scaling the other 8 values appropriately.

11.2.1.1 Perspective Correction

At each pixel there is a division operation to achieve perspective correction of the texture coordinates and derive the s, t coordinates used to index the texture map through the equations:



After the division, the s, t coordinates are wrapped to lie in the range 0.0 to 1.0 inclusive (and therefore within the range of the defined texture map). The wrapped coordinates are denoted as u, v. These are used to index the raw texel data in memory.

Automatic perspective correction is only available in texture 0.

11.2.2 Level of Detail calculation

The Level Of Detail (LOD) calculates the approximate area a fragment projects onto the texture map. The LOD value is then used:

- To select between the minification and magnification filter modes provided in the **TextureReadMode** register.
- The one or two texture maps to use when mipmapping.
- The between-maps interpolation factor if the mipmapping requires two maps.

The LOD calculation requires the dSdy, dTdy and dQdy values to proceed. These are not supplied by the onboard Delta unit or Gamma accelerator so must be provided by the Texture unit.

Note: To support both mipmapped textures the polygon LOD calculation must be user-supplied. Trilinear filtering is not supported with multi-texture.

The *EnableDY* bit in the **TextureCoordMode** register selects the data source for the calculation. If the *EnableDY* bit is *not* set the **dSdy**, **dTdy** and **dQdy** values can be provided externally by writing into the corresponding registers.

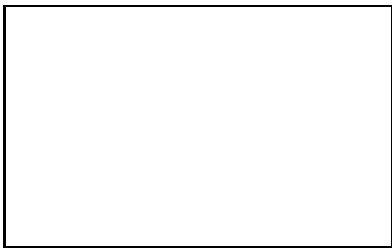
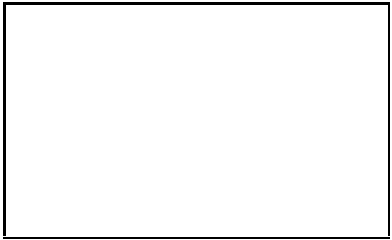
The LOD calculation itself is enabled by the *EnableLOD* bit in the **TextureCoordMode** register. When this bit is clear a constant LOD from the **LOD** register is used (when it is required by **TextureReadMode**). The format is unsigned 4.8 fixed point and can be interpreted as follows: the integer part selects the higher resolution map of the pair to use with 0 using the map at the address given by **TextureBaseAddr[0]** register; the fraction gives the between map interpolation coefficient measured from the higher resolution map selected.

Lod0 is the LOD value calculated as described above. This always relates to texture 0. **Lod1** is a user-supplied value relating to texture 1. Both LOD values can be clamped using **LODRange0** and **LODRange1** respectively. LOD values can be further clamped or constrained by setting the width and height values in **TextureCoordMode**, biased in

TextureIndexMode and clamped in **TextureReadMode**. These constraints allow large textures to be loaded at a low resolution and gradually, by continuous clamping, raised to its final resolution without "popping" artefacts.

11.2.2.1 Texture Coordinate Wrapping Modes

Three wrapping modes are available - Clamp, Repeat and Mirror - and s and t can be wrapped individually. The selected mode is held in the *WrapS* and *WrapT* fields in the **TextureCoordMode** register, and in the *WrapU* and *WrapV* fields in the **TextureIndexMode** register. The wrapping modes are listed in the register descriptions in the *Reference Guide*.

Wrapping Mode	Description
Clamp	<p>This tests the coordinate against 1.0 and if the coordinate is larger sets the coordinate to 1.0. Similarly if the coordinate is less than 0.0 it is set to 0.0.</p> <p>This causes texels outside of the texture map to be set to the edge values.</p> 
Repeat	<p>The integer part of the coordinate is discarded just to leave the fractional part. The Repeat mode creates a saw-tooth transfer function, which as the name suggests, causes the texture pattern to be repeated (i.e. tiled) over the polygon. Abutting edges are from opposite sides of the texture map so unless care is taken a discontinuity may be seen.</p> 

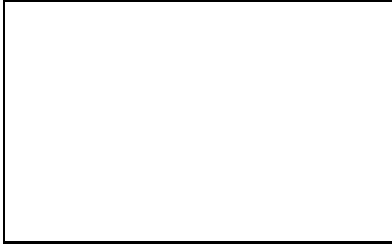
Mirror	<p>This is similar to Repeat, but when the integer part is odd the value (1.0 - fraction) is used instead of just the fraction. This creates a triangle transfer function, which has the advantage that butting edges always match.</p> 
---------------	--

Table 4-1 Texture Wrapping - Repeat and Clamp modes are as defined by OpenGL.

11.2.2.2 Texture Address Registers

The following registers set up the texture interpolation deltas :

Register	Description
Sstart	S start value
DSdx	S derivative per unit X
DSdyDom	S derivative per unit Y, dominant edge
Tstart	T start value
DTdx	T derivative per unit X
dTdyDom	T derivative per unit Y, dominant edge
Qstart	Q start value
DQdx	Q derivative per unit X
DQdyDom	Q derivative per unit Y, dominant edge
DSdy	S derivative per unit Y
DTdy	T derivative per unit Y
DQdy	Q derivative per unit Y

Table 11.2 Texture Interpolation Registers

11.2.2.3 Mipmapping

A mipmap is an ordered set of arrays representing the same image. Each array has half the linear resolution of the preceding one. This technique allows minification filtering to occur with a constant time overhead irrespective of the size of the projected area.

The first filter name for mipmapping in the *MinFilter* field specifies the filtering to be done on a level, and the second filter name specifies the filtering to be done between levels.

Mipmap is enabled by setting the *MipMapEnable* bit (bit 20) in the **TextureIndexMode** register. Other Mipmap parameters are also controlled by **TextureIndexMode**, including Magnification and Minification filter types.

Note: Both single and dual texture mipmaps are supported but the calculation must be implemented manually.

11.2.3 Texture Read

The texture read phase fetches and formats texel data. This involves taking the u, v coordinates generated by the texture address unit and possibly the LOD value and calculating the physical address in the local buffer where the texture is stored. The texture information (texels) is read and forwarded for Texture Filtering. The interpolation coefficients (if any are needed) are derived from the u, v coordinates (and possibly the LOD value) and passed on as well. The texture cache management process is described in Volume I, Section 4-6 - Primary Cache.

The Texture Read operation is controlled by **TextureReadMode0** and **TextureReadMode1** which are the same. However most modes cannot be enabled in both caches at the same time. The supported combinations are:

- One nearest or linear filtered texture using both halves of the cache to achieve higher cache hit rates on larger texture maps or polygons.
- Any two independent nearest or linear filtered textures, one per half of the cache.
- One automatically (or per pixel) mip mapped texture (always texture 0) using both halves of the cache to store alternate levels of the mip map.
- One 3D texture map using both halves of the cache to store alternate slices of the 3D volume.
- Two independent mip mapped textures where the minification filters only use texels from one level at a time (i.e. the filter are NearestMipNearest or LinearMipNearest). Each texture uses half the cache.

There are no interlocks to prevent the user selecting a non-supported combination and in this case the mode settings in **TextureReadMode0** take priority.

TextureReadMode0 TextureReadMode0And TextureReadMode0Or

Name	Type	Offset	Format
TextureReadMode0	Texture	0xB400	Bitfield
TextureReadMode0And	Texture	0xAC30	Bitfield Logic Mask
TextureReadMode0Or	Texture	0xAC38	Bitfield Logic Mask

Control registers

Bits	Name	Read 16	Write	Reset	Description
0	Enable	✓	✓	x	When set causes any texels needed by the fragment to be read. This is also qualified by the TextureEnable bit in the <i>Render</i> command.
1...4	Width	✓	✓	x	This field holds the width of the map as a power of two. The legal range of values for this field is 0 (map width = 1) to 11 (map width = 2048). This is only used when Texture3D is enabled and then is only used for cache management purposes and <i>not</i> for address calculations.

