GLINT R5®

Programmer's Guide

DRAFT ONLY

PROPRIETARY AND CONFIDENTIAL INFORMATION

3D*labs*®

GLINT R5®

Programmer's Guide Volume I - Overview

PROPRIETARY AND CONFIDENTIAL INFORMATION

Issue 2

Proprietary Notice

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

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

3D*labs* GmbH
Breckenheimer Weg 29
65205 Wiesbaden
Deutschland
Tel: +49 6122 916 778
Fax: +49 6122 919 646

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

3D*labs* Inc.
480 Potrero Avenue
Sunnyvale, CA 94086,
United States
Tel: +1 (408) 530-4700
Fax: +1 (408) 530-4701

Change History

Document	Issue	Date	Change
172.2.1	1	14/08/2000	Creation
172.2.1	2	05/03/2001	Extensive changes to data input, especially texturing; removed incorrect footnote ref to PEREN004, update logical DMA, corrections to DMA, added note about WCEnable/Byte Swap toggle, removed refs to FeedbackSelectCount,output DMA, cleaned up 3D Pipeline, added hyperlinks to Reference Guides, clarified opaquespan, removed spanmask.

Table of Contents

1 Introduction

This is the Programmer's Reference Manual for the GLINT R5 device. GLINT R5 ("R5") integrates the functionality and features previously found in a rasterizer chip with a compatible geometry and lighting accelerator chip, such as Gamma 1 and GLINT R4.

Programmers used to working with boards containing both kinds of chip will find much that is familiar in R5's overall high-level programming approach, with significant improvements in both functionality and performance. One of these, not surprisingly, is the ease with which lighting and rasterizer functions can be controlled at all levels from API to register-specific bitfields. In fact, the chip can be used as a conventional rasterizer as it is, simply by ignoring the Texture and Lighting capabilities.

A Geometry and Lighting Processor offloads a substantial amount of computation from the host computer into a 3D pipeline. This removes a major performance bottleneck and drastically improves both performance and stability. The 3D pipeline handles transformations, lighting, culling and clipping and sets up the rasterizer by preloading vertex arrays, texture co-ordinates and display lists.

R5 can be programmed to work as a legacy GLINT Delta and separate rasterizer but this wastes much of its power and feature set. R5 has been designed to fully support OpenGL and to faithfully implement all its modes and interactions, so the OpenGL specification and documentation is an excellent guide to what R5 does.

This is not achieved at the expense of other mainstream graphics APIs. Direct3D from Microsoft, QuickDraw3D from Apple, and X-Windows are all catered for. In many respects Direct3D and QuickDraw are subsets of OpenGL so are well covered by OpenGL functionality. However additional functionality has been incorporated into R5 specifically to address the needs of these APIs. Although predominantly oriented to 3D graphics, R5 also supports 2D GUIs.

R5 implements the following parts of the OpenGL Geometry and Lighting Pipeline. This is not an exhaustive list, simply a summary of the main elements:

- OpenGL Begin/End paradigm for describing primitives.
- Texture coordinate generation.
- Normal and texture coordinate transformation.
- Normalization of the normal after transformation.
- Full OpenGL RGB material and lighting for up to 16 light sources.
- Two sided lighting.
- Polygon Offset.
- OpenGL Polygon mode and Color Material operations.
- Backface culling.
- Fog calculations.
- Full frustum clipping with an optional 6 user-defined clipping planes.

- Perspective division and Viewport mapping.
- Render, select and feedback modes.
- Triangle set up (aliased and antialiased).
- Line set up (aliased, antialiased, wide and stippled).
- Point set up (aliased, antialiased and wide).
- Raster Position.

Parts of OpenGL still left to software (excluding system level things such as multiple contexts, etc.):

- Matrix generation, including matrix stack.
- Attribute Push and Pop.
- Display list creation, management and parsing (in the general case).
- Evaluators.
- General state management and Get functions.
- Color index lighting.

In addition to these the following general facilities are available:

- Queued Input DMA.
- Output DMA controller.
- Scatter/Gather operations on DMA.
- Rectangular read and write of host memory.
- Rectangle (2D) set up.
- Hierarchical DMA.
- Blended cursor overlay (Windows 2000) and 8-bit pseudo color (X Windows)

2D/GUI support is limited to:

- Rectangle upload DMA controller.
- Rectangle download DMA controller.
- Rectangle set up.

1.1 Conventions Used In This Manual

Registers or commands accessed via the input FIFO or DMA (i.e. core registers) are in bold. Registers accessed directly from the PCI bus are in italics. Code snippets are in `courier`.

For convenience, registers and other features specified in the *R5 Reference Guide* are frequently hyperlinked to a discussion in the *Programmer's Guide*. The only requirement is that the Reference Guide and Programmer's Guide files be in the same directory.

1.2 Performance

1.2.1 Performance Considerations

Under ideal circumstances R5 would receive instructions and data at its optimum handling rate. The limiting factor for performance is then the processing path taken by the primitives. In order of decreasing speed, the possibilities are:

- The primitive is not in view.
- It is in view, but backface culled.
- It is fully in view, but not backface culled.
- It is partially in view, but not backface culled.

A typical scene will have a mixture of primitives which are whole, clipped or backface culled and which will therefore make differing demands for processor resources and time.

Performance also depends on the number and type of lights and texturing operations, the proportion of reused vertices (fans, meshes), the sequencing and complexity of rendering operations, and, of course, the efficiency of coding (among many other constraints). For this reason performance figures are almost meaningless without suitable benchmarking contexts[1] and an understanding of the programming paradigm and facilities available on this powerful integrated graphics processor.

This manual is primarily intended to help programmers understand the techniques that can be used to programme R5 for optimum performance under a variety of real-world conditions. Ideal performance figures are quoted (when available) in the *R5 Reference Manual*, volume I.

1.2.2 Input Data Format

There are a number of ways of loading R5 registers or issuing commands:

- The host writes a value to the mapped address of the register. This goes via the input FIFO even though the register appears to be written directly.
- The host performs a Block Command Transfer by writing address and data values directly to the FIFO interface registers.
- The host writes address-tag/data pairs into a host memory buffer and uses on-chip DMA to do the transfer.

Regardless of the specific means chosen, the bulk of the data is likely to be vertex definitions.

Each vertex of a primitive can be considered to consist of a three-component Cartesian coordinate and a three-component normal. Each component is a single precision floating point number which takes 4 bytes.

The most compact data format packs the vertex data into 7 words - one tag word and 6 actual vertex data words. In this case the tag is repeated for every vertex so there are 28 bytes per vertex. This format is useful in display lists where the driver has the time to work out the optimum format for vertices between a *glBegin* and *glEnd*.

[1] Benchmark results for R5 board implementations are available on the 3Dlabs websaite at http://www.3dlabs.com/product/card/

Note: *The least compact format, but the one most suitable for 'on the fly' processing (i.e. dispatching the data as soon as it is given to you) is: normal tag, three components, vertex tag, three components i.e. 8 words or 32 bytes per vertex.*

1.3 Further Reading

This manual is not an introduction or tutorial on 3D graphics or OpenGL. The reader is assumed to be familiar with these subjects. Similarly the details of using and programming the GLINT rasterizer family are not covered here. Suitable books or on-line material which cover these topics are:

- *OpenGL Programming Guide*, Jackie Neider et al, Reading MA: Addison-Wesley
- *OpenGL Reference Manual*, Jackie Neider et al, Reading MA: Addison-Wesley
- *The OpenGL Graphics System: A Specification (Version 1.1)*, Mark Segal and Kurt Akeley, SGI
- *Computer Graphics: Principles and Practice*, James D. Foley et al, Reading MA: Addison-Wesley.
- *The GLINT R4 Programmer's Guide*, 3Dlabs Inc., Egham, Surrey, UK.

2

General Programming Notes

2.1 R5 Programming Model

This manual does not try to specify all the interactions between the modes (for example when flat shaded, which vertex provides the color) - to do so would result in a substantial portion of the OpenGL specification being included. R5 follows the OpenGL specification exactly and any deviations are unintentional. 3Dlabs would be very pleased to hear of them so they can be fixed in future products.

Unless otherwise noted all input values are in single precision IEEE floating point format numbers. Denormalized numbers are treated as if they were zero and NaNs are treated as if they are very large numbers.

The remainder of this section describes the programming model for R5. It describes the interface conceptually rather than detailing specific registers and their exact usage. In-depth descriptions of how to program R5 for specific operations may be found in later chapters.

R5 is an integrated graphics processor containing both Geometry Accelerator and Rasterizer. Historically these were two separate chips which were in most respects programmed as one. Physical integration, from the programmer's point of view, simply follows a well-established virtual integration.

2.2 R5 as a Register File or FIFO Interface

The simplest way to view the interface to R5 is as a flat block of memory-mapped registers (*i.e.* a register file). This register file appears as part of Region 0 of the PCI address map for R5. See the *R5 Reference Guide,* Volume I, for details of the address map.

When an R5 host software driver is initialized it can map the register file into its address space. Each register has an associated address tag giving its offset from the base of the register file (since all registers reside on a 64-bit boundary, the tag offset is measured in multiples of 8 bytes). The most straightforward way to load a value into a register is to write the data to its mapped address.

In reality the chip interface comprises a 32 entry deep FIFO. Each write to a register causes the written value and the register's address tag to be written as a new entry in the FIFO.

Programming R5 to draw a primitive consists of writing initial values to the appropriate registers followed by a write to a command register. The last write triggers the start of rendering.

R5 has several hundred registers. All registers are 32 bits wide and should be 32-bit addressed. Many registers are split into bit fields.

Note: *bit 0 is the least significant bit.*

Graphics registers are shown in bold font (for example: **GeometryMode**). In addition there are registers related to initialization and I/O which are documented in the *R5 Reference Guide,* Volume II..

Note: In future chip revisions the register file may be extended and currently unused bits in certain registers may be assigned new meanings. Software developers should ensure that only defined registers are written to and that undefined bits in registers are always written as zeros.

2.3 Register Types

R5 has three main types of register:

- Control and Data Registers
- Command Registers
- Internal Registers

2.3.1 Control and Data Registers

Control and Data Registers are updated only by the host - the chip effectively uses them as read-only registers. Examples of control registers are GeometryMode and LightingMode. Examples of data registers are FrontDiffuseColorGreen and ViewPortOffsetX.

Once initialized by the host the chip only reads these registers. They can be read back at any time, however writing to a register and then immediately reading it is not guaranteed to return the value written. Writes are buffered in a FIFO so it may take several cycles before the register itself is updated. The only way to be sure that a register is not about to be updated by state held in the FIFO or internal pipeline is to synchronize using the Sync command.

To read back a particular register read the address you would use to write to the register. See the *R5 Reference Manual*, volume III, for readback capability of specific registers.

All registers are loaded in the order they are given in and all commands are executed in order as well. A register can be updated at any time and it is guaranteed *never* to corrupt or affect any command which may be in operation and which depends on the previous register contents for its correct operation. Updating a register only affects subsequent commands.

2.3.2 Command Registers

Command Registers typically cause the chip to start rendering (or some other type of internal processing) when they are written to. Normally, the host initializes the appropriate control and data registers and then writes to a command register to start drawing. Command registers cannot be read back.

Note: For convenience in this document we often refer to "sending an XXX command to R5" rather than saying "the XXX Command register is written to, which initiates drawing (or some other action)".

R5 uses the fact that a vertex has been written to initiate an appropriate action. This mechanism is used to reduce the amount of commands sent by the host to render a primitive - important when millions of primitives per second are being processed.

R5 also supports direct access to the rasterizer to draw primitives using the DrawLine and DrawTriangle commands. This is a cumbersome way to program the chip and is retained for legacy purposes only.

This manual does not cover rasterizer registers (shown in the *R5 Reference Guide* volume III without shaded headers), which can be reviewed elsewhere[2]. Where R4 and R5 differ, for example hierarchical display lists and improvements in the Read Monitor unit, the changes are discussed below.

2.3.3 Internal Registers

Internal Registers are not accessible to host software. They are used internally by the chip to hold intermediate values and share information between adjoining primitives (e.g. in a mesh or polyline).

For the most part internal registers can be ignored, however they do need to be saved and restored when context-switching during a primitive group (demarcated by a **Begin** and **End** pair of commands). This is covered in a later chapter.

2.4 Register behaviour

2.4.1 Updating Mode Registers (and/or)

Many of the mode registers have **And** and **Or** variants. For example the **GeometryMode** register can also be written to using **GeometryModeAnd** or **GeometryModeOr**.

The And and Or variants combine the new data with the existing register contents using a bitwise AND or a bitwise OR operation respectively. This allows single bit fields to be set or cleared in a single write or multi-bit fields to be set to any value by first clearing the bits in question with the And variant and then using the Or variant to merge in the new value.

This facility makes it unnecessary to keep a software copy so that the fields we wish to retain are not overwritten. The real benefit is that updates to these mode registers can be held in native format display lists (i.e. display lists which don't need to be parsed by software but can be read directly by R5). Native display lists are obviously faster than input display lists which must be parsed.

2.4.2 Initialization

Very few registers are initialized by power-on reset or by the software reset. It is advisable to set all registers into a well defined state before using R5.

2.4.3 Efficiency

Software developers wishing to write device drivers for R5 should become familiar with the different types of registers. Some registers such as the **Vy** and **Vz** registers have to be updated for almost every primitive whereas other control registers such as **ViewPortScaleX** or **TransformMode** can be updated much less frequently. Pre-loading the appropriate control registers can reduce the amount of data that has to be loaded into the chip for a given primitive thus improving efficiency.

The entire register set for R5 is specified in the *R5 Reference Guide* volumes II and III. Graphics core registers are shown in volume III.

---

[2] For a detailed discussion of rasterizer registers and programming see the *R4 Reference Guide* and *R4 Programmer's Guide.*, but note chip-specific differences in Clock programming etc.

Most of the registers used for programming are shown with shaded headers, e.g. **BoundingVertexX**. Those without shading, such as **DrawLine01**, are rasterizer-specific and retained for compatibility with earlier GLINT products.

3 Loading Data into R5

The facilities for getting data into R5 are a superset of those available in earlier members of the GLINT family. Some of these are aimed at increasing system performance while others introduce new functionality.

The memory-mapped registers and input FIFO can still be used to pass commands and data to R5 but this limits the functionality available and requires significant management overhead to report the condition of the FIFO and control the order in which commands are executed.

A general design philosophy is that much of the new functionality is driven via the DMA buffer rather than dedicated PCI registers or FIFO writes. This brings many of the benefits normally associated with core commands such as queuing and asynchronous operation. R5 also allows data to be developed by the host and loaded outside the command stream, particularly using vertex data and index buffers and virtual texture cacheing.

All memory reads and writes started by R5 go via the PCI/AGP interface to reach the host's memory. The PCI Interface also implements a layer of byte swapping (enabled by **ControlDMAControl** register bit 0) over and above any byte swapping described below.

There are three basic ways of loading R5 registers or issuing commands:

- The host writes a value to the mapped address of the register. This is given the register tag value and forwarded to the input FIFO for processing.
- The host performs a Block Command Transfer by writing the tag address and data values directly to the FIFO interface registers.
- The host writes address-tag/data pairs into a host memory buffer and uses the on-chip DMA to do the transfer.

3.1 Command FIFO

The graphics processor FIFO access area is a direct route to the GP Input and Output FIFOs. Writing to any address in this region causes the data to be sent to the Input FIFO; reading from any address accesses data from the Output FIFO. When writing to the raw FIFO address an address tag description must first be written *followed by* the associated data. In fact, the format of the tag descriptions and their data is identical to that described below for DMA buffers[3]. The DMA mechanism can be thought of as an automatic way of writing to the raw input FIFO address.

Note: *When writing to the raw FIFO address the FIFO full condition must still be checked by reading the **InFIFOSpace** register. However, writing tag descriptions does not put tag entries into the FIFO – it simply establishes a set of tags to be paired with the subsequent data.*

[3] For example, instead of using the R5 DMA it would be possible to transfer data by constructing a DMA-style buffer of data and then copying each item in this buffer to the raw input FIFO address. Based on the tag descriptions and data written, R5 constructs tag/data pairs to enter as real FIFO entries.

Because direct writes to the FIFO do not place tag values in the FIFO itself, the FIFO space can be used for data only - free space need be ensured only for actual data items that are written (not the tag values). For example, where each tag is followed by a single data item it would be possible to do 64 writes to an empty buffer before checking for free space.

See the *R5 Reference Guide* for more details of the Graphics Processor FIFO Interface address range.

3.1.1 PCI Disconnect

Setting bit 0 of the **FIFODiscon** register to 1 enables FIFO disconnection. It does not require host polling, but forces host write retries until the data is accepted. This may affect other time-critical peripherals on the PCI bus, e.g. sound cards, and it sometimes drops interrupts. Despite the speed advantage it is therefore advisable to use this facility judiciously and for short transfers.

3.1.2 Polling InFIFOSpace

The input FIFO is 32 entries deep and each entry consists of a tag/data pair. The **InFIFOSpace** register can be read to determine how many entries are free. The value returned by this register will never be greater than 64.

An example of loading registers using the FIFO is given below. The pseudocode fills a series of rectangles. Details of the conventions used in the pseudocode examples may be found in Appendix B.

Assume that the data to draw a single rectangle consists of 8 words (including the **Render** command).

Note: Some data values are in 16.16 fixed point format.

```
for (i = 0; i < nrects; ++i) {
while (*InFIFOSpace < 8);
    // wait for room
StartXDom(rect->x1 << 16);
StartXSub(rect->x2 << 16);
dXDom(0x0);
dXSub(0x0);
Count(rect->y2 - rect->y1);
YStart(rect->y1 << 16);
dY(1 << 16);
Render(GLINT R5_TRAPEZOID_PRIMITIVE);
       }
```

Checking the status of the FIFO before each write is inefficient so it is checked before loading the data for each complete rectangle. Since the FIFO is 64 entries deep, a further optimization is to wait for all 64 entries to be free after every second rectangle. Further optimization is possible by moving **dXDom**, **dXSub** and **dY** outside the loop (as they are constant for each rectangle) and doing the FIFO wait after every third rectangle.

The **InFIFOSpace** FIFO control register contains a count of the number of entries currently free in the FIFO. The chip increments this register for each entry it removes from the FIFO and decrements it every time the host puts an entry into the FIFO.

The graphics core registers cannot be read through the core FIFO interface. Command buffers generated to be sent to the FIFO interface can be read directly using the DMA

controller. We discuss the difference between graphics core DMA and Control space DMA registers below.

The input FIFO can be written to at any time, even when DMA is in progress. This differs from earlier GLINT chips where the input FIFO was shared between the DMA controller and regular host use.

3.2 DMA

Loading individual registers directly via the FIFO is often an inefficient way to load data into R5. Given that the FIFO can accommodate only a small number of entries the FIFO has to be interrogated frequently to determine how much space is left. Also, if an API function requires a large amount of data then the function cannot return until almost all the data has been consumed. This may take some time depending on the types of primitives being drawn.

R5 supports DMA to improve data transfer efficiency at key points in the architecture. The DMA types include:

- Input DMA - various flavours discussed below
- Output DMA - writes directly from the 8 word deep output FIFO to the host or PCI Bus devices.
- Bypass DMA - used to access memory, memory control registers, video unit and VGA.
- Rectangular DMA - The rectangle DMA unit loads bit map data from system memory. It can request a rectangular array of packed data and separate out the individual pixels. Supported pixel sizes are 8, 16, 24 and 32 bits.
- Texture DMA - Dedicated DMA engine to manage texture downloads, particularly with host-resident virtual textures.

3.2.1 DMA Security

R5 supports a security mode to avoid accidental writes during DMAs to registers which can hang the entire graphics pipeline. When the **Security** register *secure* bit is set the following tags are filtered out of DMA command buffers:

- FilterMode
- VTGAddress
- VTGData
- Security
- DMARectangleWrite
- DMAOutputCount
- DMAFeedback
- ContextDump
- ContextRestore
- ContextData

In Secure Mode these registers can only be accessed through the memory mapped control registers.

3.2.2 DMA Tag Formats

When a secondary DMA is carried out each 32-bit tag description in the DMA buffer conforms to the following format:

- A packet is made up of a header followed by some number of data items. The format of the header is:

Bits	Field	Description
0-3	Offset	Index into 16 tags in each group
4-13	Group	16 tags usually grouped by association
14-15	Mode	Type of packet, 0 = hold tag 1 = increment tag 2 = indexed tag 3 = reserved
16-31	Count or mask	Meaning changes with type of packet

Table 3.1 DMA Tag Description Format

- There are 3 different tag addressing modes for DMA: hold, increment and indexed. The different DMA modes are provided to reduce the amount of data which needs to be transferred, hence making better use of the available DMA bandwidth.

  Hold: Tag formed from group and offset, kept constant for the next count + 1 data items.

  Increment: Tag formed from group and offset, incremented by one for each of the next count + 1 data items.

  Index: Tag formed from group (offset ignored) and mask showing which of the 16 tags in the group are valid. Tags formed in incrementing order and paired with the data.

3.2.2.1 DMA Tags - Hold Format

In this format the 32-bit tag description contains a tag value and a count specifying the number of data words following in the buffer. The DMA controller writes each of the data words to the same address tag. This is useful e.g. for image download where pixel data is continuously written to the FrameBuffer. The bottom 11 bits specify the register to which the data should be written; the high-order 16 bits specify the number of data words (minus 1) which follow in the buffer and which should be written to the address tag

Note: The 2-bit mode *field for this format is zero so a given tag value can simply be loaded into the low order 16 bits.*

A special case of this format is where the top 16 bits are zero indicating that a single data value follows the tag (i.e. the 32-bit tag description is simply the address tag value itself). This allows simple DMA buffers to be constructed which consist of tag/data pairs. For example to render a horizontal span 10 pixels long starting from (2,5) the DMA buffer could look like this:

StartXDom
2 << 16
StartY

5 << 16
StartXSub
12 << 16
Count
1
Render
(trapezoid render command)

3.2.2.2 DMA Tags - Increment Format

address-tag with Count=n-1, Mode=1
value 1
...
value n

This format is similar to the Hold format except that as each data value is loaded the address tag is incremented (the value in the DMA buffer is not changed; R5 updates an internal copy). Thus, this mode allows contiguous R5 registers to be loaded by specifying a single 32-bit tag value followed by a data word for each register.

The low-order 11 bits specify the address tag of the first register to be loaded. The 2 bit mode field is set to 1 and the high-order 16 bits are set to the count (minus 1) of the number of registers to update. To enable use of this format the register file has been organized so that registers which are frequently loaded together have adjacent address tags. For example, the 32 **AreaStipplePattern** registers can be loaded as follows:

AreaStipplePattern 0, Count=31, Mode=1
row 0 bits
row 1 bits
...
row 31 bits

3.2.2.3 DMA Tags - Indexed Format

R5 address tags are 11 bit values. For the purposes of the Indexed DMA Format they are organized into major groups and within each group there are up to 16 tags. The low-order 4 bits of a tag give its offset within the group. The high-order bits give the major group number. The *R5 Reference Guide* volume III, chapter 6 lists the individual registers with their Major Group and Offset.

This format allows up to 16 registers within a group to be loaded while still only specifying a single address tag description word.

address tag with Mask, Mode=2
value 1
...
value n

If the Mode of the address tag description word is set to Indexed Mode then the high-order bits are used as a mask to indicate which registers within the group are to be used. The bottom 4 bits of the address tag description word are unused.

The group is specified by bits 4 to 8. Each bit in the mask is used to represent a unique tag within the group. If a bit is set then the corresponding register will be loaded. The number of bits set in the mask determines the number of data words that should be following the tag description word in the DMA buffer. The data is stored in order of increasing corresponding address tag. For example,

0x003280F0
value 1
value 2
value 3

The Mode bits are set to 2 so this is indexed mode. The Mask field (0x0032) has 3 bits set so there are three data words following the tag description word. Bits 1, 4 and 5 are set so the tag offsets are 1, 4 and 5. The major group is given by the bits 4-8 which are 0x0F (in indexed mode bits 0-3 are ignored). Thus the actual registers to update have address tags 0x0F1, 0x0F4 and 0x0F5 corresponding to registers dRdx, dGdx and dRdyDom. These are updated with value 1, value 2 and value 3 respectively.

3.2.2.4 DMA Example

The following pseudo-code example shows how to draw a series of rectangles using the DMA controller. This example uses a single DMA buffer and the simplest case which is Hold Mode for the tag description words in the buffer.

```
UINT32         *pbuf;
DMAAddress(physical address of dma_buffer)
while (*DMACount != 0)
;            // wait for DMA to complete
pbuf = dma_buffer;

*pbuf++ = GLINT R5TagdXDom;
*pbuf++ = 0;
*pbuf++ = GLINT R5TagdXSub;
*pbuf++ = 0;
*pbuf++ = GLINT R4TagdY;
*pbuf++ = 1 << 16;
for (i = 0; i < nrects; ++i) {
 *pbuf++ = GLINT R5TagStartXDom;
 *pbuf++ = rect->x1 << 16; // Start dominant edge
 *pbuf++ = GLINT R5TagStartXSub
 *pbuf++ = rect->x2 << 16; // Start of subordinate
 *pbuf++ = GLINT R5TagCount;
 *pbuf++ = rect->y2 - rect->y1;
 *pbuf++ = GLINT R5TagYStart;
 *pbuf++ = rect->y1 << 16;
 *pbuf++ = GLINT R5TagRender;
```

```
 *pbuf++ = GLINT R5_TRAPEZOID_PRIMITIVE;
}
// initiate DMA
DMACount((int)(pbuf - dma_buffer))
```

The example assumes that a host buffer has been previously allocated and is pointed at by "dma_buffer".

3.2.3 DMA Buffer Addresses

R5 DMA buffer addresses can be accessed either at physical locations or via logical-to-physical lookup table. Logical addressing allows DMA buffers to be in non-contiguous physical memory.

Logical addressing is discussed below (*DMA Address Mapping*).

3.2.4 DMA Interrupts

R5 provides interrupt support as an alternative means of determining when a DMA transfer is complete. The particular interrupt schemes depend on the type of DMA and are discussed in chapters 3 and 4 below.

In practice, however, these are too time-consuming to be used extensively. Normally the host develops data buffers and uses a compact DMA transfer method which eliminates unnecessary tags, duplicate vertices, unused parameters etc.

There are several forms of DMA, or DMA machines, appropriate for specific tasks (for example indexed DMA for vertex loading, hierarchical DMA, Bypass DMA for memory and video control, Rectangle DMA, Texture DMA).

