P10 ®

Programmer's Guide

DRAFT

PROPRIETARY AND CONFIDENTIAL INFORMATION

3D*labs*®

P10 ®

Programmers Guide

PROPRIETARY AND CONFIDENTIAL INFORMATION

Issue 1

Proprietary Notice

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

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

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

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

Change History

Document	Issue	Date	Change
174.1.4 01	1	25/06/2001	Creation
174.1.4 02	2	17/10/2001	Final edit and release

Table of Contents

1 Introduction

1.1 Introduction

The Miranda P10 Graphics Processor is the first of a radical new chipset series with a highly scalable, multi-texture/multi-fragment per clock cycle architecture. This industry-leading design uses extensive parallelism and programmability to provide future-proof support for new, texture-intensive APIs such as Microsoft DX8.

Using ***programmable T&L*** and ***programmable pixel shaders*** in conjunction with highly optimised fixed-function units results in a simpler, faster and more flexible design.

Programmable registers also allow dynamic reconfiguration of the number of vertex shaders, the number of texture pipes and the number of rasterizers per chip to deliver the greatest possible throughput under changing task conditions.

Fixed-function registers for specialised tasks have been optimised for simplicity and speed with hand-polished main routines and the removal of legacy code. Memory bandwidth and DMA performance have been enhanced with support for high-density DDR memory configurations up to 8 x 8Mx32 (when available) and low overhead circular buffers to provide up to 17Gbytes/second peak throughput.

3Dlabs has achieved this without compromising its long-standing commitment to quality 3D rendering. P10 delivers accuracy, stability and full OpenGL compliance while providing a feature-rich device with unparalled real-world single-chip graphics performance.

1.2 How to Use This Manual

The *Miranda P10 Programmers' Guide* should be read together with the *Miranda P10 Reference Guide* which contains all of the referenced register descriptions.

The Programmers Guide contains:

- an overview of Miranda P10, its capabilities and architecture, highlighting key differences between the P10 and earlier chipsets.
- details of the programming model for the chip, including DMA circular buffers, host bypass to unified framebuffer, programmable unit encoding and examples, vertex loading and context caching.
- describes the data structures that P10 supports in the framebuffer and the localbuffer.
- describes the Video System including timing, RAMDAC and overlays.

- Appendix A gives the format used in the pseudocode examples throughout the document.
- Appendix B (TBC) gives further examples for unit microcoding

Following the body of the manual, a technical glossary defines many of the 2D/3D graphics terms used throughout.

These documents are cross referenced and hyperlinked in the Word DOC format. To take advantage of this hyperlink facility please keep your P10 manuals in one directory.

1.3 Further Reading

The following texts may be helpful in providing additonal information or clarifying 3D programming concepts:

- **3Dlabs Publications:**
- *Miranda P10 Reference Guide*, 3Dlabs
- *Miranda P10 Architecture Overview*
- *Miranda P10 Datasheet*

OpenGL References:

- *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 (see below)*
- *Computer Graphics: Principles and Practice, James D. Foley et al, Reading MA: Addison-Wesley*

Platform and API References:

- *PCI Local Bus Specification Rev2.1, 1Jun95, PCI Special Interest Group, PO Box 14070, Hillsboro, Oregon 97214 (503-797-4207)*
- *Multiprocessor Methods For Computer Graphics Rendering, Scott Whitman, ISBN 0-86720-229-7*
- *Microsoft WIN32 Software Development Kit 3.1, Microsoft*
- *Windows NT 3.1 Graphics Programming, Emeryville CA, Ziff-Davis Press*
- *The X Window System, Sebastopol CA, O'Reilly & Associates Inc.*
- *The X Window System Server, Elias Israel and Erik Fortune, Digital Press*

2 Memory

2.1 Data Formats

As data is gathered, cached as vertices, indexed, DMAd, transformed, clipped and rendered it takes a number of forms, from raw input to floats and fixed ints, to vectors and normals, to rendered primitives and video bus data. Much of this is common and described in industry reference standards. However in some respects P10 is unusual, particularly in its use of tiled memory instead of conventional linear memory.