¹⁶ Logic Op register readback is via the main register only

5...8	Height	✓	✓	x	This field holds the height of the map as a power of two. The legal range of values for this field is 0 (map height = 1) to 11 (map height = 2048). This is only used when Texture3D is enabled and then is only used for cache management purposes and <i>not</i> for address calculations.
9...10	TexelSize	✓	✓	x	This field holds the size of the texels in the texture map. The options are: 0 = 8 bits 1 = 16 bits 2 = 32 bits 3 = 64 bits (Only valid for spans)
11	Texture3D	✓	✓	x	This bit, when set, enables 3D texture index generation. The CombinedCache mode bit should not be set when 3D textures are being used.
12	Combine Caches	✓	✓	x	This bit, when set, causes the two banks of the Primary Cache to be joined together, thereby increasing the size of a single texture map which can be efficiently handled.
13...16	MapBaseLevel	✓	✓	x	This field defines which TextureBaseAddr register should be used to hold the address for map level 0 when mip mapping or the texture map when not mip mapping. Successive map levels are at increasing TextureBaseAddr registers upto (and including) the MapMaxLevel (next field). 3D textures always use TextureBaseAddr0.
17...20	MapMaxLevel	✓	✓	x	This field defines the maximum TextureBaseAddr register this texture should use when mip mapping. Any attempt to use beyond this level will clamp to this level.
21	LogicalTexture	✓	✓	x	This bit, when set, defines this texture or all mip map levels, if mip mapping, to be logically mapped so undergo logical to physical translation of the texture addresses.
22	Origin	✓	✓	x	This field selects where the origin is for a texture map with a Linear or Patch64 layout. The options are: 0 = Top Left. 1 = Bottom Left A Patch32_2 or Patch2 texture map is always bottom left origin.
23...24	TextureType	✓	✓	x	This field defines any special processing needed on the texel data before it can be used. The options are: 0 = Normal. 1 = Eight bit indexed texture. 2 = Sixteen bit YVYU texture in 422 format. 3 = Sixteen bit VYUY texture in 422 format..

25...27	ByteSwap	✓	✓	x	This field defines the byte swapping, if any, to be done on texel data when it is used as a bitmap. This is automatically done when spans are used. Bit 27, when set, causes adjacent bytes to be swapped, bit 26 adjacent 16 bit words to be swapped and bit 27 adjacent 32 bit words to be swapped. In combination this byte swap the input (ABCDEFGH) as follows: 0 ABCDEFGH 1 BADCFEHG 2 CDABGHEF 3 ABCDEFGH 4 EFGHABCD 5 FEHGBADC 6 GHEFCDAB 7 HGFEDCBA
28	Mirror	✓	✓	x	This bit, when set will mirror any bitmap data. This only works for spans.
29	Invert	✓	✓	x	This bit, when set will invert any bitmap data. This only works for spans.
30	OpaqueSpan	✓	✓	x	This bit, when set allows the color of each pixel in the span to be either foreground or background as set by the supplied bit masks. If this bit is 0 then any supplied bit masks are anded with the pixel mask to delete pixels from the span.
31	Reserved	0	0	x	

- Notes:
- The unit is controlled by the *TextureReadMode0* and *TextureReadMode1* registers for texture 0 and texture 1 respectively. Not all combinations of modes across both registers are supported and where there is a clash the modes in *TextureReadMode0* take priority. For per pixel mip mapping the *TextureRead0* and *TextureReadMode1* register should be set up the same as should the *TextureMapWidth0* and *TextureMapWidth1* registers.
 - N.B. The layout and use of the *TextureReadMode* register is not compatible with GLINT MX: 1, 2, and 4 bit textures are no longer supported.
 - The OpaqueSpan field determines how constant color spans are written (recall the Render command selects between constant color or variable color spans). Transparent spans just use one color for the foreground pixels and the background pixels are not written. Opaque spans write to foreground and background pixels using *FBlockColor* for the foreground pixels and *FBlockColorBack* for the background pixels. This bit should be set to 0 for performance reasons when foreground/background processing is not required.
 - 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.

Figure 11-1 TextureReadMode Register

11.2.4 Filter Modes

All the filter modes of OpenGL are supported, that is:

Minification	Magnification
Nearest	Nearest
Linear	Linear
NearestMipMapNearest	
NearestMipMapLinear	
LinearMipMapNearest	
LinearMipMapLinear	

“Minification” is the name given to the filtering situation where multiple texels map to a single fragment, while magnification is the name given to the filtering situation where only a portion of a single texel maps to a single fragment.

“Nearest” is the simplest form of filtering where the nearest texel to the texture coordinate location is selected.

“Linear” is a more sophisticated filtering algorithm which is dependent on the type of primitive. For lines (which are 1D), it involves linear interpolation between the two nearest texels. For polygons and points which are considered to have finite area, linear is in fact bi-linear interpolation which interpolates between the nearest 4 texels.

11.2.4.1 Texture Patching

In GLINT R4 the data part of the primary cache is managed by the **TextureFilterMode** register, while the tag part is managed by the **TextureReadMode** register. The Filter functionality includes data formatting and alpha mapping.

11.2.4.2 Primary Cache

The primary cache holds the texel data in 8, 16 or 32 bits per texel format. The cache is divided up into 8 banks and there is a fixed relationship between a texel's position in the texture map and which bank of cache it must be stored in. The 8 banks are assigned depending on the type of texture mapping being done:

Single bilinear	The texture map is stored in both banks of the cache. This is achieved by connecting the output of the second bank's register files to the corresponding register files in bank 0. This is controlled by the <i>CombineCaches</i> bit in TextureFilterMode . This allows the full size of the cache to be used on a single texture, so a larger texture map can be handled before scanline coherency starts to break down, with the consequential loss of performance.
Dual bilinear	Texels from texture map 0 are stored in banks 0...3 and texels from texture map 1 are stored in banks 4...7.
Mip mapping	Even mip maps are stored in banks 0...3, odd mip maps are stored in banks 4...7.
3D texture maps	Texels with an even k coordinate (i.e. the third coordinate) are in banks 0...3 and maps with an odd k coordinate are in banks 4...7.

Note: It is not possible to perform dual texturing and perspective correction at the same time.

The texels within a map have a fixed allocation to the cache banks as shown by the following diagram:



where T0...T3 represents the cache banks and the numbers in brackets are the coordinate of the texel in the map.

Storing the texture map in memory with one row following the next can give poor access times when scanning along a column due to the page breaks. This does not apply if the texture map is smaller than the page size.

When the texture map is significantly larger than the page size, make access time less dependent on scanning direction by patching the texture map. This ensures that a 2D region of the map is stored in one page.

All the texels within a word are always sequential along a row and a patch is 16x16 words, hence the patch size in texels varies from 16x16 (for 32 bit texels) to 512x16 (for 1 bit texels). If packed texture maps are required then the packing can be done automatically during texture download¹⁷, or must be done by the host if the local buffer bypass is used. Note that some wastage of the memory space will occur if the texture map dimensions are not an integer multiple of the patch size.