3.2.5 Input DMA

R5 supports several types of DMA which can help to boost performance under differing conditions:

- Control ("legacy", "Single" or "Primary") DMA: the host writes DMA tag and value data directly to the PCI DMA registers[4]. On completion of the transfer, usually signalled by an interrupt, a new *Count* value can be loaded to initiate a new transfer. Control DMA does not support Hierarchical or Queued[5] DMA.
- Queued DMA: the host writes appropriate values to the DMA Address and DMA Count registers[6] in the graphics core. The GP FIFO is monitored for DMA tags, and when found they are used to reference a host DMA buffer which is sent to a DMA Controller for execution. The DMA Controller can be thought of as a command shell - it manages its own FIFO buffer and processes the data transfer as instructed. The PCI DMA Controller is distinct from the GP FIFO and manages its own buffer to prevent overflow. When the GPInFIFO is in queued DMA mode, DMA requests can be submitted up to the size of the GPInFIFO (max. 128 words depending on what other data is present).

---

[4] **ControlDMAAddress** and **ControlDMACount**

[5] The **DMAMode** command in Gamma 1 is not supported. DMA transfers can be queued by adding them sequentially to the GPInFIFO command stream using **DMAAddr** and **DMACount**, or **DMAContinue**.

[6] **DMAAddr, DMACount**: these must not be part of an indexed or incrementing tag

- Continue DMA: This is not in fact a different kind of DMA but an extension to CommandDMA capabilities generally. It allows extended DMA transfers by writing additional data to the DMA buffer and using **DMAContinue** to add to the *Count* value for the buffer most recently loaded. This avoids most of the overhead needed to initiate a new transfer. Continues can be used with any other DMA type including Control DMA.
- Hierarchical DMA: As with Queued DMA, the host writes to the DMA Address and Count registers. When the tags are encountered in the GPInFIFO, the buffer information is passed to a DMA Controller for execution. But if the DMA Controller finds new Address and Count tags in the buffer while its count is non-zero, it spawns another DMA Controller to manage the new transfer, similar to a nested command shell. The new DMA unrolls, then the previous transfer resumes. This is particularly useful for OpenGL display lists. Only two levels of nesting are supported.
- Conditional DMA: As for Queued DMA, but instead of providing a *DMACount* value to initiate the transfer (by writing it to the DMA Controller's FIFO) we use the **DMABoundingBoxJump** register to provide a *Count* value and test the current DMA Address buffer. If the Bounding Box test fails, then a DMA transfer defined by **DMAFailAddr** and **DMAFailCount** is executed. An improper Bounding Box test sets an error in **CoreErrorFlags1**. This flavour of DMA is most useful in improving the performance of MIPmaps by discarding non-displayed geometry at an early stage in processing. The Fail Address allows buffer data that should be implemented regardless of the bounding box test (e.g. setting the Current Color) to be carried out unconditionally.

These are all described in greater detail on the following pages.

Control DMA and Command DMA take different routes through the GPInDMA Controller.. Their relationship is shown in the diagram below.

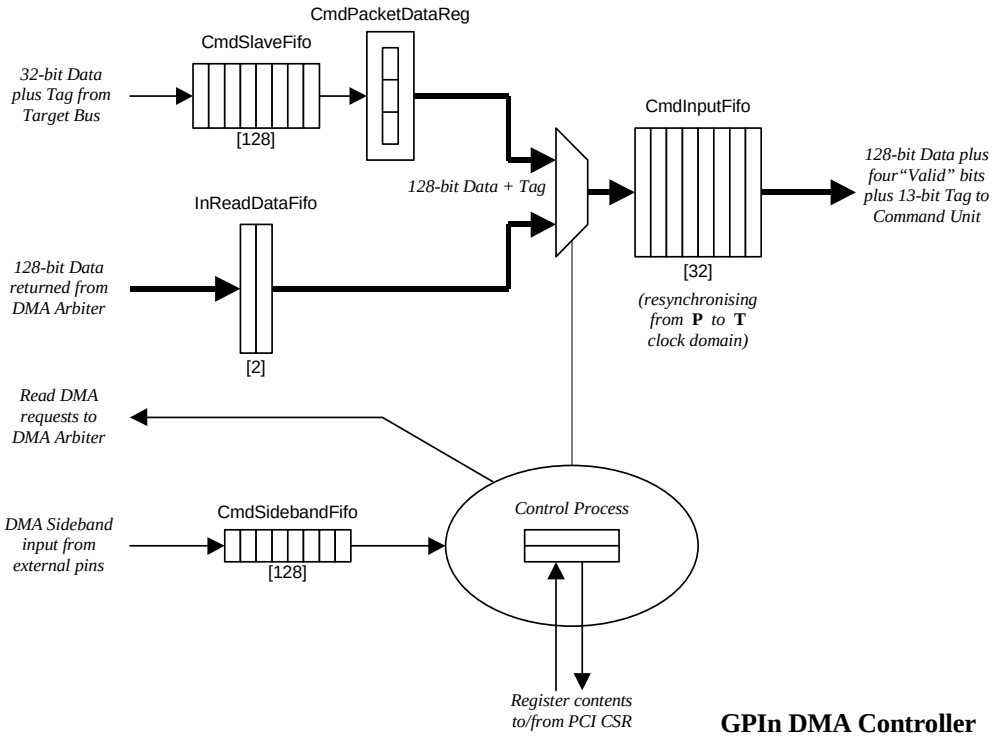


Figure 3.5 - GPIn DMA Controller

3.2.5.1 Control DMA

Also called Legacy, Primary or Single DMA, the address written to the PCI **ControlDMAAddress** is decoded and converted directly into a tag by inspecting its lower bits. The tag formed from the address is used to determine which register the data should be written to. The **ControlDMACount** tag initiates the DMA transfer and is decremented as the transfer proceeds.

The end of each DMA transfer can be signaled using the *ControlDMA* interrupt available in **IntEnable**. In some cases a **CommandInterrupt** command may be useful to raise an interrupt at the end of a group of transfers or at some other defined point within the GPInFIFO or DMA buffer. **CommandInterrupt** is convenient because it only sets an interrupt flag when the previous transfers, including nested transfers, have completed.

Host software must generate the correct DMA buffer address for R5's DMA controller. R5 includes address translation hardware so the address may be a logical address, in which case it is translated to the correct physical address via a page mapping table. Alternatively if logical addressing isn't being used the address passed to R5 must be the physical address of the DMA buffer in host memory and the buffer must also reside at contiguous physical addresses as accessed by R5. This is expanded in section 3.2.5.6.

Command DMA can be controlled by commands loaded through Control DMA. Further, other Command DMAs may be controlled through Command DMAs to a nesting level of 2.

3.2.5.2 DMA Interrupts

R5 provides DMA interrupt support as (among other things) a way of determining when a DMA transfer is complete. If enabled, the interrupt is generated whenever the last data item is transferred from the DMA buffer.

The Interrupts and other flag and error conditions described in the **IntFlags**, **IntEnable** and **ErrorFlagsl** PCI registers apply to both Control and Command DMAs. For example, a **CommandInterrupt** can be loaded as command register or as part of a DMA buffer.

To enable interrupts the appropriate bit must be set in the **IntEnable** register. The interrupt handler should check the *ControlDMA* Flag bit in the **IntFlags** register to determine that a DMA interrupt has actually occurred. To clear the interrupt a word should be written to the **IntFlags** register with the *DMAFlag* bit set to one[7].

Control DMA sets a **Commandinterrupt** whenever the current DMA transfer is finished. Command DMA does not automatically set an interrupt - it must be seeded in the DMA buffer of FIFO using **CommandInterrupt**.

Using interrupts frees the processor for other work while the DMA is being completed. Since the overhead of handling an interrupt is often quite high for the host processor the scheme should be used with care, and probably tuned to the target host and operating system to allow a period of polling before sleeping on the interrupt.

A typical use of DMA interrupts might be as follows:

```
prepare DMA buffer
DMACount(n);   // start a DMA transfer
prepare next DMA buffer
while (*DMACount != 0) {
1111111            mask interrupts
set DMA Interrupt Enable bit in IntEnable register
sleep on interrupt handler wake up
unmask interrupts
}
DMACount(n) // start the next DMA sequence
```

The interrupt handler could then be:

```
if (*IntFlags & DMA Flag bit) {
reset DMA Flag bit in IntFlags
send wake up to main task
}
```

Interrupts are complicated and depend on the facilities provided by the host operating system. The above pseudocode only hints at the system details.

[7] This is different from Gamma 1, where **CommandIntFlags** and **CommandErrorFlags** were used specifically to clear the **IntFlags** and **ErrorFlags** registers.

3.2.5.3 Command DMA

Where Control DMA uses the PCI **ControlDMAAddress** and **ControlDMACount** registers to initiate the transfer, Command DMA uses the GPInFIFO registers **DMAAddress** and **DMACount** to place the DMA request in the GP FIFO command stream (see Figure 3-5 above).

At chip reset the *BusMasterEnable* bit in the **CFGCommand** register must be set to allow DMA to operate (see the *R5 Reference Guide* for further details). Then the host software has to prepare a host buffer containing register address tag descriptions and data values and/or packets constructed as discussed above.

An optimization is to use a double buffered mechanism with two DMA buffers. This allows the second buffer to be filled before waiting for the previous DMA to complete thus further improving the parallelism between host and R5 processing.

```
do any pre-work
get free DMA buffer and mark as in use
put render data into this new buffer
DMAAddress(address of new buffer)
while (*DMACount != 0)
;           // wait for DMA to complete
           // using a back off algorithm
DMACount(number of words in new buffer)
mark the old buffer as free
return
```

Double Buffering is discussed in greater detail in Chapter 4, section 4.

Command DMA can be controlled by commands loaded through the primary DMA. The **DMACurrentAddress** register returns the position in a DMA that has been reached. It guarentees that the memory address it points to has been read by R5 and can be modified by the CPU. There is one register for each command unit. Other Command DMAs can be controlled through Command DMA to a nesting level of 2.

3.2.5.4 Queued DMA

The host writes the base address of this buffer to the **DMAAddress** register and the count of the number of words to transfer to the **DMACount** register. Writing to the **DMACount** register starts the DMA transfer and the host can now perform other work. There is enough room in the GP input FIFO to queue up 64 pending DMA operations, assuming that each is in the address+tag format which requires 2 dwords.

Any normal PCI writes to the **ControlDMAAddress** and **ControlDMACount** PCI registers while in this mode are simply placed in queue pending completion of the current transfers.

Two DMA buffers can be maintained at the same time. If a new address is sent, it can be read from the FIFO before the current DMA completes.

In general, if the complete set of rendering commands required by a given call to a driver function can be loaded into a single DMA buffer, then the driver function can return immediately. Meanwhile, in parallel, R5 is reading data from the host buffer and loading it into its FIFO.

Some algorithms require that data be loaded multiple times (e.g. drawing the same object across multiple clipping rectangles). Since the DMA only reads the buffer data it can be

downloaded many times simply by restarting the DMA. This can be very beneficial if composing the buffer data is a time consuming task.

An additional mechanism, the **CommandID** register, can be used to mark any point in the command stream so that the use of index and vertex buffers can be monitored. This is a register that is loaded with an ID field; and like the DMA address register it can be read at any time.

3.2.5.5 Continued DMA

The **DMAContinue** command is used as a lightweight DMA that concatenates with the DMA currently in progress. It is, in fact, simply an extension of Command DMA capabilities generally. It holds a count as the data field that is used to extend the current DMA, so small DMAs may be combined into long bursts for better bus efficiency.

To avoid the overhead of polling the input FIFO space every time a command is sent, the continue is absorbed immediately and added to the current count, so taking no space in the input FIFO.

Two separate address/count pairs are maintained to enable double buffering of commands so that a new address may be loaded and Continues applied to it while the current DMA is executing, and still take no space in the FIFO.

The CPU operates much more efficiently when it uses write-combining. If something is mapped in as write combined the CPU can empty its write-combine buffer in any order. R5 supports a 64 byte buffer (16 dwords) so the lower 4 bits of the tag represent the position within the write combine buffer. R5 uses them to put the data back to the correct order. When used in this mode, the CPU writes data to incrementing addresses, but the data may be sent to R5 jumbled up.

*Note: Configuring R5 for Write Combining (bit 14, WC Enable in **ChipConfig**) automatically disables byte swapping.*

The normal mode of operation is that a start address for the DMA buffer is loaded, then **DMAContinue** commands are sent until the buffer has been completed, when a new address is sent. Using packets to load the **DMAContinue** command is inefficient because it takes two words (header plus continue). To make this more efficient, any data written to the address space marked as write-combine capable is interpreted as a **DMAContinue** command, and does not need a header.

Actually write-combining the FIFO is a function of the CPU. If it is not enabled then the one word continue command still operates. If write-combining is enabled all writes must be to consecutive incrementing addresses and the first combined write following a non-combined write must be to an address on a 32 byte boundary.

Note: This behaviour is not the same as in previous GLINT rasterizers .

3.2.5.6 DMA Address Mapping

Logical to physical DMA buffer mapping allows the programmer to lock down pages of DMA buffers[8] at non-contiguous addresses in host memory. This avoids a common restriction of virtual host memory addressing which requires contiguous physical addresses.

This simplifies DMA buffer allocation, especially if the allocation is done when the system has been up and running for some time and very few pages are still physically contiguous.

[8] To avoid having them swap out and become unavailable

Logical Addressing mode for Command DMA is set in **AddressTranslationControl** (PCI offset 0x3408) by selecting the DMA type (Bypass, InFIFO, Rasterizer, GPFIFO etc.) then enabling them for Logical Addressing by setting the *Enable* field. For Control DMA, enable *InputFIFOEnable* in **ArbiterRequestControl** and *GPEnable* in A**ddressTranslation Control.**

When running in Logical Addressing mode reads and writes generated by the selected DMA controller(s) are translated into physical addresses before the memory read or write is issued on the primary PCI bus. The address translation, or more accurately the page translation from the logical page number to the physical page number, is done by indexing into a page table held in the host's memory. This then returns the corresponding physical page address which is merged with low order address bits of the logical address to give the physical address.

A typical setup routine is shown below:

Note: Code examples are taken out of context and are not intended to be workable as shown. All code examples are for purposes of illustration only.

```
// Resgister defs
// R5's internal Gamma registers
PADRANGE(12a, 0x3400-0x3178);
GLINT_REG  G3R5ArbiterRequestControl;       // 0x3400
GLINT_REG  G3R5AddressTranslateControl;     // 0x3408 --
       Switches on logical translation and 32/64 bit addressing
GLINT_REG  G3R5Unused1;             // 0x3410
GLINT_REG  G3R5PhysicalAddressOffset;       // 0x3418 - Bits
       32-63 of address when in 64 bit mode with translation
       reset (=0)
GLINT_REG  G3R5PageTableBaseAddrLower;      // 0x3420 -- Base
       address of page table in system memory
GLINT_REG  G3R5PageTableBaseAddrUpper;      // 0x3428
GLINT_REG  G3R5MinLogicalPage;              // 0x3430 --
       Minimum and max logical addresses
GLINT_REG  G3R5MaxLogicalPage;      // 0x3438
GLINT_REG  G3R5Flush;               // 0x3440
GLINT_REG  G3R5InvalidateTLB;               // 0x3448
GLINT_REG  G3R5Restart;             // 0x3450
GLINT_REG  G3R5Unused2;             // 0x3458
GLINT_REG  G3R5PageFaultSource;     // 0x3460
GLINT_REG  G3R5ArbiterStatus;               // 0x3468
GLINT_REG  G3R5ArbiterReadIndex;    // 0x3470
GLINT_REG  G3R5ArbiterReadData;     // 0x3478
GLINT_REG  G3R5PipeSelect;                  // 0x3480


// Sets the address and length of the translation table...


void ProgramGammaPageTableAddress(PHW_DEVICE_EXTENSION
       hwDeviceExtension)
{
       GAMMA_LOCKMEM_MASTER_REC    Master = *hwDeviceExtension-
       >GammaLockmemMaster;
```

```
            GLINT_DECL;

            if (IS_GLINT_R5_ID(hwDeviceExtension-
            >deviceInfo.DeviceId))
            {
                  // Set up page table address

                  VideoPortWriteRegisterUlong(G3R5_PAGE_TABLE_ADDR_L
            OWER, Master.PhysTableAddress.LowPart);

                  VideoPortWriteRegisterUlong(G3R5_PAGE_TABLE_ADDR_U
            PPER, Master.PhysTableAddress.HighPart);

                  // Set up page table address and size (Note: this
            assumes 32-bit operation).
                  VideoPortWriteRegisterUlong(G3R5_MIN_LOGICAL_ADDR,
            0);
                  VideoPortWriteRegisterUlong(G3R5_MAX_LOGICAL_ADDR,
            ((Master.LogicalTableSize/4) * GAMMA_PAGE_SIZE));
            }
            else  //   G1 and G2
            {
                GLINT_SELECT_SETUP_CHIP;
                  VideoPortWriteRegisterUlong(GAMMA_PAGE_TABLE_ADDR,
            Master.PhysTableAddress.LowPart);

                  VideoPortWriteRegisterUlong(GAMMA_PAGE_TABLE_LENGT
            H, Master.LogicalTableSize / GAMMA_PAGE_SIZE);
            }
}
```

The **ArbiterRequestControl** register enables DMA requests to be issued to the PCI and AGP bus masters. Address translation is enabled by the **AddressTranslationControl** register and the individual enable bits for each source of request as described above.

The page tables used for translation are in system memory. Their base address is set by the **PageTableBaseAddressLower** and **PageTableBaseAddressUpper** registers.

The format of an entry in the page table is:[9]

Bit	Name	Description
0	Resident	0 = page not resident; 1 = page resident
1	Read	0 = error to read this page; 1 = read access permitted
2	Write	0 = error to write this page; 1 = write access permitted
3	Valid	0 = page invalid, error condition; 1 = page valid

[9] The page table entries are always held in little-endian format in system memory. On a big-endian machine it is the responsibility of the driver software to ensure the page tables are in the correct format for the DMA Arbiter, as no mechanism is provided for byte-swapping table entries as they are read.

4-6	PageSize	0 = 4K byte page size 1 = 8K byte page size 2 = 16K byte page size 3 = 32K byte page size 4 = 64K byte page size 5 = 128K byte page size 6 = 256K byte page size 7 = 512K byte page size
7-11	Reserved	
12-63	Address	Start address of page in physical memory.

Table 3-3 Translation Page Table Entry Format

The standard table entry allows for a 64 bit byte aligned physical address. To achieve this each entry takes 64 bits and is packed on 64 bit boundaries. If the *TableFormat* field in the **AddressTranslationControl** register is set to 32 bits then each entry is 32 bits wide and packed on 32 bit boundaries and the missing address bits (32-63) from the table entry are assumed to be zero.

- ***PhysicalAddressFormat***
  If the *PhysicalAddressFormat* field in **AddressTranslationControl** is set to 32 bits then the system is not capable of 64 bit addressing - the upper 32 bits of the physical address are ignored. The format of the table and the size of the physical address are orthogonal.
- ***Valid***
  The *Valid* bit is used to mark pages as valid or not valid for use; accessing an invalid page is an error and causes a page fault interrupt. Attempting to access a page without the appropriate permission also results in a page fault interrupt.
- ***Resident***
  The *Resident* bit is used to mark pages which are valid but which are not resident in system memory. These pages may need to be loaded from disk or allocated from a system heap.
- ***Address***
  Pages in system memory are always 4K bytes in size. Each 4K page has its own page table entry, with the *Address* field giving the start address of that 4K page in physical memory.
- ***PageSize***
  The *PageSize* field in the page table provides further information about allocation, indicating that a number of 4K pages are allocated consecutively and start on a suitable boundary. For example, if the *PageSize* field is "2" it indicates the page is one of a group of four consecutive 4K byte pages in physical memory, and also that the logical and physical start addresses of this 16K "page" are aligned to a 16K byte boundary. The *PageSize* field allows address translation hardware to optimise reading of the page tables and reduce the number of TLB updates. All entries in the Page Table must be defined so that every 4K page has an entry.

DMAAddressDecoding

The **MinLogicalPage** and **MaxLogicalPage** registers define the valid range of logical addresses. Even when address translation is enabled, any logical address outside this

range will be treated as a physical address which does not need to be translated. For each DMA source, requests are still issued to the bus masters in the order in which they arrive at the Arbiter, whether they are physical requests or logical requests requiring translation. The page starting at the logical address in **MinLogicalPage** is described by the first entry in the page tables, which is located at address **PageTableBaseAddress** in system memory. No page table entries need to be maintained for logical addresses outside the Min/MaxLogicalPage range since they will never be accessed by the hardware.

When a page fault occurs that needs CPU intervention, the faulting address and the source of the address are made available in the relevant unit's **PageFaultAddress** and **PageFaultSource** registers. Each source of DMA requests can raise a Page Fault without affecting requests being issued from the other DMA sources. A separate **PageFaultAddress** register is provided for each request source to handle this. For example geometry processor read accesses can continue during a bypass page fault.

Where request sources share a common pool of system memory or page table entries, it is the responsibility of page-fault handler sofware on the host CPU to disable all other affected request sources before updating the common resource pool. The **ArbiterRequestControl** register allows each source to be disabled individually for this purpose, and re-enabled once the fault has been fixed. Once all affected requests sources are disabled, the page-fault handler can read the **ArbiterStatus** register to check that the sources are idle. If necessary the handler can write to the **Flush** register to ensure that AGP write data is flushed out to system memory. The **ArbiterStatus** register includes "write in progress" and "Flush in progress" flags to allow the page-fault handler to monitor the status of the PCI and AGP bus masters, as well as the status of each DMA request source.

When the system memory has finally been made available and the page tables updated, a write to the **Restart** register triggers another read of the tables. If everything is correct the transaction completes.

Invalidating obsolete TLB entries

If the page tables are modified then the Translation Look-aside Buffers (TLBs) may hold obsolete data.

The TLB entries for each source can be invalidated by writing to the InvalidateTLB register. This must always be done if the page tables are modified and the TLBs When a bit in this register is set the corresponding TLB is invalidated and remains invalid until the bit is cleared. Leaving a bit set effectively disables caching of the page table entry and forces a new lookup for every request; this may be useful for debug.

If the MinLogicalPage or MaxLogicalPage registers are to be updated, it is advisable to disable requests and wait for all request sources to become idle first. Once these control registers have been updated, then all TLB entries should be invalidated before requests are re-enabled, as the TLBs may now hold incorrect data.

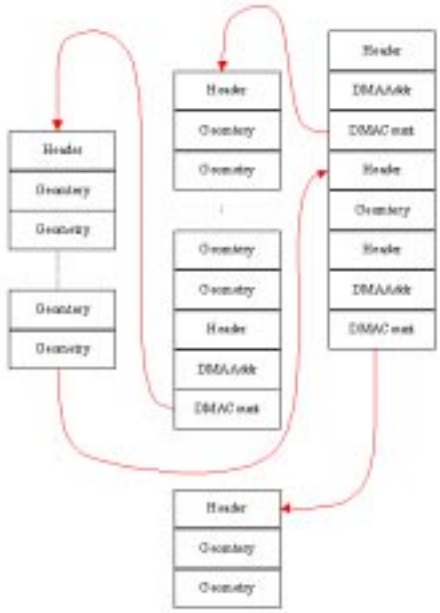
The contents of all the TLB entries can be read back using the ArbiterReadIndex and ArbiterReadData registers. As well as being useful for debugging, these registers provide information which allows the page fault handler to distinguish between a normal non-resident page fault and an invalid access caused by a programming error.

Note that setting the ArbiterReadIndex register to select the Geometry Processor TLB array stall the address translation for the Geometry Processor. When the required data has been read from the ArbiterReadData register the ArbiterReadIndex register should be set to point to a different part of the DMA Arbiter indirect address map, to allow Geometry Processor address translation to resume.

3.2.5.7 Hierarchical DMA

During a DMA transfer, if the **DMAAddr** and **DMACount** commands are encountered in the DMA buffer then the current DMA is temporarily suspended and a new DMA started with the new address and counts. When a 'new' DMA finishes the previously suspended DMA is restarted immediately after the **DMACount** command which initiated the new DMA.

The motivation for hierarchical DMA is that it allows any number of individual DMAs to be



queued up, independent of the input FIFO depth, and it supports hierarchical display lists.

Display lists are held in 'native' R5 format so they can be executed directly without having to be parsed by software first. An OpenGL display list can call other OpenGL display lists and models are typically built as hierarchies of objects.

Those familiar with previous chip families such as the GLINT MX may find it useful to know that restrictions on the use of Hierarchical DMA have largely vanished. In particular, mode changes no longer require that the host keep a software copy of the register to re-generate the bits (in case the register contents change between the time the display list is created and the time it is executed). All R5 mode registers now have an **And** and an **Or** version to allow individual bits to be changed without affecting the display list.

Areas where some care is still required are:

- *Display lists which involve matrix generation and concatenation*
  These still need host intervention. When the user specifies a new view matrix in OpenGL it must be concatenated with the current model view matrix to allow hierarchical models.
- *Interdependencies in mode bits*
  For example the framebuffer, in general, needs to be enabled for reading when alpha

blending is done, however some alpha blending modes don't use the framebuffer contents. Meanwhile, in a different mode register the logical operation may or may not require framebuffer data to be read as well.

3.2.5.8 Rectangle DMA (Input)

Rectangle DMA image data may be a sub image of a larger image and have any natural alignment. Information regarding the rectangle transfer is held in registers loaded from the input FIFO or a DMA buffer.

The pixel data read from host memory is always packed, however when passed to R5 it can be in packed or unpacked format.

The base address of the rectangle is written to the **DMARectangleReadAddress** register; the stride of the rectangle is loaded into the **DMARectangleReadLinePitch** register. The width and height of the rectangle are held in the **DMARectangleRead** command which triggers the DMA. This command also defines the pixel size and the packing of the pixels; the data is always packed in system memory, but if the pack bit is not set it it is broken into individual pixels on load. When the data is loaded it is assigned to the data type defined by the **DMARectangleReadTarget** register.

The **DMARectangleReadControl** register holds system information about how to access the data. It defines the bus protocol to use (AGP or PCI) and whether to byte swap the data. It also has control over the burst size to use, and alignment control which breaks bursts on 64 byte boundaries which can help some chip sets.

If a **CommandInterrupt** is sent, it causes an interrupt to be sent when all previously started rectangle DMAs have completed.