2.1.1 Local Memory Data Format

P10 data is **not** laid out in the traditional linear model. Instead all data is grouped into 8 byte by 8 byte tiles which are then stacked through memory. Instead of addressing by byte position, data is accessed by tile number, as shown in the following illustration:

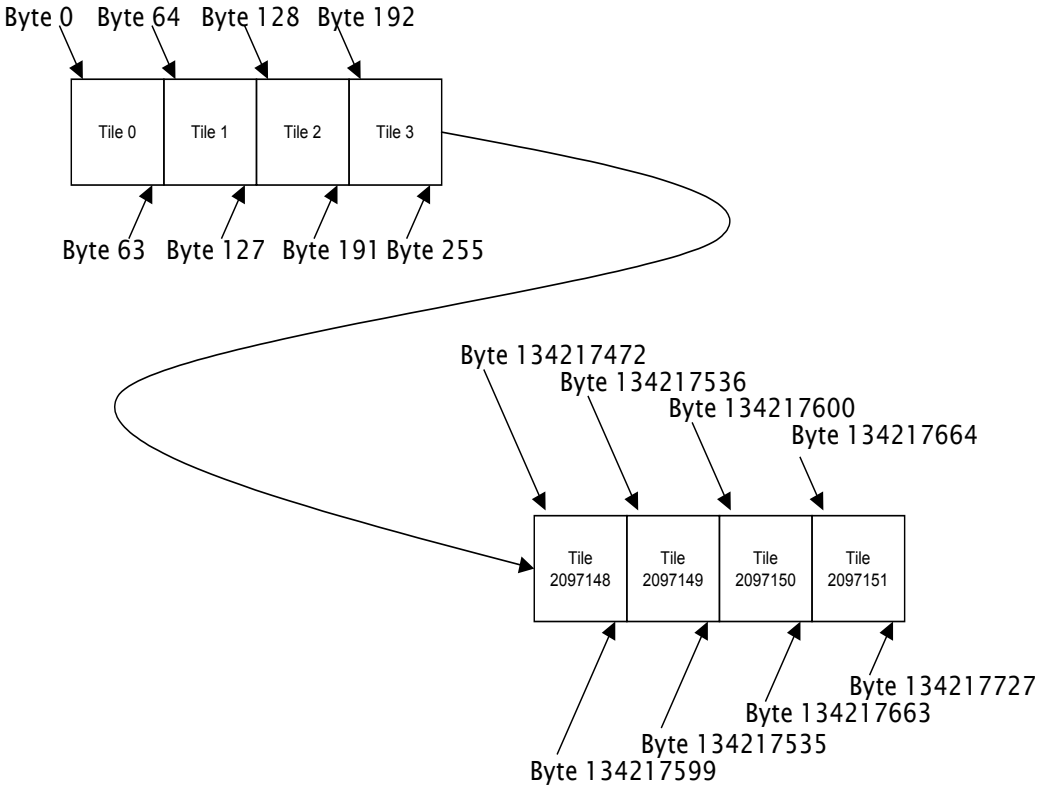


Figure 2.1 Linear Tile Addressing

Normally, each tile corresponds to a region of a buffer (perhaps the framebuffer or a texture); if the data type held in the buffer needs more than one byte per entry the bytes are held in separate tiles so this is called a 'byte planar' format.