¹⁷ See Volume I, Section 4.5.1.2 - Patch Layout Rules

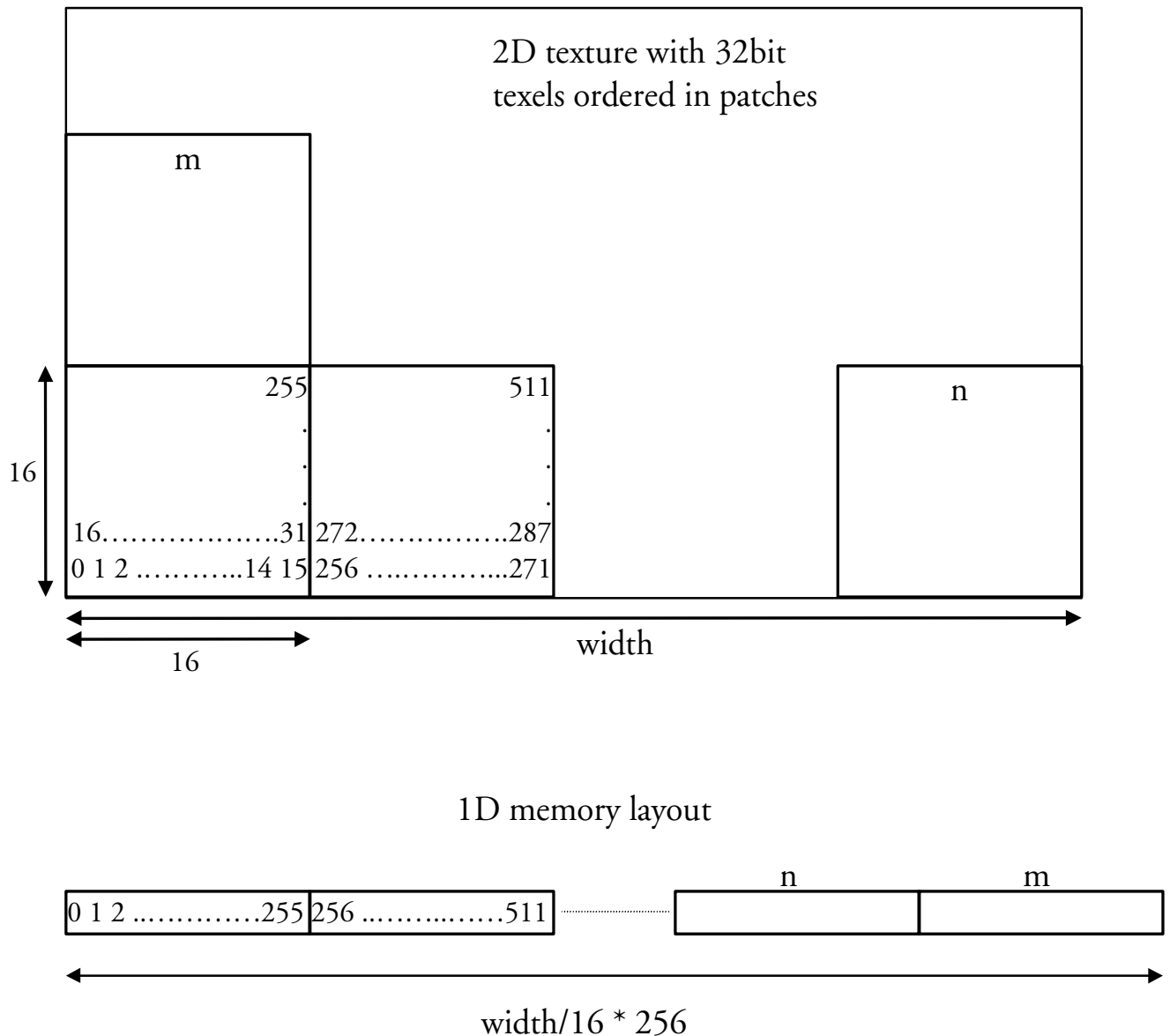


Figure 11-2 Texture Patch Example

- *Map Width:* The patch mode is only useful when the width of the map exceeds 16 words.
- *Map Height:* The patch mode works best when the height of the map is greater than 16 texels. For maps which are less than this in height a portion of the patch will not be used so the texel data will be spread out in memory. Consider a 1K word x 4 texture map. This will occupy a quarter of the patch memory so 16K words need to be set aside for 4K of texels. Moving between rows will occur without page breaks, where as in the non patch case it would incur a page break. It is possible to interleave

4 such maps so getting the benefit of less page breaks without the cost of the additional memory.

- **Filter and MapType:** The filter (Nearest or Linear) and map type (1D or 2D) determine how many addresses are generated.

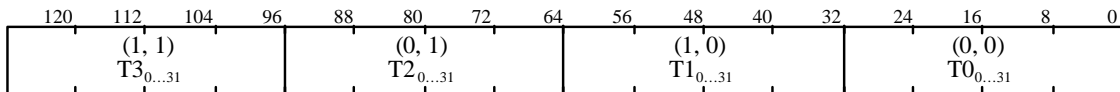
A texel on the map has the integer coordinates i, j and these are calculated from u, v and the width and height values. These integer coordinates are guaranteed to lie on the texture map (excluding the border texels, if present), so for the nearest filter mode the texel is just read and used.

For the linear filter mode and 2D MapType the four texels (i, j) , $(i+1, j)$, $(i, j+1)$ and $(i+1, j+1)$ are read, with obvious reductions for the 1D MapType. The coordinates $(i+1)$ and/or $(j+1)$ may not lie on the texture map. If the texture map has a border (specified in the *Border* field) then the appropriate texel from the texture map is read, otherwise texel is taken from the **BorderColor0** or **BorderColor1** registers. The texel color stored in this register is in 8:8:8:8 format.

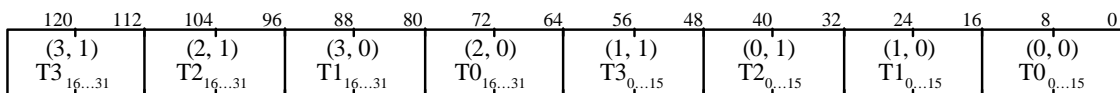
Texture maps are preferably stored in memory as a 2x2 patch so that the texels in the patch are in the same memory word. When texture maps are not in this format (i.e. the memory layout is Linear or Patch64) the Texture Read Unit passes the texel data on in the patched format.

The following diagram shows the layout of texels assumed by this unit when loading up the cache. This exactly matches the layout in memory when one of the 2x2 patch modes are used.

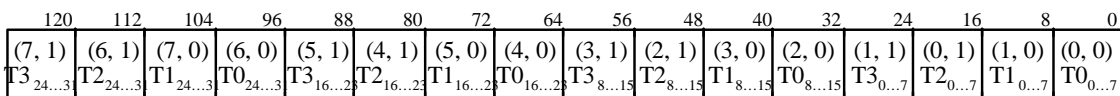
32 bits per texel



16 bits per texel



8 bits per texel



11.2.5 Texel Formatting

Texel formatting is controlled by the **TextureFilterMode** register:

8	Combine Caches	✓	✓	x	This bit, when set, combines both banks of the cache so they are used for texture 0. This is an optimisation and allows larger textures to be handled before scanline coherency starts to break down.
9...12	Format1	✓	✓	x	This field selects the format of the texel data T4...T7. The options are 0 = A4L4 1 = L8 2 = I8 3 = A8 4 = 332 5 = A8I8 6 = 5551 7 = 565 8 = 4444 9 = 888 10 = 8888 or YUV
13	ColorOrder1	✓	✓	x	This bit selects the color component order of the texel data T4...T7. The two options are: 0 = AGBR 1 = ARGB
14	AlphaMap Enable1	✓	✓	x	This bit, when set, enables the alpha value of texels T4...T7 to be forced to zero based on testing the color values.
15	AlphaMap Sense1	✓	✓	x	This bit selects if the alpha value for texels T4...T7 should be set to zero when the colors are in range or out of range. The options are: 0 = Out of range 1 = In range
16	AlphaMap Filtering	✓	✓	x	This bit, when set, will allow the alpha mapped texels (AlphaMapEnable must be set) to cause the fragment to be discarded depending on the comparison of the number of texels to be alpha mapped with the following three limit fields.
17...19	AlphaMap FilterLimit0	✓	✓	x	This field holds the number of alpha mapped texels in the group T0...T3 which must be exceeded for the fragment to be discarded.
20...22	AlphaMap FilterLimit1	✓	✓	x	This field holds the number of alpha mapped texels in the group T4...T7 which must be exceeded for the fragment to be discarded.
23...26	AlphaMap FilterLimit01	✓	✓	x	This field holds the number of alpha mapped texels in the group T0...T7 which must be exceeded for the fragment to be discarded.
27	MultiTexture	✓	✓	x	This bit, when set, prevents the Alpha Map Filtering logic from testing the I4 interpolant and maybe disregarding the alpha map result of T0...T3 or T4...T7. This bit should be set for multi texture operation when alpha map filtering is required. It should be clear otherwise.