3.3 Vertex Loading

The I/O units support various strategies to make data loading more efficient including early culling, vertex loading and texture formatting, described below. Vertex loading allows a vertex to be described in 4 dwords and DMAd individually or as part of a more complex API array structure

3.3.1 Primitives

Further I/O pre-processing is possible if the data represents a primitive. Vertex data for primitives can be loaded into graphics memory space in a number of ways. It can be:

- written directly to the appropriate vertex store
- loaded indirectly from an index into a vertex array, or
- loaded as part of a special vertex sequence corresponding to one of the following primitive types:

Name	Description
TriangleList	Individual triangles with no shared vertices.
TriangleFan	Triangles with a common center vertex, and another shared vertex between each adjacent triangle.
TriangleStrip	Triangles with two shared vertices between adjacent triangles.
LineList	Individual lines with no shared vertices.
LineStrip	Lines joined head to tail.
PointList	Individual points

Each of these is available in an indexed form (e.g. **IndexedTriangleList**) and a vertex form (e.g **VertexTriangleList**). The indexed form specifies vertices through an index into a vertex array; the vertex form specifies a vertex array directly, so the vertices must be in the correct order.

The register tag marks the start of a primitive and the data field is a count of the number of vertices to be processed. If the number is zero then the data is assumed to be inline and the primitive is terminated by another primitive command.

IndexedVertex or **IndexedDoubleVertex** can be used as inline indices into a vertex array by first sending the **IndexedTriangleList** primitive (for example) with data equal to zero. **IndexedVertex** supplies a single 32 bit index into the array, **IndexedDoubleVertex** supplies two 16 bit indices into the vertex array.

The vertex data can be loaded without specifying a primitive type using **Vertex0**, **Vertex1**, and **Vertex2.** These registers specify the vertex store to load, and the data field holds the index into the array. The **VertexData** register is used for inline vertex data.

The format of the vertex is specified by four controls:

- The vertex size controls the amount of memory each vertex takes (i.e. the Stride).
- The tag list describes the order of the data within the vertex, so the tag loaded into TagList0 defines the data type of the first entry in the vertex.
- The **VertexFormat** mask controls which data elements are present, so if bit 0 is clear but bit 1 set, the first data element read is associated with TagList1 instead of TagList0.
- The **VertexValid** mask specifies which items of data within the current vertex should be read.

Typically we would use the tag list to define the order in which data is delivered. The format mask and vertex size set modes (so if z is enabled the z bit in the format mask is set and the vertex size increased by 1).

Multiple Textures: The **VertexValid** mask is generally used for multi-pass algorithms, such as emulation of multiple textures. The vertex structure holds several sets of texture coordinates for the same x,y,z coordinates, and on each pass a different set is enabled by defining it as valid. Because invalid data is discarded the format only needs to recognize a single field (although the vertex size must be big enough to include all data).

Context Switching: A state register supports context switching within a primitive. This allows a primitive such as a triangle strip to be interrupted part way through and then restarted cleanly. If data is being read by DMA it completes before a sync can get through the pipeline: this applies to single vertices read in by inline indexes and to multiple vertices read in via index arrays.

To restart an interrupted primitive the **HostInState** register should be restored along with the rest of the pipeline and the primitive continued from the point at which it was broken.

3.3.2 Overview

The vertex array unit loads vertex data for processing by R5. It is used to access data that is not in the command stream but is held in separate buffers. This separation of data from commands is common in many APIs as it is generally more efficient. Examples are the flexible vertex format buffer in D3D and vertex arrays in OpenGL.

The difference between OpenGL and D3D is the number of distinct data buffers they support:

- D3D packs everything into a single buffer and requires the application to fit this format.
- OpenGL trades simplicity for flexibility by giving each data type its own data buffer, specifically one each for position, normal, color, edge flags, and eight for each set of texture co-ordinates.

Because OpenGL defines a stride for the data buffers, it can mimic the D3D format by interleaving its separate arrays. DX8 will support multiple data arrays, bringing it into line with OpenGL.

Index buffer: Both D3D and OpenGL also support an index buffer that holds an indirection to the data to be used. Both APIs define a single index that is used by all data types. There is some demand for separate indices for each data type because this can be more efficient and allow better meshing; 3DStudio Max would use this if it were available.

The design allows a single index buffer and up to sixteen data buffers. All the data buffers are accessed via the same data block. The efficiency of bus transfers will decrease as more arrays are enabled. If more index buffers are needed, additional index blocks an be added - one for each buffer.To complete the unit a cache block removes unnecessary vertex loads if the data is already in the vertex stores, and the tag block assigns meaning to the data which is loaded.

When the data has been loaded into R5 it has to be assigned to the correct message type. It may also be necessary to reorder the data to fit R5 protocol rules. R5 requires the position data to be last in a vertex as it is used as a trigger to begin processing, but there is no guarantee that this will be the order in which the data is supplied (D3D guarantees that it will definitely not be the order).

All the data discussed here is read from system memory, using either PCI or AGP protocols. This means that there can be a significant latency between requesting data and receiving it. It is also much more efficient to access incrementing consecutive addresses on PCI than random address; even AGP suffers if addresses do not maintain some degree of locality. Sometimes it is necessary to use PCI protocols even if AGP is available due to PCI cache coherency.

3.3.3 Programming Interface

The normal input format for the data associated with a vertex is IEEE single precision floating point numbers. All the variants supported by OpenGL such as byte, long, double, etc. must be converted to single precision floats first. The only exceptions to this are colors which can be accepted as unsigned packed bytes.

The vertex parameters are multi valued entities (i.e. vectors) so the vectors are assembled before being processed as they can only be written one component at a time. Frequently one or more of the vector components are left to take a default value, for example a vertex

coordinate is commonly specified with a w component set to 1.0. Allowing this component to take its default value saves sending 4 bytes to R5 which can amount to a substantial saving in bandwidth.

The vector assembly uses the x component (for vertices and normals), the red component (for colors) and the s component (for textures) to indicate the assembled vector is complete and should be processed. The address (or tag) of the trigger component used informs R5 which of the shorthand formats to use. Several shorthand formats exist to reduce the amount of information the host needs to provide. The possible formats are:

Vertex:	xy, xyz and xyzw	z = 0, w = 1 if missing
Raster Position:	xy, xyz and xyzw	z = 0, w = 1 if missing
Normal:	Xyz	
Reserved	Xyz	(Face normal)
Color:	Rgb, rgba	a = 1 if missing
Texture:	s, st, strq	t = 0, r = 0, q = 1 if missing

Each shorthand format has a unique register for the trigger component (the number, when present, indicates the number of components supplied):

Parameter	Non Trigger registers	Trigger registers
Vertex:	Vy, Vz, Vw	Vx2, Vx3, Vx4
Raster Position:	RPy, RPz, RPw	RPx2, RPx3, RPx4
Normal:	Ny, Nz	Nx
Color:	Cg, Cb, Ca	Cr3, Cr4
Texture:	Tt, Tr, Tq	Ts1, Ts2, Ts4

The components for a vector are sent in reverse order to ensure the trigger component is always last. The tags are organized so this naturally fits in to the index tag format. The main group for the vertex parameters allows any of the vertex formats, a normal, any of the color formats and the **Ts2** texture format to be encoded into one group. This allows a Gouraud shaded, depth buffered meshed triangle to be described by only 7 words (index tag, **Nz**, **Ny**, **Nx**, **Vz**, **Vy**, **Vx3**).

The storage used to assemble a vector is shared by all vectors so it is not possible to mix components from different vectors. Sending all the components associated with a vector together is not an onerous restriction.

Some general rules when sending vertex parameters are:

- Don't mix up Vertex, Color, Normal, Face Normal, Raster Position or Texture components. All the components for a particular vector type must be sent together.
- The components must be sent in reverse order from the 'natural' way. For example send W, then Z, then Y and finally X (to the appropriate register).
- All components are single precision IEEE floating point numbers.

- Shorthand formats will fill in some components for you to save having to send W =1, for example. These are analogous to the OpenGL *glVertex[2|3|4]f, glColor[3|4]f* and *glTexCoord[1|2|4]f* function calls.
- Only send the vertex coordinate after any color, normal or texture data has been sent. A vertex must be sent to provoke the triangle, line, etc. to be drawn The color, normal and texture update the current color, normal and texture values respectively and are applied to vertices until subsequently changed. This follows the usual OpenGL rules.

These rules are enforced automatically if the parameters are sent as a group using an index tag.

Readback: The above registers, except the vertex, can be read back. All the trigger registers for a parameter read back the same register but the non trigger registers will not read back the current values until the trigger has been written. Non trigger registers will return the default values if the last trigger register substituted the defaults for absent values.

Packed colors: In addition to defining color information [via the **Ca, Cb, Cg Cr3** and **Cr4**] registers, two packed color formats are also available. These are **PackedColor3** and **PackedColor4**. The format of each color component is an unsigned 8 bit number with red in the lower byte, then green, then blue and finally alpha in the upper byte. In the **PackedColor3** case the alpha byte is ignored and always interpreted as 255.

The current edge flag is updated by writing to the **EdgeFlag** register. The current edge flag is used for edges in polygons, independent triangles and independent quads to determine if they should be drawn when the polygon mode is set to Lines. Connected primitives always have their edges drawn. Interior edges introduced as part of the rendering process (for example decomposing a polygon in to several triangles) are never drawn.

3.3.3.1 Vertex Format

The API and internal data formats are defined separately. The input is defined by the API in use, the internal format is defined by Gamma programming rules. This unit maps the API format to the internal format.

The first step in programming this unit is to define the API data format. The input can be thought of as a stream of 32 bit words each holding data for a vertex. Each word has a particular meaning such as X position, texture coordinate S, Diffuse color. The data that makes up this stream may come from different data buffers or from the command stream, but the end result is a sequence of words that have to be loaded into Gamma registers.

The internal data format is the same set of 32 bit words that make up the input, but reordered to meet R5 requirements. The data has to be loaded into R5 registers defined by the following table. The first column shows the type of data, the second column the register name (refered to by its tag) and the remaining 4 columns the contents of each 32 bit field in the 128 bit wide registers. If a field is empty it defaults to a sensible value.

Type	Tag	0..31	32..63	64..95	96..127
Position	Vx2	X	Y		
Position	Vx3	X	Y	Z	
Position	Vx4	X	Y	Z	W
RasterPosition	RPx2	X	Y		
RasterPosition	RPx3	X	Y	Z	
RasterPositon	RPx4	X	Y	Z	W

Type	Tag	0..31	32..63	64..95	96..127
Normal	Nx	X	Y	Z	
Texture	Ts1	S			
Texture	Ts2	S	T		
TextureA	Ts1A	S			
TextureA	Ts2A	S	T		
TextureA	Ts3A	S	T	R	
TextureA	Ts4A	S	T	R	Q
TextureA	Ts3Aq	S	T		Q
TextureB	Ts1B	S			
TextureB	Ts2B	S	T		
TextureB	Ts3B	S	T	R	
TextureB	Ts4B	S	T	R	Q
TextureB	Ts3Bq	S	T		Q
TextureC	Ts1C	S			
TextureC	Ts2C	S	T		
TextureC	Ts3C	S	T	R	
TextureC	Ts4C	S	T	R	Q
TextureC	Ts3Cq	S	T		Q
TextureD	Ts1D	S			
TextureD	Ts2D	S	T		
TextureD	Ts3D	S	T	R	
TextureD	Ts4D	S	T	R	Q
TextureD	Ts3Dq	S	T		Q
TextureE	Ts1E	S			
TextureE	Ts2E	S	T		
TextureE	Ts3E	S	T	R	
TextureE	Ts4E	S	T	R	Q
TextureE	Ts3Eq	S	T		Q
TextureF	Ts1F	S			
TextureF	Ts2F	S	T		
TextureF	Ts3F	S	T	R	
TextureF	Ts4F	S	T	R	Q
TextureF	Ts3Fq	S	T		Q
TextureG	Ts1G	S			
TextureG	Ts2G	S	T		
TextureG	Ts3G	S	T	R	
TextureG	Ts4G	S	T	R	Q
TextureG	Ts3Gq	S	T		Q
TextureH	Ts1H	S			
TextureH	Ts2H	S	T		
TextureH	Ts3H	S	T	R	
TextureH	Ts4H	S	T	R	Q
TextureH	Ts3Hq	S	T		Q

Type	Tag	0..31	32..63	64..95	96..127
Color	Cr3	R	G	B	
Color	Cr4	R	G	B	A
EdgeFlag	EdgeFlag	Flag			
PackedColor	PackedColor3	RGB			
PackedColor	PackedColor4	RGBA			
PackedDiffuse	PackedDiffuse	RGBA			
PackedSpacular	PackedSpecular	RGBF			
PackedNormal	PackedNormal	XYZ			
PackedFace Normal	Reserved	XYZ			

The *ArrayTagTable* registers should be set to the order of the internal format. *ArrayTagTable0* should hold the first register to load, *ArrayTagTable1* the next, and so on. R5 requires that the last register loaded is the position data.

The input data stream is mapped to the registers by the **ArrayFormatRef** and **ArrayFormatField** registers. The maximum size of a vertex is 64 dwords, and for each dword there is an entry in these registers. The **ArrayFormatRef** register holds the entry in the *ArrayTagTable* that the dword should use as a reference to the register to load. The **ArrayFormatField** register specifies which 32 bit word in the 128 bit wide register to load.

The specification of data formats has so far assumed that all possible data for a vertex is being loaded. The **ArrayFormatLoad** register is used to specify that data which will be read from memory, so, for example, if texture is disabled no texture coordinates need be read. Using **ArrayFormatLoad** avoids reprogramming the **ArrayFormatRef** and **ArrayFormatField** registers when the vertex format changes.

In some situations it may be useful to load data that will not be used, in which case the **ArrayFormatValid** register is used to specify data that has been loaded but should not be used. Normally this register can be set to 0xFFFFFFFF to mark all data as valid, but sometimes data needs to be skipped; for example, a multipass algorithm may need access one set of texture coordinates on the first pass and another set on the second pass. It may be more efficient (in terms of bus bandwidth) to load a vertex in a single operation even if some data will not be used, rather than load exactly the amount needed but in several smaller transfers.

Note: *All data is 32 bits wide, matching the normal floating point format. Some data such as color or normals may be packed, but they are always packed into 32 bits. Edge flags are single bit booleans, but are also held in 32 bit words. Although this is inefficient they are rarely loaded through a vertex array.*

3.3.3.2 Array Format

When the vertex format has been defined the source of the data has to be set up. The data can either be indexed or direct. If it is indexed an index has to be supplied that is used as an offset into an array of data. The index can be supplied inline with the command stream, or from a separate array. The index array reads from index array 0 (additional indexes will be supported in future).

The index is used to access data arrays - it calculates the address to read data from using the formula:

```
address = array base address + (index * array stride)
```

Single and Multiple Vertex Loading: There are two ways to specify the index to load from an index buffer. The **ArrayIndexSingle** command has a data field that specifies the offset into the array of a single index to load. The other way to load an index from an index buffer is with the **ArrayIndexMultiple** command which has a count of the number of indices to load as its data field. The first index read is pointed to by the base address register combined with the **ArrayIndexOffset** register.

An index can be 8, 16, or 32 bits, and this is controlled by the **ArrayIndexSize** register; also in the **ArrayIndexSize** register is the *Stride* field which can also be set to 8, 16, or 32 bits. The *stride* is used to describe the spacing between indices, so each 8 bit index can take 32 bits of storage. The *stride* specifies how much data is loaded, while the size specifies how much of that data is used.

Note: The index array is different to the vertex data arrays which load only the data specified by the size field and use the stride field to jump between vertices.

Some registers are unique for each index or data array supported. The array number that is added to the end of their name distinguishes these registers. For example, the base address for a data array is held in **ArrayDataAddress**, so the address for data array zero is in ArrayDataAddress0, for the next data array in **ArrayDataAddress1**, etc. In this text the array number may be replaced by the wildcard # if the reference is not specific.

How Much Data to Read: The amount of data read for a vertex can be programmed through the size field in the **ArrayDataSize**# register, which is separate to the array stride that is in the stride field of the same register. Each data array is programmed to respond to a particular index array, set by the **ArrayDataIndex**# register. This register holds a mask of the indicies to respond to, making it possible for a data unit to be triggered by more than one index; this can be useful when vertices are blended and different index arrays point to the same data. In the example given here there is a single index array, zero, that can be read so all of the ArrayDataIndex registers are set to 0x0001.

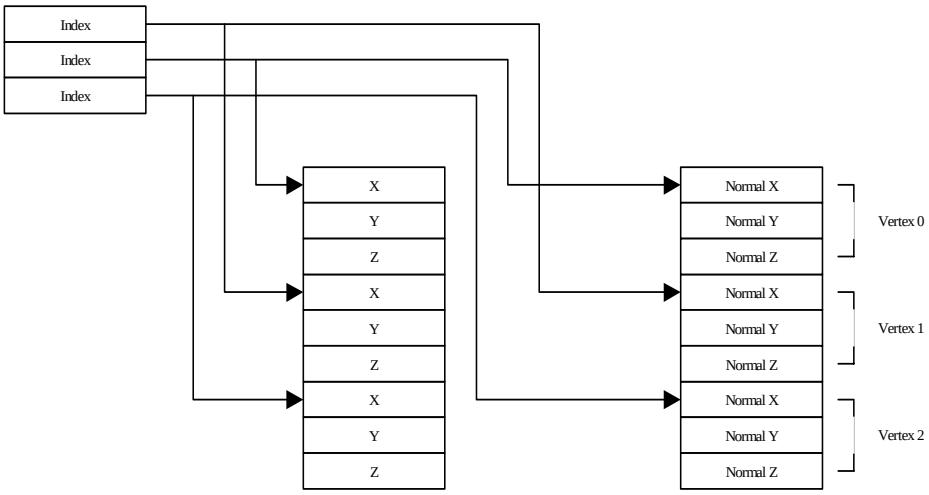
How to Load Additional Indices: If more indices are needed than there are supported index arrays, the additional indices can be loaded in line. The **ArrayDataIndex**# register for appropriate data arrays are loaded with, for example, 0x0002 and **ArrayIndex1** commands sent. These commands index into the selected data arrays (see below for an example).

Data does not have to be loaded via an index, but can be read directly into R5. Two mechanisms are provided, **ArrayDataSingle** and **ArrayDataMultiple**. The **ArrayDataSingle** command reads a single item of data from the index specified by the data field of the command. The **ArrayDataMultiple** command loads the number of data items specified in the data field of the command starting at **ArrayDataAddress**# plus **ArrayDataOffset** and incrementing consecutively until all the data has been read.

The **ArrayEnable** register specifies which index arrays and which data arrays are enabled. For OpenGL each data type comes from its own array, so the data array fields of **ArrayEnable** will match the **ArrayFormat*** register. D3D loads all data from a single buffer, so it only enables one array; OpenGL also has the concept of interleaved arrays which can be loaded as a single array.

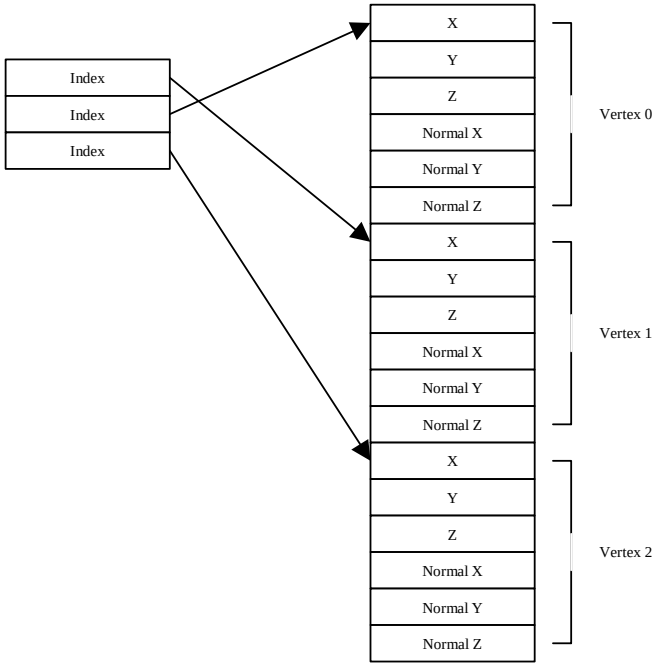
3.3.3.3 Example 1 - Typical OpenGL Layout

The diagram shows a typical OpenGL layout. The position and normal data has been separated. Each array can be referenced by a common index array (i.e. **ArrayIndexMultiple**), or they can be read as consecutive vertices (i.e. **ArrayDataMultiple**).



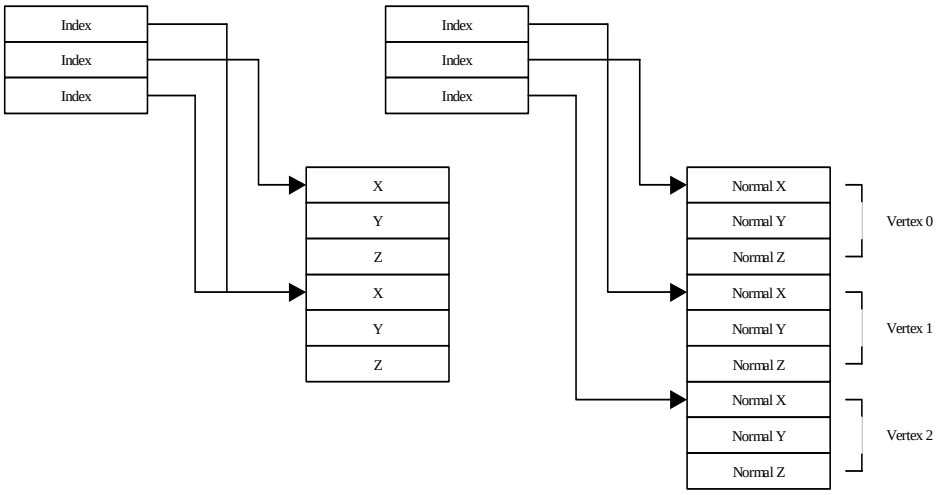
3.3.3.4 Example 2 - D3D Flexible Vertex Layout

The diagram shows a D3D flexible vertex layout that groups all elements of a vertex in the same array. An index can be used to reference the data (using the command **ArrayIndexMultiple**), or the data array can be scanned in order (using **ArrayDataMultiple**). Note that although it is possible to view a single array as a number of interleaved arrays (see example above), it is better to treat it as a single array. This may be more efficient, and should never be less efficient, due to the pattern of accesses generated on the bus.



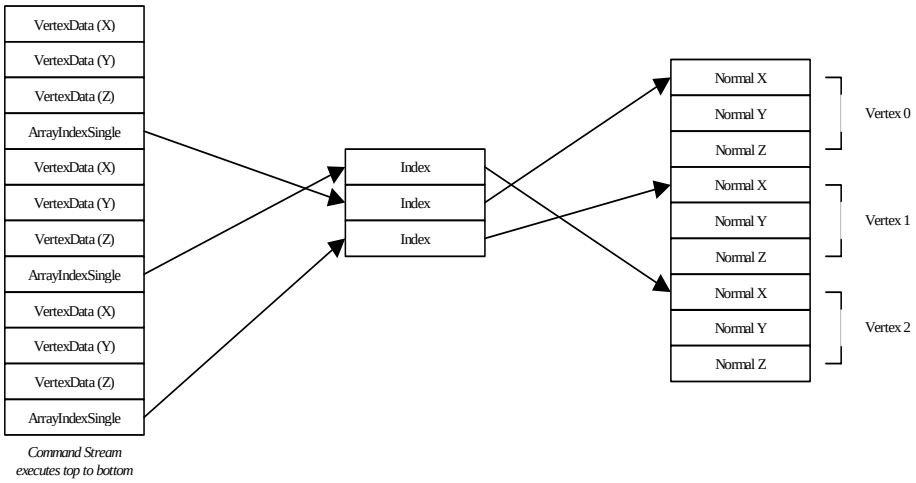
3.3.3.5 Example 3 - Index Array per Vertex Array

The next diagram shows an index array for each data array. In this example there is less position data than normal data because meshing has allowed sharing of position data even though each triangle represented by these vertices needs a different normal. This sort of data sharing relies on indirection through an index array, so this operation would be initiated by the **ArrayIndexMultiple** command.



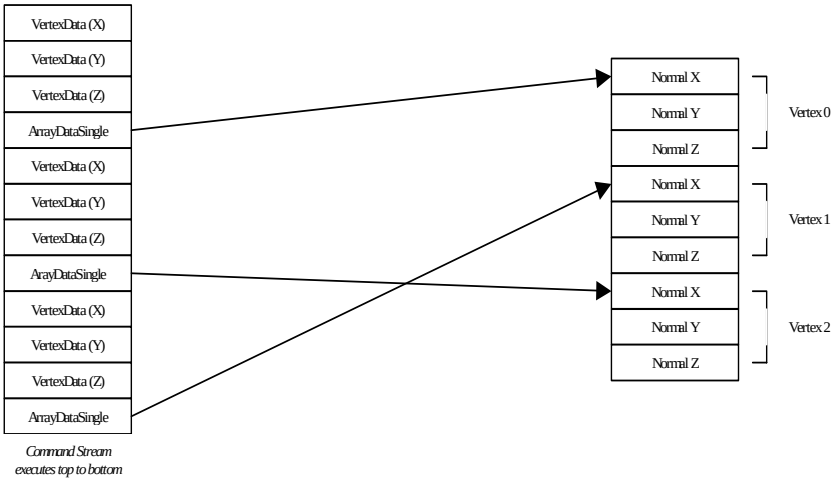
3.3.3.6 Example 4 - Loading Data Inline with Commands

Data does not have to be read from a buffer, but can instead be loaded inline with the commands. This is done by writing to the **VertexData** register in the order that the data would have been read from memory. Data loaded in this way cannot be mixed with **ArrayIndexMultiple** or **ArrayDataMultiple** commands, but can be mixed with **ArrayIndexSingle** or **ArrayDataSingle** commands that cause a single item to be read. The data stream can be made up of an arbitrary combination of inline data and array commands. The diagram shows X, Y, and Z, being loaded inline with the command stream; ArrayIndexSingle inserted at the right point in the commands loads the normal data from a separate buffer.