28	ForceAlphaToOne0	✓	✓	x	This bit, when set, will force the alpha channel of T0...T3 to be set to 1.0 (255) regardless of the color format or the presence of a real alpha channel.
29	ForceAlphaToOne1	✓	✓	x	This bit, when set, will force the alpha channel of T4...T7 to be set to 1.0 (255) regardless of the color format or the presence of a real alpha channel.
30	Shift0				This bit, when set, causes the conversion of T0...T3 for color components less than 8 bits wide to be done by a shift operation, otherwise a scale operation is needed. The shift operation is useful where the exact color (after dithering) is to be preserved for flat shaded areas, such as in a stretch blit.
31	Shift1				This bit, when set, causes the conversion of T4...T7 for color components less than 8 bits wide to be done by a shift operation, otherwise a scale operation is needed. The shift operation is useful where the exact color (after dithering) is to be preserved for flat shaded areas, such as in a stretch blit.

Notes: 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.

For most texel formats the data in the cache is held in the raw memory format. The two exceptions to this are 8 bit indexed textures and YUV422 format textures. In both these cases the original texel data is converted into 32 bit AGBR format before being loaded into the cache.

The first task is to extract the byte or short - this is given by the bottom two bits of the address for this cache channel. The second task is to isolate the individual color components from the texel data. The following table 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 data. The least significant bit position is 0. The number 255 indicates this component is hardwired to this value.

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 TextureFilterMode message, 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 format where the actual bit fields extracted need to be modified as well because the R and B components are differing widths.

Format	Color Order	Name	Width	R	G	B	A
0	ABGR	A4L4	8	4@0	4@0	4@0	4@4
1		L8	8	8@0	8@0	8@0	255
2		I8	8	8@0	8@0	8@0	8@0
3		A8	8	255	255	255	8@0
4		332	8	3@0	3@3	2@6	255
5		A8I8	16	8@0	8@0	8@0	8@8

6		5551	16	5@0	5@5	5@10	1@15
7		565	16	5@0	6@5	5@11	255
8		4444	16	4@0	4@4	4@8	4@12
9		888	32	8@0	8@8	8@16	255
10		8888 or YUV	32	8@0	8@8	8@16	8@24
0	ARGB	A4L4	8	4@0	4@0	4@0	4@4
1		L8	8	8@0	8@0	8@0	255
2		I8	8	8@0	8@0	8@0	8@0
3		A8	8	255	255	255	8@0
4		332	8	3@5	3@2	2@0	255
5		A8I8	16	8@0	8@0	8@0	8@8
6		5551	16	5@10	5@5	5@0	1@15
7		565	16	5@11	6@5	5@0	255
8		4444	16	4@8	4@4	4@0	4@12
9		888	32	8@16	8@8	8@0	255
10	8888 or YUV	32	8@16	8@8	8@0	8@24	

Table 4.2.5 - Texture Color Modes

The alpha channel can be forced to 1.0 to override the alpha value, when the alpha channel in the texel data is to be ignored (this is independent of the color conversion mode - see next paragraph).

When an extracted component is less than 8 bits wide it is made up to 8 bits by scaling or shifting. Scaling is preferred for normal 3D usage, however when the texture maps are being used for 2D operations (such as stretch blits) the shift method is preferred as it will maintain the same color during bilinear filtering over regions of constant color.

Scaling is done by replicating the extracted component from the most significant end towards the least significant end of the byte. 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	B1	B0	B2	B1

11.2.6 Lookup Table (LUT)

The LUT functionality includes:

- Translating color data on a color-by-color basis (for, e.g., un-Gamma correcting)
- Mapping CI data to 32-bit RGBA
- Conversion of span pixel data from 8bpp to 8, 16 or 32 bpp, or RGB conversion from 32bpp to 32bpp.
- Sourcing pattern fill data
- Applying motion compensation to video streams

- Map 8 bit CI texel data to 32bpp RGBA texel data needed for Texture Filter functiona..

11.2.6.1 Loading the Texel LUT

The LUT is 256 entries deep by 32 bits wide. The bottom 16 locations are directly accessed by the **LUT[0...15]** registers, and can be read back directly. The remaining entries are accessed in another way.

The LUT can be loaded via the auto incrementing register writes or from the local buffer. The ability to load the entire LUT from the local buffer by writing to two registers greatly reduces the burden on the host of managing the LUT. The LUT data can be written into the local buffer initially either via the bypass or (better) using the normal texture download mechanism.

11.2.6.2 Loading the LUT via auto incrementing registers

The start index in the LUT is written to the **LUTTransfer** register. The bottom 8 bits of the data give the index. Every subsequent write to the **LUTData** register loads the LUT with the data and increments the index. Reading back the **LUTIndex** register will return the incremented index value.

11.2.6.3 Loading the LUT from the local buffer.

The local buffer address where the LUT is held is in the **LUTAddress** register. The start index and number of words to fill in the LUT are given in the **LUTTransfer** register with the index in the bits 0...7 and the count in bits 8...16. The write to the **LUTTransfer** register starts the transfer. A count of zero loads zero words into the LUT so this effectively disables the loading operation. The transfer wraps around in the LUT if necessary.

The **LUTAddress** and **LUTTransfer** registers are not changed by the transfer and both can be read back. The restoration of these registers after a context switch automatically restores the LUT to it's previous contents. This assumes that the LUT hasn't been loaded piecemeal or via one of the other mechanisms and that the LUT data in the local buffer is still valid. If these conditions do not hold then the LUT will have to be restored manually.

The LUT data is only held in the bottom 32 bits of the local buffer memory and the red component is in the least significant byte.

11.2.6.4 Reading the LUT.

To read the LUT first read the **LUTIndex** register. As well as returning the current LUT index (as noted above) it also has the side effect of setting an *Index* counter to zero. The *Index* counter is only used during readback. Each subsequent read from the **LUTData** register returns the LUT data at the *Index* and increments the Index counter. The Index counter wraps from 255 to 0.

11.2.7 Texture Filtering and Alpha Mapping

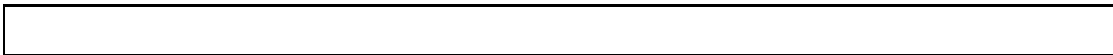
The required texture filter mode is set up in the **TextureReadMode** register as already outlined. Texture filtering must be enabled separately via the **TextureFilterMode** register. This register has the following fields:

Name	Width	Function
Enable	1	Enables texture filtering to occur when set.
AlphaMapEnable	1	Enables Alpha map processing to occur when set
AlphaMapSense	1	When clear the alpha map sense is Include, otherwise it is exclude.

Table 11.13 Texture Filtering

Alpha Map processing provides a mechanism where the color of the input texels are tested against a range of colors and the alpha value of the texel is set based on the outcome of the test. This subsequently allows an Alpha Test to be done, however it doesn't rely on the presence of an alpha channel in the texture map.. Direct3D and Quick Draw 3D both have the notion of a transparent color in the texture map for doing cut-outs so the alpha map operation allows the Alpha Test to be used.

The alpha map test is given by:



where C_L is the lower chroma value held in the **TextureChromaLower** register, C_U is the upper chroma value held in the **TextureChromaUpper** register and T is the input texel value. 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 **TextureChromaLower** and **TextureChromaUpper** registers hold the color bytes with the red component in the lower byte, then the green byte and finally the blue byte.

The alpha map test is only enabled when **TextureFilterMode** *enable* bit is set and the *AlphaMapEnable* bit in **TextureFilterMode** is set. The sense of the alpha map test (when enabled) is controlled by the *AlphaMapSense* bit and the effect of this is tabulated below:

AlphaMap Test Enabled	Test Result	AlphaMapSense	Action
N	X	X	Alpha value unchanged.
Y	False	Include	Alpha set to 0x00.
Y	True	Include	Alpha set to 0xFF.
Y	False	Exclude	Alpha set to 0xFF.
Y	True	Exclude	Alpha set to 0x00.

Table 11.14 AlphaMapTest Enabled

11.2.8 Texture Color Compositing

During compositing, the Color, Texel0 and Texel1 values are combined with constant color value(s) held in registers to produce a combined Texture value for the texel, which is passed on to the Application phase.

The whole unit operation is enabled and disabled by the **TextureCompositeMode** register. It has the following format:

Bit No.	Name	Description
0	Enable	When set causes the compositing operation to be calculated and to replace the texture0 value sent to the next unit, otherwise the texture value remains unchanged. This enable is also qualified by the TextureEnable bit in the PrepareToRender message.

The compositing is controlled by five registers:

Register	Channels	Stage
TextureCompositeColorMode0	RGB	0
TextureCompositeColorMode1	RGB	1
TextureCompositeAlphaMode0	A	0
TextureCompositeAlphaMode1	A	1

These registers all have the same format:

Bit No.	Name	Description
0	Enable	When set causes the output to be calculated as defined by the fields in this register, otherwise the texel0 data is passed through for stage0 and Output data is passed through for stage 1.
1...4	Arg1	This field selects the source value for Arg1. The options are: 0 = Output.C of the previous stage or height if the first stage 1 = Output.A of the previous stage or height if the first stage 2 = Color.C 3 = Color.A 4 = TextureCompositeFactorn.C 5 = TextureCompositeFactorn.A 6 = Texel0.C 7 = Texel0.A 8 = Texel1.C 9 = Texel1.A 10 = Sum of the color components of the previous stage or 0 if the first stage. where n is the same as the message suffix and C is the RGB or A depending on the channel. height is defined as $\text{clamp}(\text{Texel0.A} - \text{Texel1.A} + 128)$
5	InvertArg1	This bit, if set, will invert the selected Arg1 value before it is used.

6...9	Arg2	<p>This field selects the source value for Arg2. The options are:</p> <ul style="list-style-type: none"> 0 = Output.C of the previous stage or height if the first stage 1 = Output.A of the previous stage or height if the first stage 2 = Color.C 3 = Color.A 4 = TextureCompositeFactor.C 5 = TextureCompositeFactor.A 6 = Texel0.C 7 = Texel0.A 8 = Texel1.C 9 = Texel1.A 10 = Sum of the color components of the previous stage or 0 if the first stage. <p>where n is the same as the message suffix and C is the RGB or A depending on the channel. height is defined as $\text{clamp}(\text{Texel0.A} - \text{Texel1.A} + 128)$</p>
10	InvertArg2	This bit, if set, will invert the selected Arg2 value before it is used.
11...13	I	<p>This field selects what is used as the interpolation factor when the Operation field is set to Lerp, for example. The options are:</p> <ul style="list-style-type: none"> 0 = Output.A of the previous stage or 0 if the first stage 1 = Color.A 2 = TextureCompositeFactor.A 3 = Texel0.A 4 = Texel1.A 5 = Texel0.C 6 = Texel1.C <p>where n is the same as the message suffix and C is the RGB or A depending on the channel.</p>
14	InvertI	This bit, if set, will invert the selected I value before it is used.
15	A	<p>This bit selects which Arg (after any inversion) is to be used as A in the Operation. The options are:</p> <ul style="list-style-type: none"> 0 = Arg1 1 = Arg2

16	B	This bit selects which Arg (after any inversion) is to be used as B in the Operation. The options are: 0 = Arg1 1 = Arg2
17...20	Operation	This field defines how the three inputs (A, B and I) are combined. Note the inputs can be optionally inverted before being combined. The 8 bit inputs are unsigned 0.8 fixed point format, but 255 is treated as if it were 1.0 for the calculations. The possible operations are: 0 = Pass (A) 1 = Add (A + B) 2 = AddSigned (A + B - 128) 3 = Subtract (A - B) 4 = Modulate (A * B) 5 = Lerp (A * (1.0 - I) + B * I) 6 = ModulateColorAddAlpha (A * B + I) 7 = ModulateAlphaAddColor (A * I + B) 8 = AddSmoothSaturate (A + B - A * B) 9 = ModulateSigned (A * B, but A and B are biased 8 bit numbers)
21...22	Scale	This field selects the scale factor to apply to the final result before it is clamped. The options are: 0 = 0.5 1 = 1 2 = 2 3 = 4

11.2.8.1 Texture Application

The Application phase applies the texel values calculated in the previous phases of texturing to the incoming pixel color (generated in the color DDA unit). The function used to combine these two colors is defined in the **TextureApplicationMode** register and includes various types of blend, decal, replacement and modulation for the different APIs.

The available options are split into three types - OpenGL, QuickDraw 3D and Direct3D. The OpenGL options are one of:

- Decal
- Blend
- Modulate
- Replace.

The QuickDraw 3D options are any combination of:

- Decal
- Modulate
- Highlight.

The D3D options are:

- Copy
- Add
- Modulate

- Blend

11.2.8.2 OpenGL Application Modes

The fragment's color is calculated based on the following equations:

Type	Equation
Modulate	$R = C * T$
Decal	$R = C * T + K * (1 - T)$
Blend	$R = C * T + K * (1 - T)$
Replace	Base Format Alpha $R = C * T$ Luminance $R = C * T + K * (1 - T)$ LuminanceAlpha $R = C * T + K * (1 - T)$ Intensity $R = C * T + K * (1 - T)$ RGB $R = C * T + K * (1 - T)$ RGBA $R = C * T + K * (1 - T)$

...where R is the final color after texture has been applied, C is the fragment color (in a Color field), T is the texel value (in the texel field) and K is a constant color stored in a register locally (loaded by the **TextureEnvColor** register). The equations are executed on the four color components in parallel and the suffixes show how the different component values are combined.

The setting of the **TextureApplicationMode** register fields to implement these OpenGL equations is as follows.

Enable is 1, KsEnable, KdEnable are both 0 for all entries and some obvious abbreviations have been used to keep the table width down.