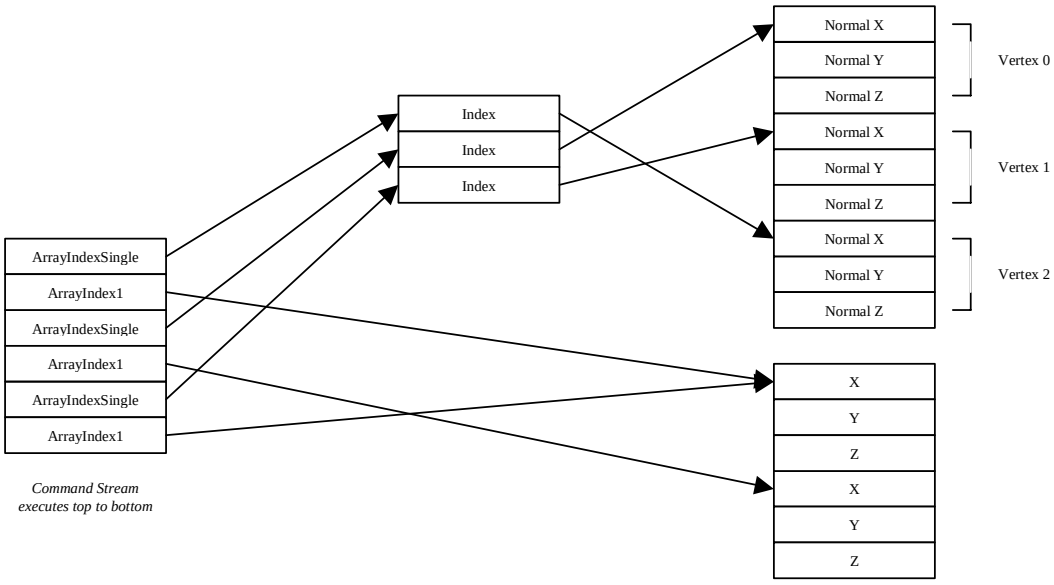
3.3.3.7 Example 5 - Using ArrayDataSingle

The same effect as example 4 can be achieved by removing the indirection through the index array and going direct to the data with **ArrayDataSingle** commands.



3.3.3.8 Example 6 - Emulating Index Buffers with an Inline Command

This example shows how two index buffers can be emulated by an inline command. If a single index array is available then **ArrayIndexSingle** commands are issued that increment through the array. The other array is emulated with the **ArrayIndex#** command. If the data unit fetching the position data has been configured to respond to references from index buffer 1 (by setting ArrayDataIndex for that data unit to 0x0002) it reads the data as though the inline indices had been read from an array. The second index buffer is copied into the command stream, which requires the CPU to do an extra read per vertex.



*Note: The **ArrayDataSingle** commands used in example 5 can be replaced by **ArrayIndex#** commands if the data units have been set up appropriately.*

3.3.3.9 Mapping to OpenGL Commands

Each OpenGL vertex array command causes the following R5 registers to be modified:

GL function	R5 Register/Command
GlEnableClientState	ArrayEnable, ArrayFormatLoad, ArrayEnable
GlDisableClientState	ArrayEnable, ArrayFormatLoad, ArrayEnable
GlVertexPointer	ArrayDataSize#, ArrayDataAddress#, ArrayTagTable
GlColorPointer	ArrayDataSize#, ArrayDataAddress#, ArrayTagTable
GlNormalPointer	ArrayDataSize#, ArrayDataAddress#, ArrayTagTable
GlEdgeFlagPointer	ArrayDataSize#, ArrayDataAddress#, ArrayTagTable
GlTexCoordPointer	ArrayDataSize#, ArrayDataAddress#, ArrayTagTable
glIndexPointer	ArrayIndexSize0, ArrayIndexAddress0
glArrayElement	ArrayDataSingle
glDrawElements	Begin, ArrayIndexAddress#, ArrayIndexSize#, ArrayIndexMultiple
glDrawArrays	Begin, ArrayDataOffset#, ArrayDataMultiple

Each data type is mapped to a particular array number that is used to define the # in the register names. If the implementation of R5 does not have enough data units the command stream would have to be constructed out of individual index commands described above.

3.3.3.10 Vertex Caching

R5 supports meshed primitives such as triangle strips and triangle fans, but it is common practice for applications to represent meshes as separate triangles. Separate triangles specify all three vertices so take three times as much AGP bandwidth as a mesh, so this unit caches data in the vertex stores to reduce unnecessary reads. Caching applies to triangle lists and line lists only and is enabled in the **ArrayMode** register *IndexCache Enable* field.

3.4 Texture Loading

The need to load and manage texture maps usually imposes severe size and performance constraints on programmers and requires careful consideration of the tradeoffs between frame rates and scene realism.

R5 implements logical and virtual texture management to remove many of the bottlenecks typically encountered during application development. However the configuration and control of the various texture DMA options requires careful planning for optimal results. The discussion below summarizes the use of the Texture DMA and texture loading. For a detailed description of Logical and Virtual texture management, see the *R4 Programmers Guide Volume I*, chapter 4 on "Buffer and Cache Management", particularly sections 4.9 ("Texture Mapping") and 4.10 (Virtual Texture Management").

More generally, Texture functionality is determined in the Geometry pipeline in the Matrix, Texture/Fog, Material and Texture Format units. These control the use of (for example) Diffuse and Specular texture enables, clipping and co-ordinate transformation. These are discussed below in "The 3D pipeline".

Texture implementation (e.g. texture composition, texture application, logical and physical texture page setup, texture LUT and LOD calculations) takes place in the Rasterizer pipeline.[10]

The design philosophy is that adequate texture control should be possible using OpenGL or D3D only - where functionality is required it will be available in the Geometry pipeline. So, for example, the **Begin** register also contains fields duplicated from the **Render** and **Draw*** registers including the master texture enable. This simplifies the match between R5 facilities and API functions, e.g. *glEnable()* or *glTexEnv*()*.

3.4.1 Overview

3.4.1.1 Start of Day Initialisation

Before any logical or virtual texture management can be done there are a number of areas which need to be initialized (in addition to the usual mode, etc. register initialization):

- Space for the Logical Texture Page Table must be reserved in the local buffer and the table initialised to zero. The **LogicalTexturePageTableAddr** and **LogicalTexture PageTableLength** must be set up.
- Space for the working set must be reserved in the local buffer and/or framebuffer. This need not be physically consecutive pages. The **BasePageOfWorkingSet** register is set up.

If virtual texture management is to be used then the following additional initialization is required:

[10] See Chapter 2 of the *R5 Reference Guide Volume I* for an illustration of the R5 pipeline architecture.

- Space for the Physical Page Allocation Table is reserved in the local buffer and **PhysicalPageAllocationTableAddr** register is set up to point to it.
- Bits 0…31 of each entry in the Physical Page Allocation Table is set to zero - to clear the valid bit.
- Each page entry in the Physical Page Allocation Table is associated to one of the four pools based on which bank of memory it resides in. All the pages in a pool are linked together as a double linked list by setting the *NextPage* and *PrevPage* fields. The order is unimportant, but sequential is simplest. The *PrevPage* field for the first entry in the double linked list and the *NextPage* field for the last entry can be set to any value as they are not used.
- Finally the **HeadPhysicalPageAllocation** and **TailPhysicalPageAllocation** registers for this memory pool are updated with first and last page numbers.

Each memory pool (to a maximum of 4) is set up as above.

Note: Unused memory pools must not be referenced in the Logical Texture Page Table.

The texture management hardware is ready to be used once logical textures have been created. The texture management can be done on a global basis so all contexts/APIs share the same mechanisms, or can be done on a context by context basis.

3.4.1.2 Creating and Loading Texture Maps

The sequence of events when the application asks for a texture to be loaded are as follows:

- Host memory to hold the texture map is allocated and locked down. This memory is private to the driver or ICD and not accessible to the application. The pages do not need to be contiguous.
- The logical pages to use for the texture map are allocated from the Logical Texture Page Table. These may be new pages or currently assigned. If they are currently assigned then the texture management hardware will do any necessary housekeeping to prevent aliasing of physical pages to the same logical page (thereby degrading the performance although still functioning correctly).
- The host physical page (or host virtual page when host virtual addressing is used) of each page reserved for the texture is found and the *HostPage* field for each corresponding entry in the Logical Texture Page Table is updated with it.
- The memory pool this texture is to be stored in is determined[11] and each logical entry has its *MemoryPool* field set appropriately.
- The *Length* field for each logical entry will normally be set to 0x100 (i.e. 4096 bytes), however as an optimisation if only part of the 4K page is used (must be the lower part) then the number of 64 bit words used can be used instead.
- The application's texture is copied into the previously allocated host memory and during the copy the texture map is patched and aligned as required by the setting the texture map will be invoked with[12].

---

[11] This may be difficult to determine as the usage of the texture maps is not available. Ideally texture maps to be used simultaneously should be in different pools, unless they can both fit into the same 4K page.

[12] It is impossible to do any patching or aligning while the page of texture is downloading as the download mechanism has no knowledge of the dimensions of the texture map, its base address, layout or texel size.

The preferred way to update the Logical Texture Page Table is to use the **SetLogicalTexturePage** and **UpdateLogicalPageInfo** commands. The **SetLogical TexturePage** command takes the logical page to update in the least significant bits. The **UpdateLogical PageInfo** command sets bits 0…31 to zero and updates bits 32…63 with the given data. The entry to update was set by the **SetLogical TexturePage** command and this is auto incremented after the update. All the necessary housekeeping is done.

Alternatively the Logical Texture Page Table can be edited by software by reading and/or writing it directly to the table in memory using bypass memory accesses. In this case it is the software's responsibility to do the necessary housekeeping to remove any references to the updated logical pages in the Physical Page Allocation Table.

After this set up has been done the texture map can be bound and used. Note that the texture map (or pages of it) are not loaded until it is actually used.

3.4.1.3 PreLoading Texture Maps

Although the texture map is only downloaded when it is used it can sometimes useful to ensure it is downloaded when it is created. This is done using the Load mode to load each logical page in the texture map. Alternatively when a texture map is bound (to a context) you may want to ensure it is resident at that time, rather than wait for a page fault. If the page is already resident then there is no need to load it (as the Load mode would do) so the Touch mode can be used instead. These can be done using the command **TouchLogicalPage**. This command has the following data fields:

Bit No.	Name	Description
0…15	Page	This field set the first Logical Page to touch.
16…29	Count	This field holds the number of pages to touch.
30…31	Mode	This field is set to 3 to touch a page(s) or to 1 to load a page(s).

As each page is touched the corresponding texture data is downloaded.

3.4.1.4 Editing Texture Maps

To edit the texture map (for example as part of a *TexSubImage* operation in OpenGL) the host's copy is edited. The texture management hardware is notified that the texture pages (if resident) are stale by using **TouchLogicalPage** to mark these pages as non resident. This command has the following data fields:

Bit No.	Name	Description
0…15	Page	This field set the first Logical Page to mark as stale.
16…29	Count	This field holds the number of pages to mark as stale.
30…31	Mode	This field is set to 0 to mark the pages as stale (i.e. non resident).

The primary texture cache is invalidated (using the **InvalidateCache** command) to ensure it doesn't hold any stale texel data for the texture map just edited.

3.4.1.5 Deleting Texture Maps

Texture maps do not need to be deleted. Simply reusing the logical address achieves the same thing. If you really want to delete the pages then the **TouchLogicalPage** command can be used to mark them non resident[13].

3.4.1.6 Locking Down Texture Maps

The best way to have locked down texture maps (i.e. they don't get swapped out) is to avoid using logical/virtual management and have them as physical textures. If a texture is to be locked down *after* is has been created as a logical texture then the software must edit the Physical Page Allocation Table (and possibly the **HeadPhysicalPageAllocation** and/or **TailPhysicalPageAllocation** registers for the affected pools). The system must be idle before these edits can take place to avoid texture downloads in mid-edit with unpredictable results.

3.4.1.7 Using Logical Mapping without Virtual Management

Logical texture mapping can be used without virtual management. This allows textures to be mapped over non-contiguous physical memory without automatic loading. Set up this way, textures are managed similarly to the GLINT MX, but memory management is simpler because physical memory allocation is now done on pages rather than variable-size texture maps.

To do this all current logical textures must be resident so a page fault will never occur. When a texture is created the software needs to do two things:

- Allocate the physical memory and update the Logical Texture Page Table with the logical to physical mappings. The physical page for each corresponding logical page is stored in bits 0…15 and the resident bit (bit 16) is set. The second word in each entry is not used (it would only be accessed on a page fault). The Logical Texture Page Table can be modified directly via the bypass (after syncing) or can be updated via the command stream. The **DownloadAddress** register and **DownloadData** commands can be used to update an arbitrary region of memory. This allows them to be used to update the logical entries in the Logical Texture Page Table[14].
- The texture map must be downloaded into the physical pages. This can be done via the bypass mechanisms or through the command stream. In either case it is the software's responsibility to do any patching and alignment consistent with how the texture map will be used.

Note *The texture download mechanism which can do the patching doesn't have any method of remapping the addresses so cannot work with non-contiguous physical memory. The* **DownloadAddress** *register and* **DownloadData** *command can be used to download each page of texture (pre-patched, if necessary) into its corresponding physical page.*

3.4.2 Programming Notes for Host Textures

Texture maps stored in host memory can be managed by the virtual management hardware. This allows a texture map to be split over non-contiguous pages of host memory

---

[13] This does not mean that these pages are made the least recently used pages so they get reused sooner - they will percolate to this status subsequently just through inactivity.

[14] The **UpdateLogicalPageInfo** command cannot be used as it zeros the physical page field and updates the fields concerned with page faults. Also this command does housekeeping work on the Physical Page Allocation Table, which presumably will not have been set up if the virtual texture management is not being used.

(without relying on the AGP GART table to do the logical to physical mapping) and texture maps to be paged in and out of this memory.

The host pages are not part of the physical memory pool managed by the hardware so all host pages are allocated (or reallocated) by host software.

3.4.2.1 Start of Day Initialisation for Host Textures

Assuming the range of logical pages reserved for host texture management is already included in the length of the Logical Page Table then no further initialisation is needed other than to set up the **BasePageOfWorkingSetHost** register with the address of the region to manage. This is a 256MByte region and can be positioned anywhere in the 4G host address range.

No changes to the Physical Page Allocation Table are needed.

3.4.2.2 Creating Logical Texture Maps for Host Textures

The sequence of events when the application asks for a texture to be loaded is as follows:

- Host memory to hold the texture map is allocated and locked down[15]. This memory is private to the driver or ICD and not accessible to the application. The pages do not need to be contiguous.
- The logical pages to use for the texture map are allocated from the Logical Texture Page Table. These may be new pages or currently assigned. If they are currently assigned then the TLB should be invalidated to prevent it from holding stale addresses.
- Each logical page has its physical page, resident and host texture fields in the Logical Page Table updated with the corresponding host physical page where the texture is located. The length field must be set to zero (to disable a download from occurring). The pool field and the *HostPage* field are not used (but are available to software to hold information about this page).
- The application's texture is copied into the previously allocated host memory. During the copy the texture map is patched and aligned as required by the setting the texture map will be invoked with[16]

[15] Virtual host memory could be used, however the driver will need to respond to every page fault and make the textures available in locked physical memory before starting the DMA off to download them.

[16] It is impossible to do any patching or aligning while the page of texture is downloading as the download mechanism has no knowledge of the dimensions of the texture map, its base address, layout or texel size.

3.4.2.3 Logical Page Table

The Logical Page Table has one entry per logical page and each entry has the following format:

Bit No.	Name[number of bits if not 1]	Description
0…15	PhysicalPage[16]	These bits hold the physical page number relative to the start of the working set where this logical page is held. If the page is not resident (next field) then these bits are ignored (but will frequently be set to zero). This field is normally maintained by P4, except when the page is marked as a HostTexture.
16	Resident	This bit, when set, marks this logical page as resident in the working set. This field is normally maintained by P4, except when the page is marked as a HostTexture.
17	HostTexture	This bit, when set, marks this logical page as resident in the host memory and it should be accessed using AGP texture execute mode rather than downloading it. The Length field should also be set to zero.
18…31	reserved	This field is not used but is set to zero whenever the Resident bit is updated.
32…40	Length[9]	This field holds the number of 128 bit words to transfer when a page fault occurs. This allows a page to hold a texture map smaller than 4K without spending the extra download time on the unused words. There is no way to download to unused portion without overwriting the used part. When the physical page is in host memory the length field must be set to zero. This field is maintained by the host.
41…42	MemoryPool[2]	This field holds the memory pool this logical page should be allocated out of. This field is maintained by the host.
43	VirtualHostPage	This bit, when set, indicates the HostPage (next field) is a virtual page in host memory so cannot be accessed directly. Setting this bit will generate an interrupt and involve the host in providing this page of texture data. When this bit is 0 the HostPage is the physical page and will be read directly with no host intervention. This field is maintained by the host.
44…63	HostPage[20]	This field holds the page in host memory where the texture data is held. This is a virtual host page or a physical host page as indicated by the VirtualHostPage bit (previous field). This field is maintained by the host.

The first word in each entry is basically read and written during the memory management activities unless the page is a host texture in which case the host is responsible for the first

word as well. The second word is written by the host (either directly via the bypass or via the core using messages) and just read by R5.

The preferred way to update the Logical Texture Page Table is to use the **Download Address** and **DownloadData** commands. The **DownloadAddress** command takes the byte address in memory of the Logical Page Table Entry to update. The **DownloadData** command writes its data to memory and then auto increments the address. Two words are written per logical page entry. After the Logical Page Table has been updated the TLB must be invalidated to prevent it holding stale data (use the InvalidateCache command with bit 2 set) and **WaitForCompletion** used to ensure the table in memory has been updated before any rendering can start[17].

Alternatively the Logical Texture Page Table can be edited by software by reading and/or writing it directly to the table in memory by using bypass memory accesses methods. In this case it is the software's responsibility to Sync with the chip first to ensure no outstanding rendering is going to use a logical page about to be updated. The TLB still needs to be invalidated after the bypass updates have been done. After this set up has been done the texture map can be bound and used.

3.4.2.4 PreLoading Texture Maps for Host Textures

This is not meaningfull unless they are virtually managed, in which case they can be touched like non-host textures. This is because the texels are read on demand and not downloaded as pages.

3.4.2.5 Editing Texture Maps for Host Textures

The procedure is identical to that for non-host textures (above).

3.4.2.6 Deleting Texture Maps for Host Textures

Texture maps do not need to be deleted explicitly. Reusing the logical page has the same effect.

3.4.2.7 Virtual Host Textures

Virtual host textures are textures which live in virtual host memory so do not need to be locked down into physical memory. As a result they are not guaranteed to be present when a corresponding page fault occurs, and in any case the Logical Texture Page Table only holds the virtual page address and not the physical page address.

The Logical Texture Page Table will have the *VirtualHostPage* bit set, the resident bit clear, the host texture bit set and length field zero for these logical pages.

The DMA controller raises an interrupt (even though no download is needed the DMA controller is involved so the same software interface can be used).

On receiving this interrupt the **HostTextureAddr**, **LogicalPage** and **TextureOperation** PCI registers are read to identify the faulting texture page. When the data is available in locked memory the Logical Page Table is updated via the bypass and the **HostTextureAddress** PCI register is written (the data is not used). The write to the **HostTextureAddress** register will wake up the texture download DMA controller but because the length field is

[17] The writes to the Logical Page Table are done as Framebuffer Writes so may still be queued up on the subsequent TLB miss, hence stale page data will be read from the Logical Page Table. The **WaitForCompletion** command ensures this cannot happen.

zero no download is done or physical page (from the Physical Page Allocation Table) allocated. The TLB is automatically invalidated.

In servicing the interrupt a physical page (or pages if the interrupt is used to allocate a whole texture rather than just a page) must be allocated by software. If these physical pages are already assigned then the corresponding logical pages must be marked as non resident in the Logical Texture Page Table. If these newly non resident logical pages are subsequently accessed (maybe by a queued texture operation) they themselves will cause a page fault and be re assigned. Hence no knowledge of what textures are waiting in the DMA buffer to be used is necessary. The physical pages are allocated from the host working set whose base page is given by **BaseOfWorkingSetHost** register.

3.4.3 Loading 3D and Other Textures

3.4.3.1 3D Textures

A 3D texture map is one where the texels are indexed by a triplet of coordinates: (u, v, w) or (i, j, k) depending on the domain and is typically used for volumetric rendering.

The texture map is stored as a series of 2D slices. Each slice is stored in an identical fashion to all other 2D texture maps. The first slice (at k = 0) is held at the address given by **TextureBaseAddr0** and the remaining slices are held at integral multiples of *TextrueMapSize* (measured in texels) from **TextureBaseAddr0**.

3D texture mapping is in this unit is enabled by setting the *Texture3D* bit (bit 11) in **TextureReadMode0** (the same bit in **TextureReadMode1** is always ignored). The layout, texel size, texture type and width should be set up the same for texture 0 and texture 1.

When 3D texture is enabled then any bits to control dual textures or mip mapping are ignored. The *CombinedCache* mode bit should not be set when 3D textures are being used.

3.4.3.2 Bitmaps

Bitmap data can be stored in memory and accessed via the texture mapping hardware. The resulting 'texel' data is treated as a bitmap and used to modify the pixel or color mask used in a span operation.

The bitmap data can be held at 8, 16, 32 or 64 bit texels and is zero extended (when necessary) to 64 bits before being optionally byte swapped, optionally mirrored, optionally inverted and ANDed with the pixel mask or the color mask. Bitmaps use the secondary texture cache, not the primary texture cache.

The bitmap data can only be held in Linear or Patch64 layouts - Patch32_2 or Patch2 formats are not supported, however no interlocks prevent their use - the results are just not interesting or useful. The bitmap data can be stored as logical or physical textures.

The bitmap data can be held as packed 8, 16, 32 or 64 bit data, usually with one scanline of the glyph held per texel. Glyphs wider than 64 bits will take multiple texels to cover the width. Packing multiple scanlines together reduces the waste of memory (in MX the texel size was limited to 32 bits for spans), and makes the cacheing more efficient.

Before the texel can be used it is processed as follows:

- The bitmap texel is zero extended up to 64 bits.
- The texel is byte swapped according to TextureReadMode0.ByteSwap field. If the 64 bit word has bytes labelled: ABCDEFGH then the three bits swap the bytes as follows:

Bit 2 (long swap)	Bit 1 (short swap)	Bit 0 (byte swap)	Swapped ABCDEF GH
0	0	0	ABCDEFGH
0	0	1	BADCFEHG
0	1	0	CDABGHEF
0	1	1	ABCDEFGH
1	0	0	EFGHABCD
1	0	1	FEHGBADC
1	1	0	GHEFCDAB
1	1	1	HGFEDCBA

- Next the texel is optionally mirrored. This is controlled by the **TextureReadMode0** *Mirror* bit. The mirror swaps bits:
- (0,. 63), (1, 62), (2, 61),…(31, 32).
- The texel is next optionally inverted under control of the **TextureReadMode0** *Invert* bit.
- When TextureReadMode0 field OpaqueSpan is zero the texel is ANDed with the pixel mask to remove pixels from the mask. When TextureReadMode0.OpaqueSpan is 1 the texel is ANDed with the color mask instead.
- 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.

Windows normally supplies its bitmasks as a byte stream with successive bytes controlling 8 pixel groups at increasing x (i.e. towards the right edge). Bit 7 within a byte controls the leftmost pixel (for that group) and bit 0 the right most pixel. To match up the pixel mask order (bit 0 controls the left most pixel, bit 63 the right most pixel) the three byte swap bits are all set and the mirror bit set.

3.4.3.3 Indexed Textures

Indexed textures are a special case because they are stored as 8 bit texels and expanded to 32 bit texels when loaded. This makes the addressing and cache management slightly more complicated as addressing uses 8 bit texels while cache management uses 32 bit texels.

The secondary cache holds the texture data in its 8 bit format which reduces the number of memory reads when the access path is mainly in u across the texture map.

3.4.3.4 YUV 422 Textures

YUV textures are a special case because two texels are stored in a 32 bit word (so in this sense they are 16 bit texels), however the U and V components are shared so the 32 bit word represents two 24 bits texels (the spare 'alpha' byte is set to 255). If the input bytes in the 32 bit word are labelled:

```
Y1  V0  Y0  U0      (U0 in the ls byte)
```

then the two output words are formed (in the internal format):

```
255  V0  U0  Y0    and     255  V0  U0  Y1    (Y in the ls byte)
```

This arrangement of the YUV pixels in memory is called YVYU, but an alternative memory format (called VYUY) is also supported. In this case the bytes are labelled:

```
V0  Y1  U0  Y0      (Y0 in the ls byte)
```

3.4.3.5 Borders

Borders (in the OpenGL sense) are used only when the filter mode is bilinear and the wrapping mode is clamp. In this case when one of the filter points goes outside the texture map the border texel is read or (if it is not present) the border color is used. The border, if present, still needs to be skipped over and this will have already been done by incrementing the i, j indices before they arrive.

The width of a texture map is (2n + 2b) where b is 0 (no border) or 1 (border). Unfortunately the texture map width cannot simply be set to this value because the lower resolution mip map levels 'divide out the border' as the width is divided by 2 for each successive level. The **TextureMapWidth0** and **TextureMapWidth1** registers hold the width of the texture map without the border (in bits 0…11). If a border is present the border bit (bit 12) in **TextureMapWidth0** or **TextureMapWidth1** is also set.

If a 1x1 texture map has a border then the 3x3 map is stored as a 4x4 map as shown:

b0	b1	b2
b3	t0	b4
b5	b6	b7

b0	b1	b2	b2
b0	b1	b2	b2
b3	t0	b2	b4
b5	b6	b7	b7

Texels which fall into the border when no border is present are flagged - these texels are not checked in the cache and no texels are read from memory. The **T0BorderColor**… **T7BorderColor** flags used for this purpose select the **BorderColor0** (T0…T3) or **BorderColor1** (T4…T7) registers instead of the primary cache to provide the texture data. The **BorderColor0** and **BorderColor1** registers would normally be set to the same value for OpenGL when mip mapping.

3.4.4 Texture DMA Controller and Texture Implementation

This topic is discussed in the chapters covering use of texture in the Geometry and Rasterizer pipelines. See Section 7, *3D Pipeline*, below, and the *GLINT R4 Reference Guide* for more information.

4

Getting Data Out of R5

The Output DMA Controller provides a means to transfer data from the Host Out FIFO in R5 to the host's memory. The output DMA is useful for image uploads, returning Feedback data, returning Select data and Context Dumps.

The DMA is initiated by commands (rather than PCI register writes) so it can be queued up in the Input FIFO or input DMA buffers.

The output DMA runs asynchronously to the input DMA so command and register updates are unaffected. If a second output DMA is initiated before the first one has finished then all subsequent command and register processing is suspended until the first output DMA has finished. Writing to any of the registers while a DMA is in progress will not disrupt the DMA as a local copy of each register is taken at the start of a DMA.