Type	Color fields					Alpha fields				
	A	B	I	Invl	Operation	A	B	I	Invl	Operation
Modulate	C.C	T.C			Modulate	C.A	T.A			Modulate
Decal	C.C	T.C	T.A	N	Lerp	C.A			N	PassA
Blend	C.C	K.C	T.C	N	Lerp	C.A	T.A			Modulate
Replace (Alpha)	C.C				PassA		T.A			PassB
Replace (Luminance)		T.C			PassB	C.A				PassA
Replace (LuminanceAlpha)		T.C			PassB		T.A			PassB
Replace (Intensity)		T.C			PassB		T.A			PassB
Replace (RGB)		T.C			PassB	C.A				PassA
Replace (RGBA)		T.C			PassB		T.A			PassB

So for example, the **TextureApplicationMode** fields for OGL Decal would be set as follows (see the **Value** column):

Bits	Name	Read ¹⁹	Write	Value	Description
0	Enable	✓	✓	1	When set causes the output to be calculated as defined by the fields in this register, otherwise the fragment's data is passed through.
1...2	ColorA	✓	✓	0	This field selects the source value for A. The options are: 0 = Color.C 1 = Color.A 2 = K.C (TextureEnvColor) 3 = K.A (TextureEnvColor)
3...4	ColorB	✓	✓	0	This field selects the source value for B. The options are: 0 = Texel.C 1 = Texel.A 2 = K.C (TextureEnvColor) 3 = K.A (TextureEnvColor)
5...6	ColorI	✓	✓	3	This field selects the source value for I. The options are: 0 = Color.A 1 = K.A (TextureEnvColor) 2 = Texel.C 3 = Texel.A

¹⁹ Logic Op register readback is via the main register only

7	ColorInvertI	✓	✓	0	This bit, if set, will invert the selected I value before it is used.
8...10	Color Operation	✓	✓	4	The possible operations are: 0 = PassA (A) 1 = PassB (B) 2 = Add (A + B) 3 = Modulate (A * B) 4 = Lerp (A * (1.0 - I) + B * I) 5 = ModulateColorAddAlpha (A * B + I) 6 = ModulateAlphaAddColor (A * I + B) 7 = ModulateBIAddA (B * I + A)
11...12	AlphaA	✓	✓	1	This field selects the source value for A. The options are: 0 = Color.C (effectively Color.A) 1 = Color.A 2 = K.C (TextureEnvColor) (effectively K.A) 3 = K.A (TextureEnvColor)
13...14	AlphaB	✓	✓	X	This field selects the source value for B. The options are: 0 = Texel.C (effectively T.A) 1 = Texel.A 2 = K.C (TextureEnvColor) (effectively K.A) 3 = K.A (TextureEnvColor)
15...16	AlphaI	✓	✓	X	This field selects the source value for I. The options are: 0 = Color.A 1 = K.A (TextureEnvColor) 2 = Texel.C (effectively T.A) 3 = Texel.A
17	Alpha InvertI	✓	✓	0	This bit, if set, will invert the selected I value before it is used.
18...20	Alpha Operation	✓	✓	0	This field defines how the three inputs (A, B and I) are combined. The possible operations are: 0 = PassA (A) 1 = PassB (B) 2 = Add (A + B) 3 = Modulate (A * B) 4 = Lerp (A * (1.0 - I) + B * I) 5 = ModulateABAddI (A * B + I) 6 = ModulateAIAddB (A * I + B) 7 = ModulateBIAddA (B * I + A)
21	KdEnable	✓	✓	0	When set this bit causes the RGB results of the texture application to be multiplied by the Kd DDA values. It also enables the Kd DDA to be updated.
22	KsEnable	✓	✓	0	When set this bit causes the RGB results of the texture application (or Kd processing) to be added with the Ks DDA values. It also enables the Ks DDAs to be updated.

23	Motion Comp Enable	✓	✓	X	
----	--------------------	---	---	---	--

11.2.8.3 Apple Texture Application

The fragment's color is calculated based on the following equations (any combination of these operations are allowed and they are done in the order given):

Type	Equation
Decal	If enabled $R_{rgb} = T_a T_{rgb} + (1 - T_a) C_{rgb}$ $R_a = C_a$ else $R_{rgb} = T_{rgb}$ $R_a = T_a C_a$
Modulate	
Highlight	

...where T is the texel color, C is the fragment color (in a Color message), Kd is the diffuse RGB components from the Kd DDA unit, and Ks is the specular RGB components from the Ks DDA unit. The equations are executed on the four color components in parallel and the suffixes show how the different component values are combined.

The final value R is forwarded in the Color field of the active step to the next unit.

The setting of the **TextureApplicationMode** fields to implement these Apple equations is as follows. Enable is 1, KsEnable is set if Modulate is required, KdEnable is set if highlight is required. Some obvious abbreviations have been used to keep the table width down.

Type	Color fields					Alpha fields				
	A	B	I	Invl	Operation	A	B	I	Invl	Operation
Decal enabled	C.C	T.C	T.A	N	Lerp	C.A				PassA
Modulate disabled		T.C			PassB	C.A	T.A			Modulate

So for example, the **TextureApplicationMode** fields for Apple Quickdraw Decal with highlighting but no modulation would be as follows (see the **Value** column):

Bits	Name	Read 20	Write	Value	Description
0	Enable	✓	✓	1	When set causes the output to be calculated as defined by the fields in this register, otherwise the fragment's data is passed through.
1...2	ColorA	✓	✓	0	This field selects the source value for A. The options are: 0 = Color.C 1 = Color.A 2 = K.C (TextureEnvColor) 3 = K.A (TextureEnvColor)
3...4	ColorB	✓	✓	0	This field selects the source value for B. The options are: 0 = Texel.C 1 = Texel.A 2 = K.C (TextureEnvColor) 3 = K.A (TextureEnvColor)
5...6	ColorI	✓	✓	3	This field selects the source value for I. The options are: 0 = Color.A 1 = K.A (TextureEnvColor) 2 = Texel.C 3 = Texel.A
7	ColorInvertI	✓	✓	0	This bit, if set, will invert the selected I value before it is used.
8...10	Color Operation	✓	✓	4	The possible operations are: 0 = PassA (A) 1 = PassB (B) 2 = Add (A + B) 3 = Modulate (A * B) 4 = Lerp (A * (1.0 - I) + B * I) 5 = ModulateColorAddAlpha (A * B + I) 6 = ModulateAlphaAddColor (A * I + B) 7 = ModulateBIAddA (B * I + A)
11...12	AlphaA	✓	✓	1	This field selects the source value for A. The options are: 0 = Color.C (effectively Color.A) 1 = Color.A 2 = K.C (TextureEnvColor) (effectively K.A) 3 = K.A (TextureEnvColor)
13...14	AlphaB	✓	✓	1	This field selects the source value for B. The options are: 0 = Texel.C (effectively T.A) 1 = Texel.A 2 = K.C (TextureEnvColor) (effectively K.A) 3 = K.A (TextureEnvColor)

²⁰ Logic Op register readback is via the main register only

15...16	AlphaI	✓	✓	X	This field selects the source value for I. The options are: 0 = Color.A 1 = K.A (TextureEnvColor) 2 = Texel.C (effectively T.A) 3 = Texel.A
17	Alpha InvertI	✓	✓	X	This bit, if set, will invert the selected I value before it is used.
18...20	Alpha Operation	✓	✓	0	This field defines how the three inputs (A, B and I) are combined. The possible operations are: 0 = PassA (A) 1 = PassB (B) 2 = Add (A + B) 3 = Modulate (A * B) 4 = Lerp (A * (1.0 - I) + B * I) 5 = ModulateABAddI (A * B + I) 6 = ModulateAAddB (A * I + B) 7 = ModulateBAddA (B * I + A)
21	KdEnable	✓	✓	1	When set this bit causes the RGB results of the texture application to be multiplied by the Kd DDA values. It also enables the Kd DDA sto be updated.
22	KsEnable	✓	✓	0	When set this bit causes the RGB results of the texture application (or Kd processing) to be added with the Ks DDA values. It also enables the Ks DDAs to be updated.
23	Motion Comp Enable	✓	✓	X	

11.2.8.4 Direct 3D Texture Application (TBlend)

The D3D texture color ops are as follows: Enable is 1, KsEnable is 0, KdEnable is set if specular highlight is required.