An interrupt can be generated whenever an output DMA transfer completes. This is enabled by bit 14, *CommandInterruptEnable*, in the **IntEnable** PCI register[18]. An interrupt can also be generated selectively by placing a **CommandInterrupt** tag into the FIFO or DMA buffer[19]. A side effect of this is that any subsequent command and register processing is also held until the output DMA finishes.

4.1 Linear DMA Transfers

A simple linear DMA transfer can be set up using the **DMAOutputAddress** register and the **DMAOutputCount** command. This is a subset of the Feedback Mode transfer (described below) which does not use packing or stride data. The output rectangle DMA function is used to transfer data from the output of the graphics processor to system memory.

The **DMAOutputAddress** register holds the address (logical or physical) where the stream of 32 bit words is to be written. The start address is given as a byte address but the lower two bits are ignored. The **DMAOutputCount** command holds the number of 32 bit words to transfer and starts the transfer if the output DMA controller is idle. The count is held as a 24 bit number.

The data to write to memory is read from the HostOut FIFO so **FilterMode** must be configured to allow the required data and/or tags through.

For linear DMA transfers the **DMARectangleWrite** register specifies the pixel size and whether to pack pixels received from the core into 32-bit words before writing the data across the PCI bus. Supported pixel sizes are 8, 16, 24 and 32 bits. The unit maps each "Rectangular" DMA message into a series of Linear DMA transfers, one for each line of the rectangle to be transferred.

---

[18] The Gamma 1 and 2 PCI **CommandMode** and **CommandStatus** registers are no longer supported.

[19] This would normally generate the interrupt as soon as the input DMA controller found it and not when the output DMA had finished. To force **CommandInterrupt** to wait for the output DMA to finish before generating the interrupt; set bit 1 (OutputDMA) to 1 (=ON).

4.2 Feedback and Select Mode DMA Transfers

Writing to the **DMAFeedback** register triggers a DMA that may be stopped before the full count (set in the DMAFeedback register) has expired if the graphics processor outputs an *EndOfFeedback* signal.

When the output DMA is used for image uploading or context dumps the amount of data to read from R5 is deterministic, however returning *Feedback* data presents the host with a problem in that it doesn't know how much feedback data is going to be generated[20]. A similar situation exists for *Select* data.

The Feedback DMA mechanism allows the collection and transfer of an unspecified amount of data from the HostOut FIFO. This can be used for OpenGL Feedback and Select modes.

A Feedback DMA transfer is set up by using the **DMAOutputAddress** register and the **DMAFeedback** command to specify the target address and number of dwords to transfer. **DMAOutputAddress** holds the start address (logical or physical) where the feedback or select data is to be written The addres1s is given as a byte address but the lower two bits are ignored. The **DMAFeedback** command with the length of the memory buffer (in words) is sent to start the Output DMA controller. Data is never written beyond the end of the given buffer length.

Once all the data to write to memory has been transferred the **EndOfFeedback** command is sent to R5. The Output DMA controller, when it reads this tag from the Host Out FIFO, terminates the DMA transfer and updates the **PCIFeedbackCount** PCI register with the number of words written into memory.

During the transfer one of two situations will be detected:

- The memory buffer will become full before the **EndOfFeedback** tag in the Host Out FIFO is detected. In this case the DMA is terminated and the host informed, however the host output FIFO will continue to be read and its contents discarded until the **EndOfFeedback** tag is found. The **EndOfFeedback** tag and its data will also be discarded.
- The **EndOfFeedback** tag is detected before the memory buffer has become full. In this case the DMA is terminated and the host informed.

The **PCIFeedbackCount** register holds the actual number of words transferred.

FilterMode is set up so that both the tag and data are entered into the FIFO - the Output DMA controller is looking for a specific tag (**EndOfFeedback**). All tags are discarded and never written into the buffer. The data for valid tags is also written into the buffer while data for invalid tags is discarded. Valid tags in this mode are:

- **FeedbackX**
- **FeedbackY**
- **FeedbackZ**
- **FeedbackW**
- **FeedbackRed**

---

[20] Feedback mode in OpenGL returns the vertex parameters rather than doing the actual rendering these vertex parameters represent. Vertices are returned after polymode, backface culling and clipping so there is no easy way for the host software to predict (without doing all the geometry work in *exactly* the same order and to the same precision as R5). For example a single triangle going in may end up producing 0, (1…13) * 3 vertices out. A clipped triangle could, in the worst case, yield a 15 sided polygon and this will be returned as 13 triangles or 13 * 3 vertices.

- **FeedbackGreen**
- **FeedbackBlue**
- **FeedbackAlpha**
- **FeedbackS**
- **FeedbackT**
- **FeedbackR**
- **FeedbackQ**
- **FeedbackToken**
- **PassThrough**
- **SelectRecord**
- **ContextData**
- **EndOfFeedback**

All these tags are part of the previously undocumented Remainder group (bits 14 and 15) in the **FilterMode** and both these bits must be set for Feedback DMA to work. The remaining filter groups do not need to be disabled as they are automatically filtered by the Output DMA controller when in feedback mode.

Note: Feedback mode can also be used for context dumps.

4.3 Rectangular DMA Transfers

The Rectangle DMA unit, part of the Vertex Unit, supports Linear and Feedback output DMA as well as rectangulat DMA images allows image data to be transferred from R5 to host memory. The image data written to memory 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.

The pixel data written to host memory is always packed, however when read from R5 it can be in packed or unpacked format.

Tag	Use
DMARectangle Write	This command initiates the image data transfer. See below for a description of the data field.
DMARectangle WriteAddress	This register provides the byte address of the first pixel in the image or sub image to write. This is treated as a logical or physical address depending on the LogicalAddressing control bit in the *CommandMode* PCI register. The address should be aligned to the natural size of the pixel, except for 24 bit pixels which may be aligned to any byte boundary.
DMARectangle WriteLinePitch	This register defines the amount, in bytes, to move from one scanline in the image to the next scanline. For a sub image this is based on width of the whole image. The pitch is held as a 32 bit 2's complement number. This is normally an integer multiple of the number of bytes in a pixel.

Table 4.1 Write Command and Registers for Rectangular DMA

DMARectangleWrite

Name	Type	Offset	Format
DMARectangleWrite	RectangleDMA *Control register*	0xA9C8	Bitfield

Bits	Name	Read	Write	Reset	Description
0-11	Width	✕	✔	x	Width of the rectangle in pixels. Range 0…4095
12-23	Height	✕	✔	x	Height of the rectangle in pixels. Range 0…4095
24-25	PixelSize	✕	✔	x	The size of the pixels in the source image to read. The pixel size is used during alignment and packing. The values are: 0 = 8 bits, 1 = 16 bits, 2 = 24 bits, 3 = 32 bits
26	Pack	✕	✔	x	1 = data is right justified and any unused bits (in the most significant end of the word) are set to zero. 0 = data read from the Host Out FIFO is packed. N.B. this is the inverse of the bit setting in **DMARectangleRead**
27..31	Reserved	0	0	x	

Notes:

1. The Rectangle DMA mechanism allows image data to be transferred from R5 to host memory. 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.
2. 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. Note that it is packed when *Reset*. It can also, optionally, be aligned on 64 byte boundaries.
3. The minimum number of PCI writes are used to align and pack the image data.
4. R5 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..
5. See also: **DMARectangleReadAddress**; **DMARectangleReadLinePitch**; **DMARectangleReadTarget**

The minimum number of PCI writes are used to align and pack the image data.

R5 is set up to rasterize the source area for the pixel data (depth, stencil, color, etc.) with the **FilterMode** set up to allow the appropriate data through (the tag should not be included). The rasterization is best set up before the Rectangular DMA is started, but as this is asynchronous it is not necessary to do things in this order.

4.3.1 Interrupts with OutputDMA

Since the **DMACount** register is decremented every time a data item is transferred from the DMA buffer this happens when the last data item is transferred from the DMA buffer:

```
case CommandInterruptTag:
          if (Data != 0)
      {
          // Assert interrupt at end of current output DMA.
          if (CheckFifoEmpty(PciWriteControlFifo))
          {
               HICommandInterrupt = True;
               Code = True;
          }
          else
               Code = False;
      }
      else
      {
          HICommandInterrupt = True;
          Code = True;
          }
          break;
```

Interrupts are enabled as discussed in section 3.2.5.2, DMA Interrupts, above[21].

4.3.2 Using Run Length Encoding

Image data frequently contains runs of the same pixel data. Run Length Encoding (RLE) is a convenient image compression method which counts adjacent identical pixel values instead of replicating them. This does not speed up the image transfer from the core's viewpoint (it still needs to read the data) but it reduces the amount of data carried over the PCI bus and, potentially, the host effort in processing/copying the image data.

When run length encoded output DMA is enabled (by setting the *RunLengthEncodeData* bit in the **FilterMode** register) then any data (except tags) which matches the 32 bit run length value is added to the run length count instead of being written to the FIFO. The accumulated run length is written to the FIFO when:

- The new 32 bit word is different from the run being encoded.
- A new scanline is started.
- The end of the primitive occurs.

The amount of data produced during the run length encoding is not known when the DMA controller is set up so an alternative mechanism is used to tell the DMA controller the upload data has finished. This is done by using the **EndOfFeedback** command. When detected in the DMA controller the DMA can be terminated as all the data has been received.

Note: *There is the potential for the software to hang if the Output DMA controller is in Feedback Mode and the* RunLengthEncodeData *bit is not set.*

If the *RLE* bit is not set during Feedback mode DMA, the **EndOfFeedback** tag is ignored by the DMA controller, hence the software will not be informed the upload has finished. The

[21] Interrupt handling is not compatible with PCI interrupts in earlier Gamma chipsets.

graphics core, etc. will continue to function as normal but the Output DMA controller will loop forever discarding data once the buffer count has expired. If the software is running with a time-out the Output DMA controller can be recovered by setting the *RunLengthEncodedData* bit in the **FilterMode** register and sending an **EndOfFeedback**.

Note: This situation only arises during Feedback Mode DMA with RLE, since in all other cases the amount of data to load is known.

Software must make allowance for the fact than run length encoding is not guaranteed to result in a smaller representation of the image and in the worst case could double the size.

Run length encoding is done 32 bits at a time irrespective of the pixel depth and is masked by the **RLEMask** register before the comparison is done. This allows bits in the data to be excluded from the test, e.g. because they are unused and have 'random' values. The

5

3D Pipeline

The 3D pipeline defines the normal sequence of operations applied to vertices and primitives and is well covered in the computer graphics literature.

We start the discussion with the Transformation Unit, by which time significant processing has already taken place in the Command, Vertex Array, Rectangle DMA and Matrix Units as discussed above.

Each of the pipeline stages will be described briefly to introduce the register(s) used to control them.

5.1 Transformation

Transformation describes the process of converting the 3-dimensional vertices and other characteristics of an object to a set of pixels positions on screen. The typical transformation process involves:

- **Modelling, viewing and projecting** transformations, such as rotation, translation, scaling and projections
- **Clipping** to remove objects outside the wondow or behind the clip plane.
- **Viewport transformation**, to map the transformed primitives to screen pixels.

The math modelling required for transformations is discussed in the *OpenGL Programming Guide* in Chapter 3, *Viewing*. Please refer to this for background information.

R5 Transformation includes:

- **Vertex Blending**. Before the incoming vertex (in the Vertex message) is transformed it can be optionally blended with a second vertex (provided in the BlendVertex message). The blending is done by the following vector equation:

  ```
  v = Vertex * VBFactor0 + BlendVertex * VBFactor1
  ```

…where *VBFactor0* and *VBFactor1* are floating point scalars loaded by messages of the same name. No restrictions are places on the scalar values but typically they are configured with alpha and 1-alpha with 0 <= alpha <= 1 to simulate linear interpolation.

- **Transforming Vertices**. The incoming vertex in the Vertex message is optionally transformed by zero, one or two matrices (4x4) under control of the *UseModelViewMatrix* and *UseModelViewProjectionMatrix* bits in the **TransformMode** register. The ModelView matrix is used to generate eye space vertex coordinates which are required by some geometry and lighting configurations. The ModelViewProjection matrix generates clip space vertex coordinates and should normally be enabled. There is no matrix stack or mechanism to concatenate matrices in this unit so this must be done by software..

5.1.1 Vertices

The most involved set of operations are done on vertex coordinates and these pass through the following coordinate systems (in order):

Object Coordinates. The ModelView matrix equation transforms the coordinates from those of the 4-dimensional model space input (o suffix) to the eye space coordinates (c suffix) in which the model is being viewed.

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = \mathbf{M} \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$$

Eye coordinates. The Projection matrix equation converts four dimensional eye space coordinates into clip coordinates (e suffix).

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = \mathbf{M} \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$$

Clip coordinates. The perspective division operation converts the clip co-ordinates (after any necessary clipping) into normalized device co-ordinates (d suffix).

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

Normalised Device Co-ordinates. This coordinate space defines object to be drawn (or the visible parts, if clipped), but the coordinates are normalized to be in the range ± 1.0.

Window Co-ordinates. The window coordinates (suffix w) are the normalized device coordinates, but now scaled to the size of the window. This scaling is done by the view port mapping defined by the S and O vectors. The x and y components of S and O are related to the window's size and the z components to the depth range.

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} s_x x_d + o_x \\ s_y y_d + o_y \\ s_z z_d + o_z \end{pmatrix}$$

R5 holds two matrices used for vertex transformation. The first matrix is the **ModelView** matrix and this is stored in the **ViewModelMatrix**[0…15] registers. The second matrix is the combined **ModelView** and **Projection** matrix (i.e. **MxP** matrices from above) and this is stored in the **ModelViewProjectionMatrix[0…15]** registers. Both these matrices are 4x4 and are laid out in the registers as follows:

$$\begin{pmatrix} M_0 & M_4 & M_8 & M_{12} \\ M_1 & M_5 & M_9 & M_{13} \\ M_2 & M_6 & M_{10} & M_{14} \\ M_3 & M_7 & M_{11} & M_{15} \end{pmatrix}$$

The numerical subscripts give the order the elements are stored in the matrix registers (i.e. M_0 is stored in **ModelViewMatrix[0],** for example) and these follow the column-major order convention. Note this is different from the convention C uses which follows the row-major order.

The viewport mapping scale and offset values are held in the **ViewPortScaleX**, **ViewPortScaleY, ViewPortScaleZ, ViewPortOffsetX, ViewPortOffsetY** and **ViewPortOffsetZ** registers.

The transformation of the vertices is controlled by bits 0 and 1 in the **TransformMode** register:

TransformMode
TransformModeAnd
TransformModeOr

Name	Type	Offset	Format
TransformMode	Transform		integer
TransformModeAnd	Transform	0xAA80	Bitfield Logic Mask
TransformModeOr	Transform	0xAA88	Bitfield Logic Mask
	Control register		*Broadcast Mode*

Bits	Name	Read	Write	Reset	Description
0	UseModelView Matrix	✓	✓	x	When set (=1) causes the incoming vertex to be multiplied by the ModelView matrix. This is only necessary if the vertex in eye space is needed for subsequent processing. A small gain in performance will be seen when this transformation is disabled. The eye space vertex is used for EyeLinear TexGen, user clipping planes, fog, lighting, or auto generation of the face normal..
1	UseModel View Projection Matrix	✓	✓	x	When set (=1) causes the incoming vertex to be multiplied by the ModelViewProjection matrix to calculate coordinates in clip space. This bit should normally be set.
2	Transform Normal				When set causes any incoming vertex normal to be multiplied by the Normal matrix. The normal transformation happens in a different unit.

3…16	Reserved				
17	BlendVertex				This bit, when set, enables the blending of vertices before transformation.

Table 5.1 TransformMode Register Vertices Control Bits

5.2 Cull

Culling is the process of rejecting polygons based on the vertex ordering (either clockwise or counter-clockwise) when projected onto the screen. This can typically reject 50% of the triangles in a scene, however the onus is on the application programmer or modeler to provide the polygons in a consistent order[22].

- Points, Lines and Triangles are clip-tested against the new view frustrum (the perspective viewing pyramid section specified by glFrustrum in OGL) and discarded if out of view. The vertices within the viewing volume are projected and mapped to a viewport as Window co-ordinates for forwarding to the Geometry unit.
- Points and lines inside the viewing volume are stripe filtered against the stripes allocated to the R5. If visible in one or more stripes the primitive is passed on with the Port Mask field suitably set.
- Triangles are backface tested before any other processing.takes place

R5 provides two methods of doing culling:

- The first method, as done by OpenGL, is to test the sign of the projected area of the polygon.
- The second method is to use a user provided face normal, as done by QuickDraw3D. The sign of the z component of the transformed face normal (assuming its transformation is enabled) is used to determine the orientation. If no face normal is provided then the area based method will be used (until a face normal is provided). Normalization of the face normal is not necessary if the face normal is just used for culling.

These two methods are mutually exclusive so only one form of culling is applied at a time. The orientation of a face (i.e. front facing or not) is still determined even if culling is disabled as this also selects which Polygon Mode to apply and which material to calculate the lighting with for two sided lighting.

Culling is largely controlled by the GeometryMode register (see *Reference Guide* Volume III).

5.3 Geometry

The Geometry Unit controls clip testing and (if necessary) clipping, perspective division, viewport mapping, polymode processing for triangles etc. for all primitives which have not been culled. The Geometry Unit also:

- Applies the *PolygonOffset* when enabled.

[22] A triangle strip, for example, reverses the order for successive triangles however this is automatically allowed for when culling.

- Monitors parameter storage in the Delta Unit so that only changed parameters are ever sent[23].
- Controls the Texture Fog Unit, Normalisation Unit, Lighting Unit and Material Unit so they only process valid vertices. The Material Unit is also instructed to generate colour information for intermediate vertices produced when a primitive is clipped. The Texture Fog Unit is also instructed to generate texture and fog information for intermediate vertices produced when a primitive is clipped.
- The OpenGL Feedback and Select modes are also supported so that Picking and 3D selection are also accelerated.

In R5, Texture and Fog processing have been moved to a new Texture/Fog unit while Face Normalization and Culling are exclusively handled by the new Cull Unit (above).

The unit is controlled by the **GeometryMode** register and it controls which vertices are processed in the downstream units (Texture Fog, Normalisation, Lighting and Material).

The *RenderMode* specified in the **GeometryMode** register selects one of three major modes: Rendering, Select and Feedback. These are also discussed in this section.

5.3.1 Clipping

Clipping is done before any lighting, fog or texture calculations are done so that the (expensive) lighting is only done for vertices which are guaranteed to be used for rendering. Traditional clipping is done once all the parameters (texture, colour, fog, etc.) have been evaluated for the primitive, however deferring the lighting (and texture, fog, etc. calculations) until after clipping prevents these calculations being done if the primitive is subsequently shown to be out of view[24]. Another saving is that during the clipping process many intermediate vertices are generated and the interpolated parameters do not need to be calculated for these intermediate points. Only the final intersection points once clipping is done have the full set of parameters calculated for them.

All vertices (original, intermediate and final intersection points) are defined using barycentric coordinates25. The value of a parameter at the intersection point is given by the linear combination of that parameter at each of the three vertices. The weighting given to a parameter at a vertex is given by the associated barycentric component. If the barycentric coordinate for the intersection point is (a, b, c) and the parameter at vertex a is Pa, at vertex b is Pb, etc. then the parameter at the intersection point is given by:

$$\mathbf{P}_i = a\mathbf{P}_a + b\mathbf{P}_b + c\mathbf{P}_c$$

For line primitives the same method is used but now c = 0. No attempt is made to optimise the equations in this case to factor out the unused term.

The clipping is done against the 3d viewing frustum and up to 6 user defined clipping planes. The viewing frustum is by the by the planes:

left	x = -w
right	x = +w
top	y = +w

[23] This is not so important when the Delta Unit in Gamma is used but conserves bus bandwidth when the parameters are sent to another chip across the PCI bus.

[24] Recall that the clip test may show a primitive is partially in view, but in fact it not be in view at all.

[25] The barycentric coordinate for vertex *a* is (1, 0, 0), for vertex *b* (0, 1, 0) and vertex *c* is (0, 0, 1). No attempt is made to special case when an original vertex is referenced (to save the linear combination calculation) to simplify the behavioural model. This will result in slightly lower performance but as clipping is not a performance critical operation this is acceptable.

bottom	y = -w
near	z = +w
far	z = -w

where w is the forth component of the vertex.

The user clipping planes are defined in eye space by the 4 coefficients (per clip plane) UserClipX, UserClipY, UserClipZ and UserClipW which are synonymous with the a, b, c, and d parameters respectively in usual plane equation formula. A point is in view, if after substitution into the user clip plane equation, the result is positive.

The result of comparing a vertex against the 6 frustum planes and the 6 user clipping planes is held in a 12 bit outcode variable. If the vertex is visible with respect to one of the planes then the corresponding bit is zero.

The algorithm used is the Sutherland and Hodgman Polygon clipper. This clipper clips the whole polygon against each clipping plane in turn. The new output polygon is submitted to the next clipping plane and so on. This clipping process is documented in the paper "Reenterant Polygon Clipping by I. Sutherland and G. Hodgman" and summarised in any good Computer Graphics text, however a brief description is included here for completeness. An extract from the paper:

The clipping algorithm for a polygon against a single clipping plane considers the input vertices P1, P2, ... Pn one at a time. For each such input vertex zero, one or two of the output vertices Q1, Q2, ... Qm will be generated, depending on the input with respect to the clipping plane. Each input vertex, P, except the first is considered to be the terminal vertex of an edge: namely the edge defined by the input vertex value, P, and the position, S, of the just previous input vertex saved internally in a register. The algorithm will generate output vertices from the set Q1, Q2, ... Qm depending on the relationship between the input vertex, P, the saved vertex, S, and the clipping plane. The four basic relationships between the plane and the edge SP are:

S and P are both visible	Output P.
S and P are both invisible	Output nothing.
S is visible and P is not	Compute the intersection of SP with the plane and output.
P is visible and S is not	Compute the intersection of SP with the plane and output. Also output P.

The beginning and end of the polygon need to be handled as a special case.

The simplest was to clip the polygon against multiple planes is to clip against the first plane as outlined above. This results in a new polygon which is clipped against the next plane and so on until all the planes have been used.

This paper also describes a very elegant method of implementing this algorithm recursively, but this has not been used as it complicated the hardware implementation, it is considerably more difficult to understand and offers no real advantages in terms of performance (which isn't critical here), or storage.

Each vertex is defined by the four values: x, y, z and w and is visible only if all the following conditions are met:

(left)	$-w \le x \le w$	(right)
(bottom)	$-w \le y \le w$	(top)
(near)	$-w \le z \le w$	(far)

Each one of these 6 conditions is represented by a bit in an outcode word. If a condition is true then the corresponding bit is 0.

If the user clipping plane is defined by the four values: a, b, c and d then the vertex is visible if:

$$ax + by + cz + d \geq 0$$

If this condition is true for a given user clip plane then the corresponding bit in the outcode word is 0.

When an intersection point is to be calculated the barycentric coordinates are found rather than the xyzw, rgba, f, stq, ks and kd values as they take up less storage. For intersection points which are never used there is a computation saving, but intersection points which are used need extra computations to yield the parameter values at the intersection.

The input and output clipped polygons are stored in a circular buffer rather than two separate buffers which are ping-ponged between as this takes up much less storage. Each clip plane can add at most one extra vertex so with 12 clip planes and 4 vertices[26] we need enough storage for a maximum of 16 vertices (or 32 if two separate buffers were to be used).

A circular buffer works because once an entry has been processed it is not needed again and the entries are processed sequentially.

Short lines and small triangles can optionally avoid being clipped providing they don't cross the near or far frustum clipping planes or any of the user clip planes. In this case the whole line or triangle is rasterised in full and the screen and window scissor is used to discard out of view fragments. This can be more efficient than doing a full clip. The threshold at which clipping occurs for lines and triangles is user defined.

5.3.1.1 Viewport Mapping

The viewport mapping takes the 4D homogeneous vertex coordinates and converts them into a 3D coordinate by dividing through by the homogenous component 'w'. The resulting coordinate is often called the Normalised Device Coordinate (or NDC)[27].

The NDC is converted to the device coordinate by multiplying by the ViewportScale vector and then adding the ViewportOffset vector. The X and Y values are scaled so the true NDC range (±1.0) fills the window (or viewport) while the Z value is nominally scaled to be in the range 0.0…1.0, although the OpenGL *glDepthRange* function can be used to change this.

It is preferable to bias the window coordinates (x, y) into a range where the number of fraction bits are constant from the minimal value in the range to the largest value in the range[28]. This can be done by adjusting the *ViewportOffset*, however this bias will need to be removed in the Delta Unit when the actual rasteriser coordinates are calculated. Also as this is an internal adjustment done for numerical reasons it is not visible to the user so any coordinate information returned during feedback mode will need to have any bias removed

5.3.1.2 Select Mode

The select mode disables rendering and keep tracks of the minimum and maximum Z value of all vertices which passes the clipping and backface testing processes. A hit flag is set when such a vertex if found.

---

[26] The first vertex is repeated so the closing edge does not need to be treated as a special case.

[27] This is a slight misnomer in this case because this is prior to clipping so the NDC may extend beyond the ±1.0 range usually associated with normalised numbers.

[28] This ensures that calculation based on the window coordinates such as the triangles area are position independent with a window and not susceptible to errors arising from fewer fraction bits being available in the top right corner than in the bottom left corner.

5.3.1.3 Feedback Mode

The feedback mode provides a mechanism where the window coordinates, texture coordinates and colour values for primitives (or their clipped versions) can be read out. No rendering is done during feedback mode. See Programming Notes for more details.

5.3.1.4 Polygon Offset

The offset applied to each vertex, when suitably enabled is given by:

$$\text{offset} = m \times \text{PolygonOffsetFactor} + \text{PolygonOffsetBias}$$

where

PolygonOffsetFactor and PolygonOffsetBias are user supplied values

$$m = \max\left\{\left|\frac{\partial z}{\partial x}\right|, \left|\frac{\partial z}{\partial y}\right|\right\}$$

The Z value is clamped[29] to be in the range 0.0…1.0 after the offset is added.

5.4 Normalization

The Normalization Unit transforms and normalizes vertex normals. Vertex normals are only transformed (if enabled) and normalised (if enabled) when a vertex has passed the clipping and backface tests in the Geometry Unit. This unit is controlled by the Geometry Unit using the **VertexColour** register. When this message arrives it ensures the correct normals are present in the Lighting Unit and Texture Fog Unit before any colours are calculated.

This unit holds the normal, but before it is updated from the message stream any outstanding normals will be transformed first.

This unit is also responsible for monitoring any material or lighting related tags so that if we are in the middle of primitive processing then the colour and texture[30] can be calculated before a material property (for example) is changed. This is only necessary because the colour calculations are deferred until as late as possible. The subsequent units are forced to calculate any outstanding colours via the ValidateVertexColours message.

The inverse length of antialiased lines is also calculated. This functionality logically belongs in the Delta Unit, however moving it to this unit is pragmatic: it increases performance, reduces gate count, uses resources which would otherwise be idle and simplifies the Delta Unit. The Delta Unit tags are monitored and the XY coordinate information interscepted and stored in the vertex store.

Before the incoming vertex normal (in the Normal message) is transformed and normalised it can be optionally blended with a second normal (provided in the BlendNormal message). The blending is done by the following vector equation:

```
v = Normal * NBFactor0 + BlendNormal * NBFactor1
```

[29] This is done in the Delta Unit.

[30] The normal is only used if the SphereMap TexGen mode is used.

where NBFactor0 and NBFactor1 are floating point scalars loaded by messages of the same name. No restrictions are places on the scalar values but typically they are configured with alpha and 1-alpha with 0 <= alpha <= 1 to simulate linear interpolation.

5.5 Texture/Fog

The incoming texture coordinates, or the texture coordinates generated under TexGen control (see later) are transformed by the 4x4 Texture matrix held in the **TextureMatrix[0…15]** registers. This matrix is laid out in the registers as follows:

$$\begin{pmatrix} T_0 & T_4 & T_8 & T_{12} \\ T_1 & T_5 & T_9 & T_{13} \\ T_2 & T_6 & T_{10} & T_{14} \\ T_3 & T_7 & T_{11} & T_{15} \end{pmatrix}$$

The numerical subscripts give the order the elements are stored in the matrix registers (i.e. T_0 is stored in **TextureMatrix[0],** for example) and these follow the column-major order convention. Note this is different from the convention C uses which follows the row-major order.

The transformation of the texture is controlled by one bit in the **TransformMode** register see **Error! Reference source not found.**.

5.6 Lighting

R5 implements the full OpenGL RGB lighting model. The Color Index lighting model is not supported. The ambient, diffuse and specular lighting components are given by the following vector equations (vectors in bold). The subscripts used are

c	color
p	position
m	material
l	light
i	light number

$$\mathbf{lightAmbient} = \sum_{i=0}^{n-1} (atti_i)(spot_i)\mathbf{a}_{cli}$$

$$\mathbf{lightDiffuse} = \sum_{i=0}^{n-1} (atti_i)(spot_i)\left(\mathbf{n} \bullet \vec{\mathbf{VP}}_{pli}\right) d_{cli}$$

$$\mathbf{lightSpecular} = \sum_{i=0}^{n-1} (atti_i)(spot_i)\left[(f_i)(\mathbf{n} \bullet \hat{\mathbf{h}}_i)^{s_{cm}} \mathbf{s}_{cli}\right]$$

where:

$$atti_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i}\left\|\mathbf{VP}_{pli}\right\| + k_{2i}\left\|\mathbf{VP}_{pli}\right\|^2}, & \text{if mode.Attenuation is true,} \\ 1.0, & \text{otherwise.} \end{cases}$$

where k_{0i}, k_{1i} and k_{2i} are the attenuation coefficients for light *i*.

$$spot_i = \begin{cases} \left(\vec{\mathbf{VP}}_{pli} \bullet \hat{s}_{dli}\right)^{s_{rli}}, & \text{mode.Spotlight is true, } \vec{\mathbf{VP}}_{pli} \bullet \hat{s}_{dli} \geq \cos\left(c_{rli}\right), \\ 0.0, & \text{mode.Spotlight is true, } \vec{\mathbf{VP}}_{pli} \bullet \hat{s}_{dli} < \cos\left(c_{rli}\right), \\ 1.0, & \text{mode.Spotlight is false.} \end{cases}$$

where c_{rli} is the spotlight cutoff angle for light *i*.

$$f_i = \begin{cases} 1, & \mathbf{n} \bullet \vec{\mathbf{VP}}_{pli} \neq 0, \\ 0, & \text{otherwise.} \end{cases}$$

$$\mathbf{h}_i = \begin{cases} \vec{\mathbf{VP}}_{pli} + \vec{\mathbf{VP}}_e, & \text{local viewer true} \\ \vec{\mathbf{VP}}_{pli} + \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T, & \text{local viewer false} \end{cases}$$

lights (numbered 0 through 15). The registers for light 0 are shown in Table 5.2. For other lights substitute the appropriate number for 0. Note the successive light parameters follow

$$\mathbf{c} = \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \mathbf{ambientLight} * \mathbf{a}_{cm} + \mathbf{diffuseLight} * \mathbf{d}_{cm} + \mathbf{specularLight} * \mathbf{s}_{cm}$$

$$\mathbf{diffuseTexture} = \mathbf{e}_{cm} + \mathbf{a}_{cm} + \mathbf{ambientLight} + \mathbf{diffuseLight}$$

$$\mathbf{specularTexture} = \mathbf{specularLight}$$

in sequential registers so it is easy to generate the tag number or register address algorithmically rather than always using symbolic tokens.

Register	Equation symbol	Offset from LightMode	Description
Light0Mode		0	Mode control for Light (see later).
Light0AmbientIntensityRed	a_{cli}	1	Ambient red intensity in floating point format.
Light0AmbientIntensityGreen		2	Ambient green intensity in floating point format.
Light0AmbientIntensityBlue		3	Ambient blue intensity in floating point format.
Light0DiffuseIntensityRed	d_{cli}	4	Diffuse red intensity in floating point format.
Light0DiffuseIntensityGreen		5	Diffuse green intensity in floating point format.
Light0DiffuseIntensityBlue		6	Diffuse blue intensity in floating point format.
Light0SpecularIntensityRed	s_{cli}	7	Specular red intensity in floating point format.
Light0SpecularIntensityGreen		8	Specular green intensity in floating point format.
Light0SpecularIntensityBlue		9	Specular blue intensity in floating point format.
Light0PositionX	P_{pli}	10	X position of the light if PositionW = 0, otherwise it is the normalized X direction.
Light0PositionY		11	Y position of the light if PositionW = 0, otherwise it is the normalized direction.
Light0PositionZ		12	Z position of the light if PositionW = 0, otherwise it is the normalized Z direction.
Light0PositionW		13	W position of the light. When zero, it changes the meaning of the Position* values to be directions.
Light0SpotlightDirectionX	s_{dli}	14	Normalized X component for the spotlight direction or the normalized X component of the half vector when the light is not a spotlight.
Light0SpotlightDirectionY		15	Normalized Y component for the spotlight direction or the normalized Y component of the half vector when the light is not a spotlight.
Light0SpotlightDirectionZ		16	Normalized Z component for the spotlight direction or the normalized Z component of the half vector when the light is not a spotlight.
Light0SpotlightExponent	srli	17	Spotlight exponent. This is held as an unsigned 7.4 fixed point number.
Light0CosSpotlightCutoffAngle	c_{rli}	18	Cosine of the spotlight cut-off angle. Its range is 0.0 to 1.0 inclusive.
Light0ConstantAttenuation	k_{0i}	19	Constant attenuation factor.
Light0LinearAttenuation	k_{1i}	20	Linear attenuation factor.
Light0QuadraticAttenuation	k_{2i}	21	Quadratic attenuation factor.

Table 5.2 Light 0 Registers

Each light has a mode word associated with it and the definition of its control bits are shown in Table 5.3 (these flags are used to optimize the calculations):

Bit No.	Name	Description
0	LightOn	When set indicates the light is on and contributes illumination to the scene, otherwise it does not.
1	Spotlight	When set indicates the light is a spotlight. If it is not set then the light is not a spot light and the SpotlightDirection is used to hold the normalized half vector between the viewer and the light.
2	Attenuation	When set indicates the light is to be attenuated, otherwise no attenuation is done.
3	LocalLight	When set indicates the light is local and the full lighting equations should be used. This allows a light to be local without it having to be a spotlight or have any attenuation applied to it.

Table 5.3 Light Mode Word Control Bits Definitions

Most of the lighting parameters should be self explanatory, however there are a few which will be clarified:

- The light's position and direction vectors are not transformed automatically when loaded into R5 and any transformation required to locate them in eye space (the coordinate system where lighting is done, i.e. the view vector is always (0, 0, 1)) should be done in software before they are loaded.
- All direction vectors (i.e. Position vector when the W component is zero, SpotlightDirection and the half vector) must all be normalized.
- The Light mode bits should be set up as necessary. Any special values assumed by OpenGL (for example the spotlight cut-off angle of 180˚) are ignored. The preferable state of each bit is zero to avoid some or all of the lighting calculation and the bits are generally set up as shown in Table 5.4 (to follow OpenGL conventions for identifying some light mode or operation that is not required).

Bit No.	Name	Description
1	Spotlight	Set when the spotlight cut off angle is not 180 degrees.
2	Attenuation	Set when the light's position has a non zero W (this indicates the position holds a position and not a direction) and the sum of the attenuation factors is not unity are not 1, 0, 0 for constant, linear and quadratic respectively.
3	LocalLight	Set when the light's position has a non zero W (this indicates the position holds a position and not a direction).

Table 5.4 Lighting Calculation Bit Set up

- When the Light mode indicates the light is not a spotlight the SpotlightDirection vector used to hold the normalized half vector. The half vector between the light's position vector and the eye vector is given by the following equation:

$$\mathbf{h} = \hat{\mathbf{P}}_{pli} + (0 \quad 0 \quad 1)$$

- The nominal range for the light's intensity colors (and the material colors) is 0.0…1.0 inclusive. Values outside this range (and even negative ones) can be used and the final vertex colors are clamped (if enabled) to be in the range 0.0…1.0 during the primitive's set up calculations prior to rasterization.

The material parameters are held in the registers shown in Table 5.5. There are two identical sets to hold the front and back materials used during two sided lighting.

Register	Equation symbol
FrontEmissiveColorRed	e_{cm}
FrontEmissiveColorGreen	
FrontEmissiveColorBlue	
FrontAmbientColorRed	a_{cm}
FrontAmbientColorGreen	
FrontAmbientColorBlue	
FrontDiffuseColorRed	d_{cm}
FrontDiffuseColorGreen	
FrontDiffuseColorBlue	
FrontAlpha	
FrontSpecularColorRed	s_{cm}
FrontSpecularColorGreen	
FrontSpecularColorBlue	
FrontSpecularExponent	s_{rm}
BackEmissiveColorRed	e_{cm}
BackEmissiveColorGreen	
BackEmissiveColorBlue	
BackAmbientColorRed	a_{cm}
BackAmbientColorGreen	
BackAmbientColorBlue	
BackDiffuseColorRed	d_{cm}
BackDiffuseColorGreen	
BackDiffuseColorBlue	
BackAlpha	
BackSpecularColorRed	s_{cm}
BackSpecularColorGreen	
BackSpecularColorBlue	
BackSpecularExponent	s_{rm}
SceneAmbientColorRed	a_{cs}
SceneAmbientColorGreen	
SceneAmbientColorBlue	

Table 5.5 Material Parameters Registers

The OpenGL specification associates an alpha value with each lighting term, however only the diffuse alpha is used. The diffuse alpha value is held in the **FrontAlpha** or **BackAlpha** registers.

The lighting calculations are very expensive and can easily dominate the overall performance so it is very worth while to check the light's parameters to see if a simpler form of the lighting equation can be used (by setting the appropriate bits in the LightMode register) For example if the Constant Attenuation is set to 1.0 and both the Linear and Quadratic Attenuation factors are set to 0.0 then the attenuation part of the calculation can be avoided.

In a similar line if the product of the attenuation and spot values for a light falls below the value given in the AttenuationCutOff register then (if suitably enabled) the calculations for this light will be automatically terminated. This optimization allows lights which are becoming too faint to contribute to be terminated early. A suitable value is given by: $1.0/512_n$ where n is the number of lights. The 512 constant was chosen as it is less than the smallest representable color when converted to a byte integer assuming the light and material colors are restricted to the range 0.0…1.0.

The lighting calculations are controlled by the LightingMode and the MaterialMode register. Refer to the *R5 Reference Guide* volume III for register field descriptions.

Bit No.	Name	Description
0	Enable	When set causes the vertex to be lit using the lighting equations, otherwise the current color is assigned.
1, 2	TwoSidedLighting	The three options are: 0 Use the front side material. 1 Use the back side material and invert the normal before it is used in the lighting calculations. 2 Use the orientation of the face to select between front or bacl materials and lighting. The orientation is determined by fielc in the **GeometryMode** register.
3	LocalViewer	When set causes the viewer to be at (0, 0, 1) in eye coordinates, otherwise the viewer is at (0, 0, ∞). When the viewer is at infin some of the lighting equations can be simplified so run faster, however the position of the specular highlights are not as correc
4	FlipNormal	When set causes the absolute value of the lighting dot products t be taken, otherwise negative dot products are clamped to zero. Clamping is used for OpenGL, but some other APIs allow the normal to be flipped - this gives a cheap form of two sided light and is useful when the normals are not consistently facing 'outwards' in the model or scene.
5	AttenuationTest	When set forces the lighting calculation for the current light to b aborted when the product of *atti* and *spot* (in the lighting equatio for the light falls below the threshold given in the **AttenuationCutOff** register.
6…14	NumberLights	This 9 bit field holds the number of lights to use. Its legal range 0…16 inclusive. Numbers greater than 16 are clamped to be 16
15	SpecularLightingEnable	When this bit is set the specular part of the lighting calculations done, otherwise they are skipped. This bit would normally be s for OpenGL, however some APIs allow non specular lighting models to be used.
16	Reserved	

Table 5.6 LightingMode Register Fields

Bit No.	Name	Description
0	Enable	When set causes the vertex to be calculated from the lighting equations otherwise the current color is assigned.
1	DiffuseTextureEnable	When set allows the diffuse texture color to be calculated and sent to GLINT. This is further qualified by the TextureEnable bit in the **Begin** command so is only done when texture mapping is enabled.
2	SpecularTextureEnable	When set allows the specular texture color to be calculated and sent to GLINT. This is further qualified by the TextureEnable bit in the **Begin** command so is only done when texture mapping is enabled.
3	MonochromeDiffuseTexture	When set the diffuse texture color is converted to a monochrome value before it is sent to GLINT. This allows the diffuse texture DDA in GLINT 500TX to be set up. When clear the true color value is sent and is used when the target rendering chip is GLINT MX as it has true color diffuse texture DDAs.
4	MonochromeSpecularTexture	When set the specular texture color is converted to a monochrome value before it is sent to GLINT. This allows the specular texture DDA in GLINT 500TX to be set up. When clear the true color value is sent and is used when the target rendering chip is GLINT MX as it has true color specular texture DDAs.
5	PremultiplyAlpha	When set premultiplies the diffuse and ambient colors by the material alpha value.
6	ColorSource	This field selects where the color should be taken from when the Enable field is 0. The options are: 0: Current color value. 1: Diffuse material value.
7, 8	TwoSidedLighting	The three options are: 0 Use the front side materials. 1 Use the back side materials. 2 Use the orientation of the face to select between front or back materials and lighting.

Table 5.7 MaterialMode Register Fields

The selection between Gouraud and flat shading is controlled by the FlatShading bit in GeometryMode. The SmoothShadingEnable bit in the DeltaMode register is ignored.

5.7 Material

5.8 Primitive Set-Up

The general primitive set up operation is controlled by the DeltaMode register. Where additional control is needed for individual primitive types this can be found in the PointMode, LineMode, TriangleMode and RectangleMode registers.

The DeltaMode register fields are shown in Table 5.8 (it is identical to the DeltaMode register in GLINT Delta, but with the addition of BiasCoordinates, ColorDiffuse, ColorSpecular and FlatShadingMethod fields).

Bit No.	Name	Description
0, 1	TargetChip	This two bit field selects which chip the calculations are tailored to. The options are: 0 = 300SX 1 = 500TX, MX
2, 3	DepthFormat	This two bit field defines the depth format GLINT is working in and hence the final form of the depth parameters to be written into GLINT. The options are: 1 = 16 bits (300SX, 500TX, MX) 2 = 24 bits (300SX, 500TX, MX) 3 = 32 bits (300SX, 500TX, MX) The depth format is used regardless of any other modes bits.
4	FogEnable	When set enables the fog calculations. This is qualified by the FogEnable bit in the **Begi** or **Draw*** commands.
5	TextureEnable	When set enables the texture calculations. This is qualified by the TextureEnable bit in the **Begin** or **Draw*** commands.
6	SmoothShadingEnable	When set enables the color calculations. This field only has an effect when the **Draw*** commands are used and is ignored when the R5 3D pipeline is used. In this case the FlatShading bit in the **GeometryMode** register is used.
7	DepthEnable	When set enables the depth calculations.
8	SpecularEnable	When set enables the specular texture calculations. This is qualified by the TextureEnabl in the **Begin** or **Draw*** commands.
9	DiffuseEnable	When set enables the diffuse texture calculations. This is qualified by the TextureEnable in the **Begin** or **Draw*** commands.
10	SubPixelCorrectionEnable	When set provides the subpixel correction in Y. This is qualified by the SubPixelCorrectionEnable in the **Begin** or **Draw*** commands.
11	DiamondExit	When set enables the application of the OpenGL 'Diamond-exit' rule to modify the start and end coordinates of lines.
12	NoDraw	When set prevents any rendering from starting after the set up calculations are done and parameters sent to GLINT. This only effect the **Draw*** commands and is ignored when tl R5 3D pipeline is used.
13	ClampEnable	When set causes the input values to be clamped into a parameter specific range. Note tha the texture parameters are not included. This should normally be set.
14, 15	TextureParameterMode	This two bit field causes the texture parameters to be: 0 = Used as given 1 = Clamped to lie in the range -1.0 to 1.0 2 = Normalized to lie in the range -1.0 to 1.0 The normal setting for this field is to select texture normalization.
16…18	reserved	
19	BiasCoordinates	When set causes the **XBias** and **YBias** registers values to be added to the x and y coordinates respectively.
20	ColorDiffuse	When set causes the diffuse texture calculations to be done on the red, green and blue components, otherwise the red component (representing monochrome) is done by itself.
21	ColorSpecular	When set causes specular texture calculations on red, green and blue components, otherwise the red component (representing monochrome) is done by itself.

22	FlatShadingMethod	This field determines how the ColorDDA unit in GLINT is to do flat shading. The t options are use the **ConstantColor** register (0) or the DDA (1) by setting zero gradie The rasterization performance is the same in both cases, however the ConstantColor method is faster to set up. Consider the situation when smooth shading is enabled (i **GeometryMode** register) and a point is to be drawn. The point is always flat shaded field would normally be the inverse of the FlatShading field in the **GeometryMode** register.

Table 5.8 DeltaMode Register Fields

How a primitive gets rendered depends on the primitive type given with the **Begin** command and the PolyMode setting (for triangles, quads and polygons) in the **GeometryMode** register.

Antialiasing is controlled by the AntialiasEnable and AntialiasingQuality bits in the individual point, line and triangle mode registers. Antialiasing will only occur for a particular primitive if it is enabled in the corresponding mode register and the AntialiasEnable bit in the **Begin** command is also set.

The x and y coordinates for point, line, triangle and 3D rectangle can optionally have a bias added to them before rendering. This is controlled by the BiasCoordinates bit in the **DeltaMode** register and the biases are held in the **XBias** and **YBias** registers. This biasing facility has several uses:

- The coordinates may be window relative (almost guaranteed with 3D graphics) but the rasterizer may be set up to do screen relative rendering. Adding a bias value will do the conversion.
- The view port mapping may be set up to add a bias to remove any differences in the accuracy of the set up calculations as a function of the primitive's position on the screen. This bias needs to be removed before rendering is actually done.

More consistent (or position independent) set up calculations can be achieved by biasing the x and y coordinates coming out of the view port mapping. This ensures that all the calculations on the x and y coordinates all have the same degree of precision irrespective of their location on the screen. Consider a floating point value of 1.0 - this will have 23 bits of fractional precision whereas a value of 1024.0 will only have 14 bits of fractional precision. Biasing the values by 8K (for example) forces both cases to have 11 bits of fractional precision and hence yield the same rasterized pixels for a given triangle anywhere on the screen.

Note: This bias will need to be removed from any coordinate information returned during Feedback mode or by reading back the Raster Position because the application will not be expecting biased coordinates to be returned.

5.8.1 Points

The **PointMode**, **PointSize** and **AAPointSize** registers control how points are drawn.

5.8.1.1 Aliased Points

For aliased points the **PointSize** register holds the desired point size. The range of actual integral point sizes are 1…255 held in the bottom 8 bits; a 0 point size is treated as a point size of 1. Points are drawn according to the OpenGL rules so wide points are drawn as squares centered on the vertex. Any parameters (such as color, depth, etc.) are held constant for each fragment rasterized as part of the point.

5.8.1.2 Antialiased Points

Antialiased points have their width, as a floating point number, defined by the **AAPointSize** register. In theory any size antialiased points can be defined, however GLINT places some restrictions on what these widths can be. The Point Table in GLINT restricts the diameter of antialiased points to be from 0.5 to 16.0 in steps of 0.25 when the antialiasing quality is 4x4 or 0.25 to 8.0 in steps of 0.125 for 8x8 quality. R5 does not set up the Point Table. Points with a zero size will draw a single fragment and points with a negative size will draw a point of the same positive size.

It is the user's responsibility to have set up any alpha blending modes. The style of rendering used for a point is determined by the following registers:

The antialiasing quality is held in the **PointMode** register.

Bit No.	Name	Description
0	AntialiasEnable	This field, when set, enables antialiasing of points. This is qualified by the AntialiasEnable field in the **Begin** command. Note the Point Table in GLINT must be set up for the corresponding point size (held in **AAPointSize** register) and the selected antialiasing quality (next field).
1	AntialiasingQuality	This field defines the quality of antialiased points: 0 = 4x4 1 = 8x8 The Point Table in GLINT must be set up appropriately for the quality and the **AAPointSize.**

Table 5.9 Antialiasing in the PointMode Register

5.8.2 Lines

The **LineMode**, **LineWidth**, **LineWidthOffset** and **AALineWidth** registers control how lines are drawn.

5.8.2.1 Aliased Lines

For aliased lines the **LineWidth** register holds the desired line width. The range of actual integral line widths are 1…255 held in the bottom 8 bits; a 0 line width is treated as a line width of 1. Lines are drawn according to the OpenGL rules so wide lines are drawn as a sequence of lines offset in X or Y depending on whether the line is X major or Y major. The **LineWidthOffset** register is normally set to (line width - 1) / 2. For one pixel wide lines the **LineWidthOffset** is set to 0.

If line stipples are enabled (in the **LineMode** register) then wide aliased lines will be stippled correctly by repeating the line (offset in X or Y), but with the stipple position re-established for each line used to make up the width.

5.8.2.2 Antialiased Lines

Antialiased lines have their width defined as a floating point number by the **AALineWidth** register so can take any width. Lines with a zero width will not be drawn. Lines with a negative width will draw a line of the same positive width. An antialiased line is drawn as rectangle aligned to the direction of the line. This is decomposed into trapezoids so no retained alpha buffer is needed (which would be the case if the line were decomposed into two triangles).

It is the user's responsibility to have set up any alpha blending modes.

If line stipples are enabled (in the **LineMode** register) then antialiased lines will be stippled using the RepeatFactor, StippleMask and Mirror fields in the **LineMode** register. These fields would normally track the fields of the same name in the GLINT **LineStippleMode** register. The stipple pattern is converted into a series of short lines which are drawn as antialiased lines.

The line stipple hardware in GLINT is not used and does not get updated for stippled antialiased lines, however this does not cause a problem because OpenGL does not allow switching between aliased and antialiased lines between polyline segments and the stipple pattern is always reset on a **Begin** command

Bit No.	Name	Description
0	StippleEnable	This field, when set, enables the stippling of lines. It only effects wide lines or antialiased lines. This will normally b the same value as the Enable field in the **LineStippleMode** GLINT register.
1…9	RepeatFactor	This 9 bit field holds the repeat factor for antialiased stipple lines. This will normally be the same value as the RepeatFactor field in the **LineStippleMode** GLINT register The repeat factor stored here is one less than the desired repeat factor.
10… 25	StippleMask	This 16 bit field holds the stipple pattern to use for antialiased lines. This will normally be the same value as tl StippleMask field in the **LineStippleMode** GLINT register.
26	Mirror	This field, when set, will mirror the StippleMask before it i used for antialiased lines. This will normally be the same value as the Mirror field in the **LineStippleMode** GLINT register.
27	AntialiasEnable	This field, when set, enables antialiasing of lines. This is qualified by the AntialiasEnable field in the **Begin** comman
28	AntialiasingQuality	This field defines the quality of antialiased lines: 0 = 4x4 1 = 8x8

Table 5.10. LineMode Register Fields

5.8.3 Polygons

Triangles, quads and polygons are controlled by the **TriangleMode** register.

Bit No.	Name	Description
0	AntialiasEnable	This field, when set, enables antialiasing of triangles. This is qualified by the AntialiasEnable field in the **Begin** command.
1	AntialiasingQuality	This field defines the quality of antialiased triangles: 0 = 4x4 1 = 8x8
2	UseTrianglePacketInterface	This field, when set, causes the triangle set up to use the Triangle Packet Interface to send the triangle parameters to GLINT. This is only supported in GLINT MX and provides a higher triangle throughput.

Table 5.11 TriangleMode Register Fields

5.8.4 3D Rectangle

OpenGL rectangles are positioned and given depth, color, texture, etc. parameters using the glRasterPos function (which translates into writes to the **RPx3**, **RPy**, etc. registers). The width and height of the rectangle is held in the **RectangleWidth** and **RectangleHeight** registers as floating point numbers and the **RectangleMode** register holds data to pass to GLINT in the **Render** command. When the **RPx2**, **RPx3** or **RPx4** registers are written to the coordinate is transformed, clipped, colored and textured as required by OpenGL and the results saved to be used by the **GeomRectangle** command.

Bit No.	Name	Description
0, 1	Type	These two bits define the type of rectangle to be inserted into the feedback buffer. They have no effect when not in feedback mode. The options are: 0 = Bitmap 1 = DrawPixel 2 = CopyPixel 3 = Don't insert into the feedback buffer.
2	OffsetEnable	When this bit is set the x and y offset values held in **RasterPosXOffset** and **RasterPosYOffset** registers respectively displace the raster position window coordinates when the rectangle is rendered. This does not update the raster position state.
3	SelectEnable	When this bit is set the rectangle takes part in the selection process.