Color fields						
Type	A	B	I	Invl	Operation	
Disable	C.C				PassA	
Copy		T.C			PassB	
CopyAlpha		T.A			PassB	
Add	C.C	T.C			Add	
AddAlpha	C.C	T.A			Add	
Modulate	C.C	T.C			Modulate	
ModulateAlpha	C.C	T.A			Modulate	
BlendFactorAlpha	C.C	T.C	K.A	?	Lerp	
BlendTextureAlpha	C.C	T.C	T.A	?	Lerp	
BlendDiffuseAlpha	C.C	T.C	C.A	?	Lerp	
ModulateColorAddAlpha	C.C	T.C	TA	?	ModulateABAddI	

The D3D texture alpha ops are as follows. Enable is 1:

Color fields					
Type	A	B	I	Invl	Operation
Disable	C.A				PassA
Copy		T.A			PassB
Add	C.A	T.A			Add
Modulate	C.A	T.A			Modulate

So for example, the **TextureApplicationMode** fields for D3D Modulate with Specular highlights would be set as follows (see the **Value** column):

Bits	Name	Read ²¹	Write	Value	Description
0	Enable	✓	✓	1	When set causes the output to be calculated as defined by the fields in this register, otherwise the fragment's data is passed through.
1...2	ColorA	✓	✓	0	This field selects the source value for A. The options are: 0 = Color.C 1 = Color.A 2 = K.C (TextureEnvColor) 3 = K.A (TextureEnvColor)
3...4	ColorB	✓	✓	0	This field selects the source value for B. The options are: 0 = Texel.C 1 = Texel.A 2 = K.C (TextureEnvColor) 3 = K.A (TextureEnvColor)
5...6	ColorI	✓	✓	X	This field selects the source value for I. The options are: 0 = Color.A 1 = K.A (TextureEnvColor) 2 = Texel.C 3 = Texel.A
7	ColorInvertI	✓	✓	X	This bit, if set, will invert the selected I value before it is used.
8...10	Color Operation	✓	✓	3	The possible operations are: 0 = PassA (A) 1 = PassB (B) 2 = Add (A + B) 3 = Modulate (A * B) 4 = Lerp (A * (1.0 - I) + B * I) 5 = ModulateColorAddAlpha (A * B + I) 6 = ModulateAlphaAddColor (A * I + B) 7 = ModulateBIAddA (B * I + A)

²¹ Logic Op register readback is via the main register only

11...12	AlphaA	✓	✓	1	This field selects the source value for A. The options are: 0 = Color.C (effectively Color.A) 1 = Color.A 2 = K.C (TextureEnvColor) (effectively K.A) 3 = K.A (TextureEnvColor)
13...14	AlphaB	✓	✓	1	This field selects the source value for B. The options are: 0 = Texel.C (effectively T.A) 1 = Texel.A 2 = K.C (TextureEnvColor) (effectively K.A) 3 = K.A (TextureEnvColor)
15...16	AlphaI	✓	✓	X	This field selects the source value for I. The options are: 0 = Color.A 1 = K.A (TextureEnvColor) 2 = Texel.C (effectively T.A) 3 = Texel.A
17	Alpha InvertI	✓	✓	X	This bit, if set, will invert the selected I value before it is used.
18...20	Alpha Operation	✓	✓	3	This field defines how the three inputs (A, B and I) are combined. The possible operations are: 0 = PassA (A) 1 = PassB (B) 2 = Add (A + B) 3 = Modulate (A * B) 4 = Lerp (A * (1.0 - I) + B * I) 5 = ModulateABAddI (A * B + I) 6 = ModulateAAddB (A * I + B) 7 = ModulateBAddA (B * I + A)
21	KdEnable	✓	✓	1	When set this bit causes the RGB results of the texture application to be multiplied by the Kd DDA values. It also enables the Kd DDA to be updated.
22	KsEnable	✓	✓	0	When set this bit causes the RGB results of the texture application (or Kd processing) to be added with the Ks DDA values. It also enables the Ks DDAs to be updated.
23	Motion Comp Enable	✓	✓	X	

11.2.9 Implementation

Texture processing has two enables which must both be set to enable modification of the **Color** register. The first enable is loaded via the **TextureApplicationMode** register and is effective until changed by a new **TextureApplicationMode** message. The second enable is the *TextureEnable* bit in the **Render** register and this is only effective until the next **Render** message is received. This second enable is used to temporarily disable texturing when a primitive must not be textured.

11.2.9.1 The Ks and Kd DDAs

The Ks and Kd DDA units interpolate the specular and diffuse RGB values. Sub pixel corrections can be applied to correct for an initial start error on a span.

The output of the DDA units is applied to the texture calculations outlined earlier when the corresponding Apple texture modes are enabled.

- The original Ks and Kd registers (e.g. **KsRStart**) when written to load the corresponding R, G and B registers. This gives some backward compatibility.
- The new KsRStart, dKsRdx and dKsRdyDom registers load up the start, dx and dyDom registers for the Ks Red DDA unit. Similarly for the Ks GB components and also the Kd RGB components. This allows for future set up chips to program these registers directly.

The format is 2's complement 2.22 fixed point format with an effective range clamped to ± 1.999 . There is a small underflow/overflow guard band - if it is exceeded the value wraps around and produces an abrupt color change artefact. (This should not happen if the setup is correct and sub-pixel correction is applied at the start of each span.)

The values of Ks and Kd at each vertex are used to calculate the gradient values in much the same way as the color gradients when Gouraud shading.

The parameters to control the two DDA units are loaded into the red, green and blue values (there is no alpha value) and are held as 1.8 unsigned fixed point numbers so values greater than 1.0 can be represented.

11.2.9.2 Texture Color Registers

The application of texture is qualified by the *TextureEnable* bit in the **Render** command register. The following registers (together with the **TextureApplicationMode** register) control the application of textures.

Register	Data Field	Description
TextureEnvColor	32 bit RGBA format, R in least significant byte	
KsStart	24 bit 2's comp fix pt	Ks start value, loads up the R, G and B DDA start registers.
DKsdx	24 bit 2's comp fix pt	Ks derivative unit X, loads up the R, G and B DDA dx registers.
DKsdyDom	24 bit 2's comp fix pt	Ks derivative unit Y, dominant edge, loads up the R, G and B DDA dyDom registers.
KdStart	24 bit 2's comp fix pt	Kd start value, loads up the R, G and B DDA start registers.
DKddx	24 bit 2's comp fix pt	Kd derivative unit X, loads up the R, G and B DDA dx registers.
DKddyDom	24 bit 2's comp fix pt	Kd derivative unit Y, dominant edge, loads up the R, G and B DDA dyDom registers.
KsRStart	24 bit 2's comp fix pt	Ks Red start value
DKsRdx	24 bit 2's comp fix pt	Ks Red derivative unit X
DKsRdyDom	24 bit 2's comp fix pt	Ks Red derivative unit Y, dominant edge
KsGStart	24 bit 2's comp fix pt	Ks Green start value

dKsGdx	24 bit 2's comp fix pt	Ks Green derivative unit X
dKsGdyDom	24 bit 2's comp fix pt	Ks Green derivative unit Y, dominant edge
KsBStart	24 bit 2's comp fix pt	Ks Blue start value
dKsBdx	24 bit 2's comp fix pt	Ks Blue derivative unit X
dKsBdyDom	24 bit 2's comp fix pt	Ks Blue derivative unit Y, dominant edge
KdRStart	24 bit 2's comp fix pt	Kd Red start value
DKdRdx	24 bit 2's comp fix pt	Kd Red derivative unit X
DKdRdyDom	24 bit 2's comp fix pt	Kd Red derivative unit Y, dominant edge
KdGStart	24 bit 2's comp fix pt	Kd Green start value
DKdGdx	24 bit 2's comp fix pt	Kd Green derivative unit X
DKdGdyDom	24 bit 2's comp fix pt	Kd Green derivative unit Y, dominant edge
KdBStart	24 bit 2's comp fix pt	Kd Blue start value
DKdBdx	24 bit 2's comp fix pt	Kd Blue derivative unit X
DKdBdyDom	24 bit 2's comp fix pt	Kd Blue derivative unit Y, dominant edge