Table 5.12 GeomRectangle Control Fields

If the rectangle passes the clip test then it is rendered with the above parameters. The fill direction is always bottom to top (i.e. increasing Y), left to right so any download data or bitmask *must* be provided in this order. If the rectangle is clipped out (this is an all or nothing test, unlike lines, triangles or quads which do geometric clipping of the primitive) then any following image data or bitmasks are automatically discarded.

The floating point **RasterPosXOffset** and **RasterPosYOffset** registers are temporarily added to the current raster position X and Y coordinates respectively before the rectangle is set up in the rasterizer. The original raster position coordinates are not updated. This temporary offset, measured in pixels, is useful when an OpenGL primitive requires several passes or is processed in strips because the color formatting or transfers modes are not handled directly by GLINT.

The floating point **RasterPosXIncrement** and **RasterPosYIncrement** registers are added to the current raster position X and Y coordinates respectively after the rectangle is rendered but not if the clip test fails. This auto increment cannot be disabled so if it is not desired the increment values should be set to zero. The increment is measured in pixels.

5.8.5 2D Rectangle

The **DrawRectangle2D** command provides a convenient way to set up the GLINT rasterizer to draw a rectangle.

The origin of the rectangle is supplied as data with the **DrawRectangle2D** command. The least significant 16 bits hold the 2's complement X coordinate while the most significant 16 bits hold the 2's complement Y coordinate.

The width and height of the rectangle, SyncOnHostData, SyncOnBitMask, SpanOperation mode and fill direction are defined by the **Rectangle2DMode** register. The choice of using

spans to fill the rectangle is determined by the least significant bit of the **Rectangle2DControl** register.

Bit No.	Name	Description
0…11	Width	Width of the rectangle. Twelve bit field with range 0…4095
12…23	Height	Height of the rectangle. Twelve bit field with range 0…4095
24	AreaStippleEnable	Passed to rasterizer in the **Render** command.
25	SyncOnBitMask	Passed to rasterizer in the **Render** command.
26	SyncOnHostData	Passed to rasterizer in the **Render** command.
27	TextureEnable	Passed to rasterizer in the **Render** command.
28	FogEnable	Passed to rasterizer in the **Render** command.
29	SpanOperation	Passed to rasterizer in the **Render** command.
30	HorizontalDirection	Sets the horizontal rasterization direction. 0 = Left to Right 1 = Right to Left
31	VerticalDirection	Sets the vertical rasterization direction. 0 = Increasing Y 1 = Decreasing Y

Table 5.13 Rectangle2DMode Fields

For OpenGL the main use of this is to clear the framebuffer. Note that in this case the ability to use the faster span method of clearing depends if GID window clipping is being used. OpenGL does not necessarily know when a window is clipped so by having the ownership of the **Rectangle2DControl** register reside with the Display Driver the Display Driver can change the rasterization method independent of OpenGL as a function of the window clipping.

The colors and modes set up by OpenGL for the clear operation need to satisfy span and non span clears.

For colors spans use **FBBlockColor** while non spans use **ConstantColor**. Also spans ignore many mode settings, such as Depth compare and Alpha blend. The additional set up is not really a performance issue as clear operations are infrequent and tend to take a comparatively long time.

For Display Driver GUI operations the **Rectangle2D** command provides a very fast way of setting up rectangles - just one register and one command compared to the 6 registers and one command when programming GLINT directly (this is obviously reduced when multiple rectangles are drawn one after the other).

The **XBias** and **YBias** are ignored for this primitive type.

6

Context Save and Restore

The natural boundary for context switching in a DMA driven system is on the completion of a DMA buffer. This requires that no internal state is carried over from one DMA buffer to the next as there are no mechanisms for saving and restoring the internal state. This has not been a problem so far because the main time this could arise is between the upper and lower trapezoids in a triangle - a situation which is easy to avoid.

OpenGL places no restrictions on the number of primitives between a Begin/End pair so circumstances will occur when it crosses one or more DMA buffer boundaries. With GLINT Delta this was not a problem as no private internal state was carried from one primitive to the next, however R5 carries a significant amount of state from one primitive to the next, especially for meshed primitives. The readback mechanism allows all user loadable registers to be read back, however it cannot return any of the internal state.

The software solution to this is for the software to track the last three vertices so a Begin/End sequence can be terminated at the end of a DMA buffer and restarted in the next DMA buffer without any change to the semantics of the original Begin/End sequence. The main objection to this technique is that the overheads of this tracking will seriously effect the system performance, particularly as the R5 interface only requires the minimal amount of host work to copy (and convert number formats) the vertex and normal arguments to the DMA buffer.

R5 has a mechanism which allows all the state (public as well as private) to be saved and restored in a consistent manner[31]. The context switch can only happen on a register load or command boundary so cannot happen part way through any internal processing, for example during clipping, so many intermediate results are not required to be saved. This simplifies the hardware while giving the software a convenient level of granularity to work with.

Even with this new context switching method, context switching in the middle of an image or texture download is not handled:

The R5 context switch is done by sending the **DumpContext** command. The data sent with this command (the context mask) dictates what subset of the full context is to be dumped. The context data (with the **ContextData** tag) will appear in the Host Out FIFO in GLINT. The last tag and data in the FIFO will be the **DumpContext** tag and the context mask. The context data is read from the Host Out FIFO and stored in memory in a context buffer (excluding any tags), the context mask is typically discarded. This context buffer can be restored by prefixing it with the three words: **RestoreContext** tag, context mask (used to generate the buffer in the first place) and the **ContextData** tag and loading it all back into R5. The **ContextData** tag has the upper 16 bits set to the number of words of *context data* in the buffer minus one[32]. The layout of the data in the context dump buffer is not important because no massaging of the data is necessary before it can be restored, and in fact, is largely undocumented.

---

[31] This will also be available in future rasterizer chips, however existing rasterizer chips will need to use the normal read back mechanisms for context switching.

[32] A tag with a count in the upper 16 bits is a hold mode tag so all the subsequent data is automatically given the same tag.

Restoring the context data is easily accomplished by the DMA controller. Saving the context data can also be done by the Output DMA controller, with a probably 10X speed up over software. The Output DMA controller can be used in one of two modes (see Section 0 for a more comprehensive description and details on how to initiate output DMA operations):

- Fixed Count. In this mode the Output DMA controller is given the *exact* number of words of context data to read from GLINT's Host Out FIFO. This count should also include the extra context mask. Setting an inappropriate count for the supplied context mask will likely lead to a lock up situation. The **FilterMode** register should be set up to only allow the context data and not tags through, and the Host Out FIFO should also be empty so as not to interpret any left over contents as context data.
- Variable Count. In this mode the Output DMA controller is placed in Feedback mode so will continue to transfer data from the GLINT's Host Out FIFO until an **EndOfFeedback** tag is found. The **FilterMode** register should be set up to allow both context data and tags through so tags and data inappropriate to this mode can be discarded and the **EndOfFeedback** tag can be identified. The Host Out FIFO does not need to be empty, however this would be preferable. The PCI **FeedbackSelectCount** register will hold the number of words written to memory when the Output DMA has finished. This method relieves the programmer from knowing before hand how much context data will be saved, however there may be a performance penalty in that one extra word (to hold the tag) passes on the secondary PCI bus between R5 and GLINT. This extra word may or may not effect the overall system performance depending on the primary PCI bus and host write performance.

The sequence of events to do a context switch are:

- Update the **FilterMode** register by setting bits 14 and 15 to enable the context tags and data respectively to be written into the host Output FIFO. These two bits are not documented in the GLINT Programmer Reference manual.
- Start the Output FIFO DMA controller transferring *n* words of context data, or in Feedback mode so the context data is terminated by the **EndOfFeedback** command.
- Send the **ContextDump** command to R5.
- Send the **EndOfFeedback** command if the Output FIFO DMA controller is being used in feedback mode.
- Send the **ContextRestore** command to R5 (for the new context).
- Copy the new context data into R5.
- The following observations can be made about this:
- No synchronization is necessary to context switch and the queued command DMA controller facility also helps here.
- The Output FIFO DMA controller cannot have transfers queued. In general this is not necessary as it is returning data which the application has asked for so it is already delayed waiting for the data. This does mean that the context switching software cannot just assume the Output FIFO DMA controller is always free.
- In a system with a R5 and one of the existing rasterizer chips the context save and restore will have to be done in two parts because the **ContextDump** command is not available in the rasterizer. The rasterizer context is saved using the normal **Sync** and readback methods. The R5 context is saved using the **ContextDump** command.

- Changing the state of the Host Out Unit in GLINT to allow the context data through will mean that the original context of this unit is not being saved. For existing rasterizer chips this will not matter as the context save is done in two parts. Future rasterizer chips will address this problem.
- The context dump method with the Output FIFO DMA controller should be capable of saving the context to host memory at an estimated rate of 40 to 60MB/s (this is *very* host dependent). This contrasts to the readback method of 4 to 6MB/s.
- All the context data is tagged with the **ContextData** tag and this is most easily achieved using a hold tag with a count. Tag and data pairs can be used but this is clearly less bandwidth and memory efficient.

R5 holds a considerable amount of state (public and private) and future rasterizer chips will add to this. Switching between two 3D contexts requires all (or nearly all) the context to be saved, while switching from a 3D context to a 2D context can get by with far fewer registers. The data associated with the **ContextDump** and **ContextRestore** commands give some control over what is saved and restored and obviously the same setting *must* be used with paired saves and restores.

Bit	Name	Context data includes	Word
0	GeneralControl	Mode and general control registers.	17
1	Geometry	Some user geometric state and much of the internal state.	377
2	Matrices	The user defined matrices.	82
3	Material	The user defined material parameters.	27
4	Lights0_7	The user defined light parameters for lights 0 to 7.	176
5	Lights8_15	The user defined light parameters for lights 8 to 15.	176
6	RasterPos	The raster position related state. This is expanded below so the current raster position, color, etc. can be read back to satisfy the OpenGL Get calls.	19
7	CurrentState	The current state. This is expanded below so the current texture, color, etc. can be read back to satisfy the OpenGL Get calls.	12
8	TwoD	The 2D related control registers.	2
9	DMA	The DMA related registers.	7
10	Select	The select related registers and name stack.	67

Table 6.1 Context Mask Fields

The actual contents of the context buffer is not particularly useful for a programmer to know - the user defined state is readily available via the read back mechanism and the internal state of R5 is not publicly documented.

The context dump mechanism is also used as a convenient way to report some internal state which can be queried by OpenGL, namely the current state (color, normal and texture) and the raster position. This data is in single precision floating point format unless otherwise noted.

If a context dump is done with only the CurrentState bit set then the resultant context buffer will hold the following information (the tags and context mask are assumed to have been discarded):

Offset	Data
0	Current edge flag in bit 5
1	Current normal, X component
2	Current normal, Y component
3	Current normal, Z component
4	Current texture, S component
5	Current texture, T component
6	Current texture, R component
7	Current texture, Q component
8	Current color, Red component
9	Current color, Green component
10	Current color, Blue component
11	Current color, Alpha component

If a context dump is done with only the RasterPos bit set then the resultant context buffer will hold the following information (the tags and context mask are assumed to have been discarded). Note that some user defined state is also included:

Offset	Data
0	Window coordinate, X component
1	Window coordinate, Y component
2	Window coordinate, Z component
3	Eye coordinate, Z component
4	Clip coordinate, W component
5	Texture, S component
6	Texture, T component
7	Texture, R component
8	Texture, Q component
9	Fog
10	In View (bit 0: 0 = out of view, 1 = in view)
11	xIncrement (user register)
12	yIncrement (user register)
13	xOffset (user register)
14	yOffset (user register)
15	Color, Red component
16	Color, Green component
17	Color, Blue component
18	Color, Alpha component

The **ContextRestore** with the CurrentState bit set can also be used to restore new current values such as might be required during the OpenGL glPopAttrib API function. In this case the new current values must be propagated through out R5 by using the **TransformCurrent** command. This command takes a four bit mask to specify which parameters are to be refreshed and in this case the simplest thing is to set all four bits (see the reference section for a description of these bits). Note if a full chip context restore is being done then the **TransformCurrent** command is not needed, however it will do no harm if it is sent after the full context restore, or indeed at any time.

The 2D operations do not make much use of R5 and can be completely switched using the TwoD and DMA settings.

7

OpenGL Specific Operations

7.1 Polygon Mode

Polygon mode allows primitives submitted as triangles, quads or polygons to be rendered as filled primitives, points at the vertices or lines connecting vertices.

The polygon mode can be different for front facing polygons and back facing polygons.

Bit No.	Name	Description
4, 5	FrontPolyMode	This field selects how a triangle, quad or polygon should be drawn when its orientation is forwards facing . The options are: 0 = Point 1 = Line 2 = Fill
6, 7	BackPolyMode	This field selects how a triangle or quad or polygon should be drawn when its orientation is backwards facing. The options are: 0 = Point 1 = Line 2 = Fill

Table 7.1 GeometryMode Register: fields that control the Polygon Mode.

7.2 Polygon Offset

Polygon offset provides a mechanism whereby a polygon is offset in Z by an amount given by the following equation:

$$offset = m \times factor + bias$$

where factor and bias are the **PolygonOffsetFactor** and **PolygonOffsetBias** registers respectively and m is an approximation to the z gradient of the triangle:

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}$$

The bias is the product of two components: a user defined value (called *units* in the OpenGL API) and an implementation constant, r. The constant, r, is the minimum resolvable difference (i.e. is the smallest difference in window coordinate z values that is guaranteed to

remain distinct throughout polygon rasterization and in the depth buffer). Typical values for r are given by:

$$\frac{3.0}{2^n - 1}$$

where n is the number of bits in the depth buffer and 3.0 is a constant found empirically to give good results.

Polygon offset is used to draw co-planar polygons slightly offset so the visual order of the polygons are guaranteed and no depth bleeding occurs between them. Examples of where this is useful is in adding decals to surfaces, shadow polygons or reflection polygons.

Polygon offset only applies to polygons, however a submitted polygon can be drawn as a series of points or lines under control of the Polygon Mode. Polygon offset can be enabled individually in each case and is controlled by three bits in the **GeometryMode** register.

Bit No.	Name	Description
28	PolygonOffsetPoint	This field, if set, causes the polygon offset to be calculated and applied to the points of a polygon when PolyMode is set to Point.
29	PolygonOffsetLine	This field, if set, causes the polygon offset to be calculated and applied to the lines of a polygon when PolyMode is set to Line.
30	PolygonOffsetFill	This field, if set, causes the polygon offset to be calculated and applied to the triangles of a polygon when PolyMode is set to Fill.

Table 7.2 Controlled Bits in the GeometryMode Register

7.3 Texture Generation

Texture generation allows some or all of the texture coordinates to be derived automatically from the incoming vertex coordinate or normal information. Each component of the texture (s, t, r and q) can have a different texture generation mode assigned to it. The four options are:

- None. The current texture coordinate is used.
- ObjectLinear. The incoming vertex is converted using the following equation:

$$g = p_a x_o + p_b y_o + p_c z_o + p_d w_o$$

where p is the user supplied coefficient held in the **TexGen**[16] registers are shown in Table 7.3

Target for *g*	p_a	p_b	p_c	p_d
s	TexGen0	TexGen4	TexGen8	TexGen12
t	TexGen1	TexGen5	TexGen9	TexGen13
r	TexGen2	TexGen6	TexGen10	TexGen14
q	TexGen3	TexGen7	TexGen11	TexGen15

Table 7.3 TexGen[16] Registers Target for g

- EyeLinear. The incoming vertex is transformed into eye space and then converted using the following equation:

$$h = p_a x_e + p_b y_e + p_c z_e + p_d w_e$$

where p is the user supplied coefficient held in the **TexGen**[16] registers are shown in Table 7.4

Target for *h*	p_a	p_b	p_c	p_d
s	TexGen0	TexGen4	TexGen8	TexGen12
t	TexGen1	TexGen5	TexGen9	TexGen13
r	TexGen2	TexGen6	TexGen10	TexGen14
q	TexGen3	TexGen7	TexGen11	TexGen15

Table 7.4 TexGen[16] Registers Target for h

- SphereMap. The transformed normal is used in the following equation:

$$\mathbf{r} = \hat{\mathbf{u}} - 2(\hat{\mathbf{n}} \bullet \hat{\mathbf{n}} \bullet \hat{\mathbf{u}})$$

$$m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$$

$$s = r_x / m + \frac{1}{2}$$

$$t = r_y / m + \frac{1}{2}$$

where

r is the reflection vector,

u is the unit vector pointing from the origin to the vertex in eye coordinates,

n is the unit transformed normal in eye space

• is a dot product

Note: SphereMap can only be applied to the s and t components.

OpenGL maintains separate 16 entry stores for the object linear and eye linear coefficients, where as in R5 they share a single 16 entry store (**TexGen**[16]). This means that the OpenGL driver needs to hold both sets and build up **TexGen**[16] depending on what the texture generation enables are set to.

Bit No.	Name	Description
5, 6	TexGenModeS	This field controls the automatic generation of texture coordinates for S texture component from the vertex or normal information. The Tex operations are: 0 None (use current texture S). 1 ObjectLinear. 2 EyeLinear. 3 SphereMap.
7, 8	TexGenModeT	This field controls the automatic generation of texture coordinates for T texture component from the vertex or normal information. The TexGen operations are: 0 None (use current texture T). 1 ObjectLinear. 2 EyeLinear. 3 SphereMap.
9, 10	TexGenModeR	This field controls the automatic generation of texture coordinates for R texture component from the vertex information. The TexGen operations are: 0 None (use current texture R). 1 ObjectLinear. 2 EyeLinear. 3 None (use current texture R, SphereMap is illegal).
11, 12	TexGenModeQ	This field controls the automatic generation of texture coordinates for Q texture component from the vertex information. The TexGen operations are: 0 None (use current texture Q). 1 ObjectLinear. 2 EyeLinear. 3 None (use current texture Q, SphereMap is illegal).
13	TexGenS	When this bit is set the S component of the texture coordinate is generated automatically, otherwise it is taken from the current texture value. This only has an effect when the TexGen operation is ObjectLinear, EyeLinear or SphereMap.
14	TexGenT	When this bit is set the T component of the texture coordinate is generated automatically, otherwise it is taken from the current texture value. This only has an effect when the TexGen operation is ObjectLinear, EyeLinear or SphereMap.

15	TexGenR	When this bit is set the R component of the texture coordinate is generated automatically, otherwise it is taken from the current texture R value. This only has an effect when the TexGen operation is ObjectLinear or EyeLinear.
16	TexGenQ	When this bit is set the Q component of the texture coordinate is generated automatically, otherwise it is taken from the current texture Q value. This only has an effect when the TexGen operation is ObjectLinear or EyeLinear.

Table 7.5 GeometryMode register Bits Controlling Texture Generation

After any texture generation has been done the resultant texture coordinate is optionally transformed.

7.4 Select Mode

The SelectMode is used typically during picking or selection operations of a user interface. Placing R5 into select mode and specifying the select data to return is controlled by the fields in the **GeometryMode** register as shown in Table 7.6.

Bit No.	Name	Description
14,15	RenderMode	The RenderMode field controls the action when processing any primitive. The options are: 0: Render 1: Select 2: Feedback

Table 7.6 GeometryMode register controlling Field

In OpenGL the select mode is broken down into two parts:

- All primitives are clipped and culled, but not rendered. If a primitive (or raster position) passed the clipping and culling phases then a hit flag is set and the minimum and maximum Z range grown, if necessary, to include this primitive. Subsequent primitives which also pass the clipping and backface culling may extend the minimum and/or maximum Z values.
- Name stack manipulation. The name stack holds names (as 32 bit integers) the user can push, pop or load to keep track of the model hierarchy. The commands **PushName**, **PopName**, **LoadName** do these actions. **PushName** and **LoadName** update the stack with the 32 bit value in the data field. The name stack is reset with the **InitNames** command. The name stack is 64 entries deep.

If the hit flag is set when a name stack manipulation is done a hit record is written to the host output FIFO (in GLINT). The hit flag is then reset along with the minimum and maximum Z range. The hit record consists of (in order):

- The count of the names (NameCount) on the stack (plus some error flags),
- The minimum Z value as a normalized floating point number,
- The maximum Z value as a normalized floating point number,
- The name stack entries, oldest first (variable number [0…64] words).

Bits 14 and 15 in the **FilterMode** register in GLINT must be set to allow the **SelectRecord** tag and data values to be written in to the FIFO - all the select record data uses the same tag.

The name stack manipulations commands are ignored when not in Select mode.

The hit record data is almost in the correct format for OpenGL. The only thing the software needs to do is to convert the minimum and maximum Z values from the floating point format (normalized to be in the range 0.0…1.0) to the 32 bit integer format required by OpenGL.

Bit	Name	Description
0…6	Count	This field holds the number of names on the name stack.
7…28		Not used.
29	InvalidOperation	A **LoadName** operation was attempted on an empty name stack when this hit record was being collected. This is cleared for subsequent hit records (unless they manifest this error) however the stack may no longer be totally valid.
30	StackUnderflow	The name stack was popped while empty when this hit record was being collected. This is cleared for subsequent hit records (unless they manifest this error) however the stack may no longer be totally valid.
31	StackOverflow	The name stack was pushed while full when this hit record was being collected. This is cleared for subsequent hit records (unless they manifest this error) however the stack may no longer be totally valid.

Table 7.7 NameCount Value Fields

The **SelectResult** command can be used to flush out a hit record (but only if the hit flag has been set) without having to do a name stack manipulation. This provides a convenient method when re-entering or leaving the select mode. The **SelectResult** command will also reset the hit flag.

During Select mode the lighting is still calculated even though it is never used so a useful optimization is to disable lighting. Similarly for texture and fog modes. Also it is advisable to disable short line and small triangle threshold testing and always do a full clip otherwise there is a chance that primitives which would have been clipped out actually get included in the selection process.

The amount of data generated in select mode is not easily determined by the host as it depends on how primitives are clipped and backface culled, and the name stack depth when the hit record is written. The **EndOfFeedback** command can be used by the host as a marker to indicate the end of the select stream. When this is found in the Host Out FIFO all the select data will have been read from GLINT.

Host software can read the Host Out FIFO, analyze the tags and build up an OpenGL conformant select buffer. This action of the host reading the FIFO must be done simultaneously with sending user commands or the Host Out FIFO will fill up and GLINT, and then R5 will stall. Alternatively the Output DMA controller can be placed in feedback mode to simplify and speed up this operation.

The Output DMA controller feedback mode:

- Discards the tags (these are still necessary in the Host Out FIFO so the type of data can be ascertained).
- Discard any surplus select data when the buffer is full.
- Terminate the transfer when the end of the select data is found (as indicated by the **EndOfFeedback** tag).

- Discard any invalid tag and data pairs while in select mode.

The overall method for implementing select with the Output DMA controller is as follows:

- The GLINT **FilterMode** is set up so that the tag and data for the Remainder group of tags (bits 14 and 15 set) are written to the host out FIFO and this FIFO is assumed to be empty after the filter mode has been set up.
- The **DMAOutputAddress** holds the address (logical or physical) where the select data is to be written The start address is given as a byte address but the lower two bits are ignored.
- The **DMAFeedback** command with the length of the memory buffer (in words) is sent to start the Output DMA controller.
- The RenderMode in the **GeometryMode** register is set to Select .
- The user rendering is done.
- The **EndOfFeedback** command is sent to mark the end of the select mode.
- Wait for all the select data to be transferred. This can be done by polling the *CommandStatus* PCI register or via an interrupt.
- Read the count of the number of words transferred from the *FeedbackSelectCount* PCI register. If the most significant bit is set then the buffer is full and there was more data to append to it (but this has been discarded).

If the select buffer becomes full before the **EndOfFeedback** tag in the tag/data stream is detected the Output DMA is terminated and the host informed, however the host output FIFO will continue to be read and its contents discarded until the **EndOfFeedback** is found. The **EndOfFeedback** tag and its data will also be discarded. The *FeedbackSelectCount* PCI register will hold the actual number of words transferred. Bit 31 is set if more data was found before the **EndOfFeedback** tag.

Once the select buffer has been updated the host software needs to parse the buffer to:

- Convert the minimum and maximum Z values from floating point format (in the range 0.0…1.0) to the integer format needed by OpenGL.
- Remove the status bits from the hit record (in the most significant byte of the name stack depth).
- Count the number of hit records present in the buffer.

7.5 Feedback

The OpenGL Feedback mode returns the vertex data for the primitives which have passed clipping and culling. Placing R5 into feedback mode and specifying the vertex data to return is controlled by the two fields in the **GeometryMode** register shown in Table 7.8.

Bit No.	Name	Description
14,15	RenderMode	The RenderMode field controls the action when processing ar primitive. The options are: 0: Render 1: Select 2: Feedback
16…18	FeedbackType	This field only has any effect if the RenderMode is Feedback. I this case it determines the parameters to be returned for eve primitive. The options are: 0: X, Y 1: X, Y, Z 2: X, Y, Z, R, G, B, A 3: X, Y, Z, R, G, B, A, S, T, R, Q 4: X, Y, Z, W, R, G, B, A, S, T, R, Q4DColorTexture

Table 7.8 Controlling Fields in the GeometryMode Register

While in Feedback mode no rendering is done, however details about what would have been rendered are returned.

The vertex data will appear in the host output FIFO in the order given here, i.e. the X value will appear first, followed by the Y value, etc. Bits 14 and 15 in the **FilterMode** register in GLINT must be set to allow the feedback tag and data values to be written in to the FIFO.

The data is identified in the FIFO by a tag starting with 'Feedback' with the appropriate name appended. For example the X value will have a tag of **FeedbackX**, the Red value a tag of **FeedbackRed**, etc.[33]. All the data is returned in floating point format.

The vertex data is associated with a particular primitive and before any of the vertex data is written to the FIFO the **FeedbackToken** tag will be written. The data with this tag defines what primitive would have been drawn and implicitly how many vertices worth of vertex data are in the FIFO.

[33]The full set of feedback tags is: **FeedbackX**, **FeedbackY**, **FeedbackZ**, **FeedbackW**, **FeedbackRed**, **FeedbackGreen**, **FeedbackBlue**, **FeedbackAlpha**, **FeedbackS**, **FeedbackT**, **FeedbackR**, **FeedbackQ**.

Data value	Primitive Type	Number of vertices
0x44e02000	Point	1
0x44e04000	Line	2
0x44e06000	Triangle	3
0x44e08000	Bitmap	1
0x44e0a000	DrawPixel	1
0x44e0c000	CopyPixel	1
0x44e0e000	LineReset	2
0x44e00000	PassThrough	0

Table 7.9 Data Field Possibilities

The LineReset is the same as a Line but the stipple pattern was reset for this line.