Table 11.15 Texture Color Registers

12

Volume II Index

Alpha Blend, 7-2, 8-5
 Alpha Blending, 7-2
 alpha buffer, 8-5
 Alpha Test, 7-2
 Antialias Application, 7-2, 8-4
 Antialias Example, 8-6
 Antialiasing, 8-5, 8-6
AntialiasMode, 8-6, 8-8
 area stippling, 9-5
 Area Stippling, 8-20, 9-4
AreaStippleMode, 9-4, 9-6, 9-7
AreaStipplePattern, 9-7
AStart, 9-12
 Bitmaps, 8-22
 BitMaskPattern, 8-23, **8-24**, 8-32, 8-36
BorderColor0, 11-12
BStart, 9-12, 9-13, 11-5, 11-31
ChromaLower, 11-18
ChromaUpper, 11-18
 Color DDA, 7-2, 9-9
 Color Formatting, 7-2
 ColorDDAMode, 9-12, 9-13
 ConstantColor, 9-12
Continue, 8-34
ContinueNewDom, 8-7, 8-34
ContinueNewLine, 8-34
ContinueNewSub, 7-8, 8-4, 8-34
Count, 8-36
dAdx, 9-12
dAdyDom, 9-12
dBdx, 9-12, 9-13
dBdyDom, 9-12, 9-13
 DDA, 9-12, 9-13
 delta, 8-2, **8-34**
 Depth, 7-4, 10-4, 10-15
 Depth Example, 10-15
 Depth Gradient, 9-11
 Depth Test, 7-2
 Depth Test, 10-11
 DepthMode, 10-11, 10-13, 10-15
dGdx, 9-12, 9-13, 11-5, 11-31
dGdyDom, 9-12, 9-13, 11-5, 11-31
dKdBdx, 11-31
dKdBdyDom, 11-31
dKddx, 11-30
dKddyDom, 11-30
dKdGdx, 11-31
dKdGdyDom, 11-31
dKdRdx, 11-31
dKdRdyDom, 11-31
dKsBdx, 11-31
dKsBdyDom, 11-31
dKsdx, 11-30
dKsdyDom, 11-30
dKsGdx, 11-31
dKsGdyDom, 11-31
 dKsRdx, 11-30
 dKsRdyDom, 11-30
 Dominant, 7-3
dQdx, 11-2, 11-5
dQdy, 11-2, 11-3, 11-5
dQdyDom, 11-5
dRdx, 9-12, 9-13
dRdyDom, 9-12, 9-13
 dSdx, 11-2, **11-5**
dSdy, 11-2, 11-3, 11-5
dSdyDom, 11-5
 dTdx, 11-2, **11-5**
dTdy, 11-2, 11-3, 11-5
dXDom, 8-36

- dXSub, 8-36
- dY**, 8-11, 8-28, 8-36
- dZdxL, 10-15
- dZdxU, 10-15
- dZdyDomL, 10-15
- dZdyDomU, 10-15
- Examples, 9-7
- flat shaded, 9-12
- Flat Shading example, 9-12
- FlushSpan**, 8-7, 8-34
- Fog, 7-2
- Gouraud shading, 9-13
- Gouraud Shading, 9-11
- Gouraud Shading examples, 9-12
- Graphics Pipeline, 7-1
- Graphics Programming, 7-1
- GStart**, 9-12, 9-13, 11-5, 11-30, 11-31
- Host Out, 7-2
- Image Copy/Upload/Download, 8-28
- Initialization, 7-3
- KdBStart**, 11-31
- KdGStart**, 11-31
- KdRStart**, 11-31
- KdStart**, 11-30
- KsBStart**, 11-31
- KsGStart**, 11-30
- KsRStart, 11-30
- KsStart**, 11-30
- LBDestReadMode**, 10-2
- LBReadFormat**, 10-2
- LBReadMode, 10-3
- LBWriteFormat**, 10-2, 10-4
- Level of Detail calculation, 11-3
- Line Stippling, 9-5
- LineStippleMode**, 9-5, 9-6, 9-7
- LoadLineStippleCounters**, 9-7
- LOD, 11-3, 11-6
- OpenGL Application Modes, 11-22
- patch, 11-10
- Patch, 10-1
- Perspective Correction, 11-3
- Pixel Ownership, 7-2
- Pixel Ownership Test, 10-4
- Pixel Sizes, 8-21
- PixelSize**, 8-21, 8-34
- PointTable**, 8-36
- PointTable0**, 8-36
- QStart, 11-2, 11-5
- Rasterization, 7-8
- Rasterizer, 7-2, 8-1
- Rasterizer Mode, 7-4, 8-32
- Rasterizer Unit Registers, 8-33
- RasterizerMode**, 8-23, 8-24, 8-32, 8-34, 8-36
- Render**, 7-5, 8-31, 8-34
- RGBA and Color-Index(CI) Modes, 9-10
- Router, 7-3
- RouterMode**, 7-3
- RStart**, 9-12, 9-13, 11-5, 11-30, 11-31
- SaveLineStippleCounters**, 9-5, 9-7
- SaveStippleLineCounters**, 9-5
- ScanLineOwnership**, 8-36
- Scissor, 9-1
- Scissor Example, 9-3
- Scissor Test, 7-2
- ScissorMaxXY**, 9-3
- ScissorMinXY**, 9-3
- ScissorMode**, 9-2
- Screen Scissor Tests, 9-1
- ScreenSize**, 9-1, 9-3
- Sides
 - Calculating the Slope, 7-6
 - Span Mask Processing, 8-20
 - Span Operations, 8-18
 - Span Operations and Bitmaps, 8-24
 - Span Operations and Image
 - Copy/Upload/Download, 8-30
 - Span Operations and Stippling, 9-5
 - Span Operations and the Scissor Unit, 9-3
- SStart, 11-2, 11-5
- StartXDom, 7-7, 8-11, 8-12, 8-28, 8-36
- StartXSub, 7-7, 8-11, 8-12, 8-28, 8-36
- StartY, 8-2, 8-11, 8-28, 8-36
- Stencil, 10-4, 10-10
- Stencil Example, 10-10
- Stencil Test, 7-2
- Stencil Test, 10-7
- StencilData**, 10-8, 10-9, 10-10
- StencilMode**, 10-7, 10-8, 10-9

Stipple, 9-3
Stipple Test, 7-2
Sub Pixel Precision and Correction, 8-21
Subordinate, 7-3
Subpixel Correction, 7-4
TexelLUT, 11-17
TexelLUTAddress, 11-17
TexelLUTData, 11-17
TexelLUTIndex, 11-17
TexelLUTTransfer, 11-17
Texture, 7-2, 11-1, 11-2
Texture Filtering, 11-17
texture mapping, 7-3, 11-1
TextureBaseAddr, 11-3
TextureChromaLower, 11-18
TextureChromaUpper, 11-18
TextureColor Generation, 11-21
TextureCoordMode, 8-20, 11-3, **11-4**
TextureEnvColor, 11-30
TextureFilterMode, 11-2, 11-17, 11-18
TextureReadMode, 8-20, 11-3, 11-4, 11-5, 11-8, 11-17
Trapezoids, 8-2
TStart, 11-2, **11-5**
UpdateLineStippleCounters, 9-5, 9-6
User Scissor Test, 9-1
WaitForCompletion, 8-35, 10-2
Window, 10-4, **10-10**, 10-13
WindowOrigin, 9-1, 9-3
Y Limits Clipping, 8-33
ZStartL, 10-15
ZStartU, 10-15