The hex values supplied match up with the tokens defined by OpenGL. Note that the Triangle token is the same as the Polygon token in OpenGL, however the vertex count is fixed at three and not included in the tag and data stream.

The Bitmap, DrawPixel and CopyPixel are all the Rectangle primitive with bits 13 and 14 (of the data value) set from the low two bits of the **GeomRectangle** command. These two bits are only used to generate the correct OpenGL feedback token. Note a value of 3 is used to indicate this rectangle shouldn't have an entry in the feedback buffer and this is used to prevent multiple feedback entries when the **GeomRectangle** is used several times to implement a single API call.

The number and type of primitives returned during feedback depends on what primitives are sent in the first place, the Polymode setting and the result of any clipping operation. Polygons and Quads are always decomposed into triangles and clipping a triangle can give anywhere between 0 and 14 triangles depending on the number of clipping planes (frustum and user) the original triangle cuts.

For some FeedbackType settings the color and/or texture values are not required. In these cases these can be disabled for higher performance (this does assume that the host reading back the feedback tags and data will not be the limiting factor). Fog can always be disabled. It is also advisable to disable short line and small triangle threshold testing and always do a full clip otherwise there is a chance that primitives which would have been clipped out actually get included in the feedback process.

It may be useful to inject markers while doing feedback (OpenGL has a function call glPassThrough to do this) to help keep track of which part of the model you are in. This is done by using the **PassThrough** command - the tag and data are written directly into the Host Out FIFO without changing any internal state in R5 or GLINT.

The amount of data generated in feedback mode is not easily determined by the host as it depends on how primitives are clipped, polygon mode and backface culling. The **EndOfFeedback** command can be used by the host as a marker to indicate the end of the feedback stream. When this is found in the Host Out FIFO all the feedback data will have been read from GLINT.

Host software can read the Host Out FIFO, analyze the tags and build up an OpenGL conferment feedback buffer. This action of the host reading the FIFO must be done simultaneously with sending user commands or the Host Out FIFO will fill up and GLINT,

and then R5 will stall. Alternatively the Output DMA controller can be placed in feedback mode to simplify and speed up this operation.

The Output DMA controller feedback mode does the following:

- Formats the feedback data from GLINT into the exact format required by OpenGL and write it into a memory buffer. This includes adding in the vertex count (3.0) for triangles (a.k.a. polygons), otherwise just involves discarding the tags (these are still necessary in the Host Out FIFO so the type of data can be ascertained).
- Converts the **PassThrough** tag to the PassThrough token and also append its associated data.
- Discards any surplus feedback data when the buffer is full.
- Terminates the transfer when the end of the feedback data is found (as indicated by the **EndOfFeedback** tag).
- Discards any invalid tag and data pairs while in feedback mode.

The overall method for implementing feedback with the Output DMA controller is as follows:

- The GLINT **FilterMode** is set up so that the tag and data for the Remainder group of tags (bits 14 and 15 set) are written to the host out FIFO and this FIFO is assumed to be empty after the filter mode has been set up.
- The **DMAOutputAddress** holds the address (logical or physical) where the feedback data is to be written The start address is given as a byte address but the lower two bits are ignored.
- The **DMAFeedback** command with the length of the memory buffer (in words) is sent to start the Output DMA controller.
- The RenderMode in the **GeometryMode** register is set to Feedback .
- The user rendering is done.
- The **EndOfFeedback** command is sent to mark the end of the feedback mode.
- Wait for all the feedback data to be transferred. This can be done by polling the *CommandStatus* PCI register or via an interrupt.
- Read the count of the number of words transferred from the *FeedbackSelectCount* PCI register. If the most significant bit is set then the buffer is full and there was more data to append to it (but this has been discarded).

If the feedback buffer becomes full before the **EndOfFeedback** tag in the tag/data stream is detected the Output DMA is terminated and the host informed, however the host output FIFO will continue to be read and its contents discarded until the **EndOfFeedback** is found. The **EndOfFeedback** tag and its data will also be discarded. The *FeedbackSelectCount* PCI register will hold the actual number of words transferred. Bit 31 is set if more data was found before the **EndOfFeedback** tag.

Note: *If the viewport mapping includes any additional biasing normally removed during the primitive set up operation (see section 5.8) using the **XBias** and **YBias** register values then the X and Y coordinates in the feedback buffer need to have the **XBias** and **YBias** values added to. This will restore the coordinates to the number range the application is expecting. This is not done automatically by R5 and is a post processing operation by software once the feedback buffer has been created.*

7.6 Raster Position

The raster position in OpenGL is set by writing to the **RP*** registers (**RPy, RPz, RPw** and one of **RPx2, RPx3** or **RPx4**). The raster position is transformed and lit as a normal vertex would be and the results saved away.

OpenGL has a rectangle primitive (glReadPixels, glWritePixels, etc.) where the raster position (previously established as above) defines the rectangle origin. The **RectangleWidth** and **RectangleHeight** registers define the width and height respectively. The texture, normal, fog, color, etc. values are calculated and stored with the raster position when the raster position is first defined.

The **GeomRectangle** command is used to render the rectangle. If the raster position is not in view then all **GeomRectangle** commands are ignored until a new raster position is established. If the raster position is in view then the operation is controlled by the data field.

Bit	Name	Description
0, 1	Type	These two bits define the type of rectangle to be inserted into the feedback buffer. They have no effect when not in feedback mode. The options are: 0 = Bitmap 1 = DrawPixel 2 = CopyPixel 3 = Don't insert into the feedback buffer.
2	OffsetEnable	When this bit is set the x and y offset values held in RasterPosXOffset and RasterPosYOffset respectively displace the raster position window coordinates when the rectangle is rendered. This does not update the raster position state.
3	SelectEnable	When this bit is set the rectangle takes part in the selection process.

Table 7.10 GeomRectangle Data Field

After every rectangle is submitted using the **GeomRectangle** command (in any RenderMode) the window coordinate x and y components are updated by the amount held in the **RasterPosXIncrement** and **RasterPosYIncrement** registers respectively. This occurs irrespective of the raster position being in view either before or after the update. If the initial raster position was in view then all subsequent raster positions updated via the increment will be in view. The converse also holds. The texture, fog and color values are only updated when the raster position is changed by an update to one of the **RPx2**, **RPx3** or **RPx4** registers.

The width and height of the rectangle is held in the **RectangleWidth** and **RectangleHeight** registers as floating point numbers. The **RectangleMode** holds the low level enables to control TextureEnable, FogEnable, etc. and has the same format as the GLINT **Render** command.

A raster position should not be sent between a Begin/End sequence as it will corrupt the vertex data being carried from one primitive to the next (an OpenGL requirement as well).

For OpenGL bitmap operations the supplied **RasterPosOffsetX** and **RasterPosOffsetY** should be set from the -origin values given in the API call. Also 0.4999 (nearly half) should

be subtracted as well to get the required floor operation the OpenGL specification required for bit maps.

All the state associated with the raster position can be saved and restored via the context dump and context restore mechanisms (see the context saving section for details). This mechanism is also used to get the current raster position when it is queried by the glGet function in OpenGL.

If a **ContextDump** is done with only the RasterPos bit set then the resultant context buffer will hold the following information (the tags and context mask are assumed to have been discarded). Note there is some user defined state also included:

Offset	Data	Offset	Data
0	Window coordinate, X component	10	In View (bit 0: 0 = out of view,
1	Window coordinate, Y component		1 = in view)
2	Window coordinate, Z component	11	xIncrement (user register)
3	Eye coordinate, Z component	12	yIncrement (user register)
4	Clip coordinate, W component	13	xOffset (user register)
5	Texture, S component	14	yOffset (user register)
6	Texture, T component	15	Color, Red component
7	Texture, R component	16	Color, Green component
8	Texture, Q component	17	Color, Blue component
9	Fog	18	Color, Alpha component

This data is in single precision floating point format unless otherwise noted.

7.7 Current Texture, Normal and Color values

The current values for texture, normal and color are updated when texture coordinates, normals and colors are written to R5. OpenGL allows these to be queried and these can be read back from R5 or the **ContextDump** command with the *CurrentState* context mask bit set.

If a context dump is done with only the CurrentState bit set then the resultant context buffer will hold the following information (the tags and context mask are assumed to have been discarded):

Offset	Data	Offset	Data
0	Current edge flag in bit 5	6	Current texture, R component
1	Current normal, X component	7	Current texture, Q component
2	Current normal, Y component	8	Current color, Red component
3	Current normal, Z component	9	Current color, Green component
4	Current texture, S component	10	Current color, Blue component
5	Current texture, T component	11	Current color, Alpha component

This data is in single precision floating point format unless otherwise noted.

The current data can be restored by writing to their assigned registers (as would normally be done to pass this data to R5 in the first place) or by using the **ContextRestore** command with the CurrentState context mask bit set. In this case the new current values must be propagated through out R5 by using the **TransformCurrent** command. This command takes a four bit mask to specify which parameters are to be refreshed and in this case the simplest thing is to set all four bits (see the reference section for a description of these bits).

The current values can also be saved and restored using a set of internal registers. This is controlled by the **SaveCurrent** and **RestoreCurrent** commands. There is only one set of registers so the **SaveCurrent** command will overwrite the data saved by the previous **SaveCurrent**. This mechanism is much faster and simpler to use than using the **DumpContext** and **RestoreContext** commands.

OpenGL evaluators (used for curved surfaces, NURBS, etc.) calculate texture, normal and color values (depending on what is enabled) and they are treated as if the programmer had supplied them directly, but with one important exception. The current values (or any derived state such as material parameters edited using ColorMaterial) are not changed. The SaveCurrent and RestoreCurrent commands are used to bracket the evaluator operations. All derived state is also refreshed by the TransformCurrent command.

7.8 Window Clipping Support

R5 does not have any support for window clipping. The method of repeating a DMA buffer used by the earlier 3Dlabs drivers can be used, however a *full* R5 context dump is required before the DMA buffer is read for the first time. On subsequent repeats of the DMA buffer the previous R5 state is restored. Clearly this is unlikely to be a high performance solution.

Window clipping should be done using the Graphics ID (GID) facility in GLINT. The GID is stored in the GLINT's local buffer and provides a pixel ownership test. See one of the GLINT Programmer's Reference manuals for more details on using GIDs for window clipping.

The OpenGL driver does not know when one of its windows is clipped and even if there were a callback to notify of this situation this does not help any DMA buffers already generated, and waiting to be read.

GID testing is controlled by the **Window** register and the **LBReadMode** register (to enable local buffer reads), unfortunately the display (GUI) driver needs to own some bits in these registers while the OpenGL driver need to own others. To prevent some sort of software handshaking or locking being necessary to control access to these register R5 intercepts writes to these registers and keeps a local copy. The local copies of these registers can be modified using the **WindowAnd, WindowOr, LBReadModeAnd and LBReadModeOr** commands. Recall that the data associated with the command is logically combined with the existing data so individual bits can be modified in isolation. Once the local copy is updated the GLINT copy is updated. One further modification to the **LBReadMode** is done and this is to enable destination reads if GID testing is required.

The **Rectangle2DControl** register holds a single bit which is set when window clipping is needed. This bit is used to force local buffer reads to be done and this register is owned by the display driver.

GID testing, for the most part on the GLINT MX, is a free test, however it is not compatible with the span operations. Normally these are used by 2D GUI operations which are pre clipped to the window so don't need to use GIDs. There is one situation where OpenGL would use spans and this is to clear the color buffer (i.e. as part of the glClear function). When GID testing is being used OpenGL can't use spans to implement the clear. To avoid OpenGL needing to know which style of clears to use, or using a driver call to do the clear it sets up the clear for both methods and lets the display driver choose depending if window clipping is necessary.

OpenGL does the set up for clears assuming a single pixel at a time rendering will be used. In addition the clear color (in framebuffer raw format) is written to the **FBBlockColor** register. The **Rectangle2D** command is used to render the rectangle. The

Rectangle2DControl register, under control of the driver, will select the appropriate rasterization method.

7.9 Color Material Support

OpenGL has the facility to allow the current color or a color sent to the graphics pipeline to change one or more selected material parameters. In R5 there is no performance penalty for changing a material's parameter directly rather than via the color material mechanism. The **ColorMaterialMode** register controls the color material mechanism.

Bit No.	Name	Description
0	Enable	When set causes a vertex color to update the material parameter(s) for the given face(s).
1, 2	Face	This field selects which face(s) any material changes should be made to by the updating color. The values are: 0 = front material 1 = back material 2 = front and back material
3…5	Parameter	This three bit field selects which material parameter(s) should be updated by the updating color. The values are: 0 = Emissive 1 = Ambient 2 = Diffuse 3 = Specular 4 = Ambient and diffuse

Table 7.11 ColorMaterialMode Register Fields

When color material is first enabled the existing current color is immediately applied to the selected material parameter(s).

The OpenGL specification states that material edits using glMaterial do not affect any material properties selected for update by **ColorMaterial.** R5 does not enforce this behavior.

Note: The OpenGL specification stipulates that when ColorMaterial is enabled the target material parameter(s) are updated by calls to glColor. This is handled automatically by R5. The OpenGL specification also goes on to state that the target material parameters are no longer updated by calls to glMaterial. This is not done automatically by R5 and the OpenGL driver must do this filtering.

7.10 Get Operations

OpenGL provides a query mechanism. The parameters which can be queried broadly fall into three categories:

- User defined state. User defined state is state the application sets directly, for example a light's color or position. The OpenGL driver can choose to track this state itself or read it back from the hardware. If the state is to be read back from the hardware any pending DMA must be finished and R5 and GLINT synchronized with first.

- Derived state. Derived state is state which could be tracked by the OpenGL, but will impact performance too much to do so. State in this category is the current texture, normal and color values. The material state may be included in this if Color Material is enabled.
- Transformed state. This is state which undergoes significant processing before the results are queried. An example of this is raster position, and unless a software geometry and lighting pipeline is maintained there is no alternative but to read back the state from R5.

Keeping a software copy of state looks like it is a big performance win, however it does have one big drawback - state changes cannot be included in native display lists as these would not be reflected by the software copy when the display list was executed.

7.11 Display Lists

Display lists can be held in two formats: - internal OpenGL format which is parsed into R5 commands when executed, or in native R5 format. Clearly the native format will be much faster and is the preferred format.

The native format suffers from three drawbacks:

- R5 relies on the host to do any matrix generation and concatenation so OpenGL commands for this cannot exist in a display list.
- Mode changes may involve changing one or two bits in a register and leaving the rest of the bits unaffected. This requires a software copy of the register to be used to re-generate the bits to keep. The register contents may be different when the display list is executed from when the display list was created so needs active involvement from software. All the R5 mode registers have an And and an Or version to allow individual bits to be changed.
- Interdependencies in mode bits. For example the framebuffer, in general, needs to be enabled for reading when alpha blending is done, however some alpha blending modes don't use the framebuffer contents. Meanwhile, in a different mode register the logical operation may or may not require framebuffer data to be read as well.

Future R5 devices and new rasterizers will address these problems so most operations can be included in display lists. Meanwhile a composite scheme of mixing native and parsed display lists based on their contents is a good compromise.

8

API Functionality Support

8.1 Diffuse Textures

Some APIs allow a diffuse lighting component to be applied after texture mapping. The diffuse values are interpolated across a triangle and are derived from the diffuse light value at each vertex (see the earlier lighting equations).

Diffuse textures are enabled by the *DiffuseTextureEnable* bit in **Material Mode**, *DiffuseTextureEnable* bit in **DeltaMode** and the *TextureEnable* bit in the **Begin** command. The diffuse texture (at each vertex) is calculated using:

$$\mathbf{diffuseTexture} = \mathbf{e}_{cm} + \mathbf{a}_{cm} + \mathbf{ambientLight} + \mathbf{diffuseLight}$$

where

$\mathbf{e}_{cm}$ is the emissive material color

$\mathbf{a}_{cm}$ is the ambient material color

8.2 Specular Textures

Direct3D allows a specular lighting component to be applied after texture mapping. The specular values are interpolated across a triangle and are derived from the specular light value at each vertex (see the earlier lighting equations).

Specular textures are enabled by the *SpecularTextureEnable* bit in **Material Mode**, *SpecularTextureEnable* bit in **DeltaMode** and the *TextureEnable* bit in the **Begin** command. The specular texture (at each vertex) is calculated using specularTexture = SpecularLight.

9
Debugging and Pipeline Control

This is a proposed new chapter covering the new and improved facilities for interrupt generation, status reporting etc. and to suggest strategies for setting up debuggable code using R5 facilities.

10 Appendices

10.1 PCI Address Map

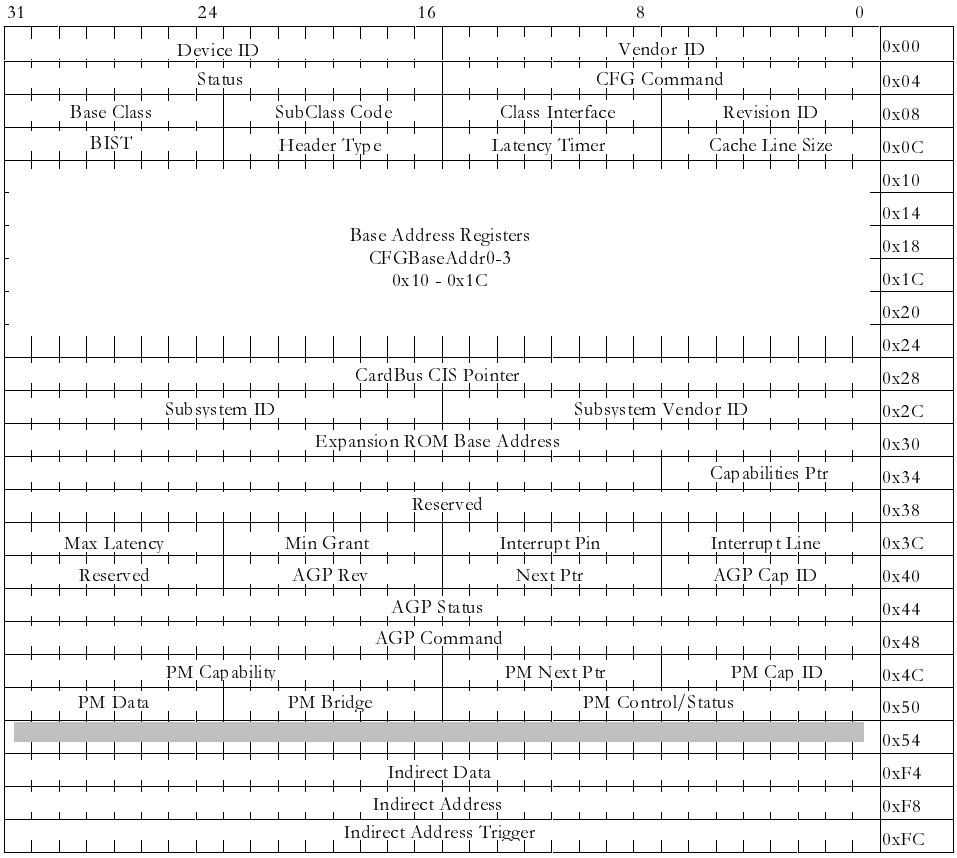


Figure 10.1 PCI Configuration Region

10.2 Region Zero Address Map

The GLINT R4 region zero address map is shown in Table 2-1:

Address Range	Region Select	Byte Swap/ Write Combined
0000.0000 -> 0000.01FF	Control Status	No
0000.0200 -> 0000.02FF	Test Control	No
0000.0300 -> 0000.03FF	Bypass Control	No
0000.0400 -> 0000.0FFF	Repeats Control, Terst & Bypass Decodes	No
0000.1000 -> 0000.1BFF	Memory Control	No
0000.1C00 -> 0000.1FFF	FB Sync Data FIFO	No
0000.2000 -> 0000.2FFF	GP FIFO Access	No
0000.3000 -> 0000.33FF	Video & Overlay Control	No
0000.3400 -> 0000.37FF	DMA Arbiter	No
0000.3800 -> 0000.3FFF	Reserved	No
0000.4000 -> 0000.4FFF	RAMDAC	No
0000.5000 -> 0000.57FF	Reserved	No
0000.5800 -> 0000.5FFF	VSCtl	No
0000.6000 -> 0000.6FFF	VGA Control	No
0000.7000 -> 0000.7FFF	TextureData FIFO	No
0000.8000 -> 0000.FFFF	GP Registers	No
0001.0000 -> 0001.01FF	Control Status	Yes
0001.0200 -> 0001.02FF	Test Control	Yes
0001.0300 -> 0001.03FF	Bypass Control	Yes
0001.0400 -> 0001.0FFF	Repeats Control, Terst & Bypass Decodes	Yes
0001.1000 -> 0001.1BFF	Memory Control	Yes
0001.1C00 -> 0001.1FFF	FB Sync Data FIFO	Yes
0001.2000 -> 0001.2FFF	GP FIFO Access	Yes
0001.3000 -> 0001.33FF	Video & Overlay Control	Yes
0001.3400 -> 0001.37FF	DMA Arbiter	Yes
0001.3800 -> 0001.3FFF	Reserved	Yes
0001.4000 -> 0001.4FFF	RAMDAC	Yes
0001.5000 -> 0001.57FF	Reserved	Yes
0001.5800 -> 0001.5FFF	VSCtl	Yes
0001.6000 -> 0001.6FFF	VGA Control	Yes
0001.7000 -> 0001.7FFF	TextureData FIFO	Yes
0001.8000 -> 0001.FFFF	GP Registers	Yes

Table 2.1 Region Zero Address Map

10.3 PCI Address Regions

GLINT R4 has six base address regions, as shown in Table 2-2:

Region	Address Space	Bytes	Description	Comments
Config	Configuration	256	PCI Configuration	PCI special
Zero	Memory	128K	Control Registers	Relocatable
One	Memory	128M	Memory Aperture One	Relocatable
Two	Memory	128M	Memory Aperture Two	Relocatable
ROM	Memory	64 K	Expansion ROM	Relocatable
VGA	Memory & I/O	—	VGA Address	Optional and Fixed

Table 10.2 GLINT R4 PCI Address Regions

10.4 Pipeline Architecture

GLINT R5 uses an integrated pipeline with a variety of additional units to provide enhancements and services such as switchable 2D/3D Routing, hardware Context Dump/Restore, Statistics and Bypass modes. Looking only at the primary data flow for 3D graphics, the pipeline would look as shown below. The units are primarily virtual, i.e. collections of related functionality, rather than physical. Units operate asynchronously but are in practice synchronised to a great extent by their common clock domains.

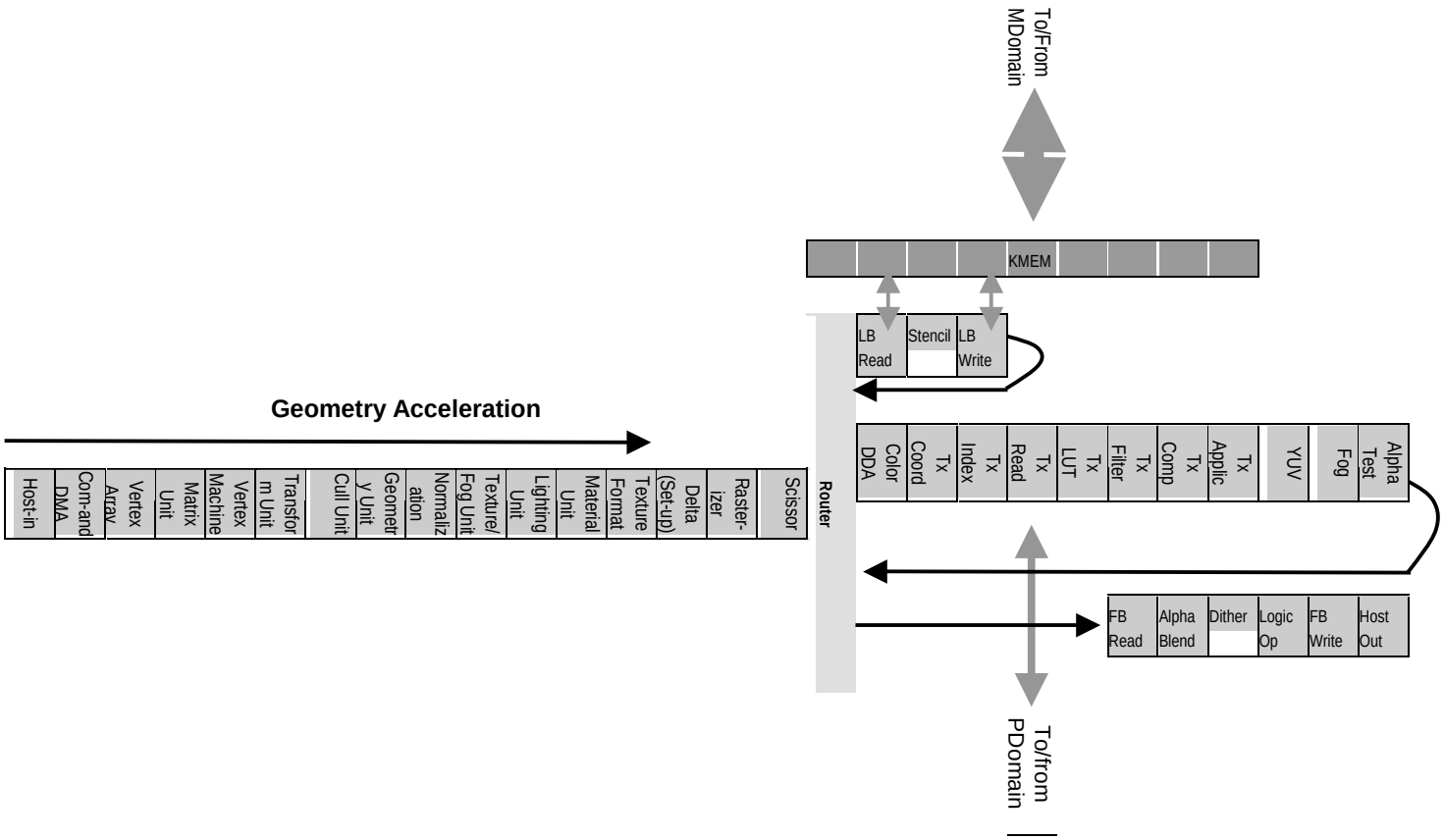


Figure 10.3 Pipeline Architecture

INDEX