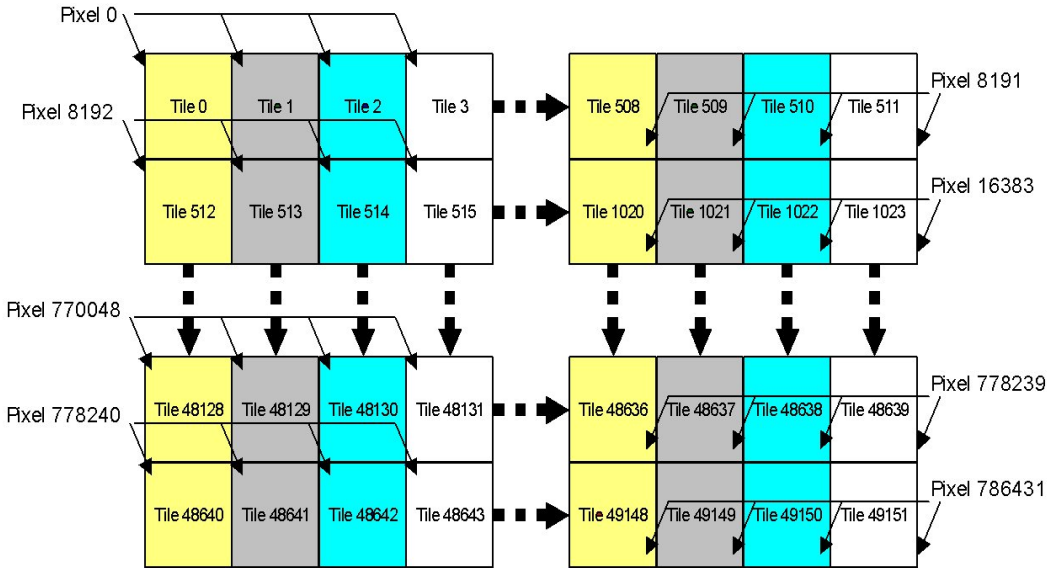


Figure 2.2 Tile to Pixel Mapping

This diagram shows a 1024x768 screen at 8 bits per pixel. At 32 bits per pixel each color component takes one byte so consecutive tiles hold red, green, blue, and alpha. The 1024x768 screen looks like this:

Figure 2.3 Tile to Color Byte Mapping

2.1.2 Bypass Accesses

The PCI Bypass Unit connects the PCI bus with the P10 memory controller. It provides a “bypass” path around the graphics core, through which software can read and write local memory directly. The memory controller reads and writes 64bytes at a time from local memory. The PCI Bypass unit has the following functions:

- Combine writes from the bus interface to reduce memory bandwidth requirements.
- Track outstanding memory writes to determine when all data has reached memory.
- Cache read data from the memory to reduce subsequent bus interface read latency.
- Optionally convert between linear and planar byte tile memory accesses based on the bypass address.

For a full definition of the registers controlling the Bypass mechanism refer to the *P10 Reference Guide* Bypass Registers

PCI Bypass can be configured to read/write from the host to local graphics memory with or without memory format conversion. The data should be converted if its destination is ultimately the graphics core, but not if it is intended for the GPIO units. For example, if a

texture is transferred from host into local memory destined for the graphics core, then format conversion should be enabled. (Or alternatively the format conversion has to be done by the host in software.)

A command stream (e.g. an OpenGL display list) on the other hand is usually written to local memory with the intention that it will be read by the GPIO units. In this case no conversion should be enabled. The next section describing the GPIO Data Format expands on this point.

2.1.3 GPIO Data Format

The GPIO input can read either from host memory or from local graphics memory. The GPIO interprets memory as a standard linear byte stream format (not the planar byte tile format). More details are given in a later section on GPIO about the alignment requirements of the data, the byte swapping capabilities, the multiple simultaneous data buffers that can be set up, etc.

Since the memory is interpreted as a linear byte stream, when the host is building a command stream into a buffer in host memory, the data should be written in the normal linear fashion. The GPIO can read the command stream from host memory using DMA and will interpret the data as expected.

This also applies when the host is building a command stream in local graphics memory. The local memory is mapped in by the host using the Bypass mechanism - in this case should be set to its conventional linear mode. When the GPIO later reads the same memory the memory subsystem will return the data in tiles. However each tile is simply interpreted as a linear stream of 64 bytes so the net result is that the command stream is interpreted correctly.

2.1.4 Re-circulating Data

The difference between the two formats - planar byte tile for the graphics core and linear bytes for the GPIO – means that some care is required where command data is recirculated from graphics core to the GPIO. For example, during displacement mapping vertex data, generated by a Vertex Shader program, is written out to local memory and then read back in via the GPIO.

Some suggested solutions to the memory reformatting necessary for this case form a part of section 10.2.2 that describes the implementation of displacement mapping in detail.

2.2 Memory Management Introduction

P10 provides the basic tools for implementing a memory management system. The mechanisms provided are:

- a page table to map a logical address to a physical address and determine the validity of pages;
- an interrupt to indicate a page fault;
- a DMA controller to facilitate the transfer of pages between system memory and graphics memory under software control.

We discuss the usefulness of these approaches the following sections.

2.2.1 Advantages and Disadvantages of Virtual Memory

The page table mechanism provided by P10 allows a level of indirection to be set up that maps from a logical address space to a physical address space. This flexibility gives rise to the advantages provided by the concept of virtual memory.

One of the advantages of virtual memory is that of easier and more efficient management of a physical memory resource. When a memory management system is running it has to handle numerous allocate and free requests of different sizes and at different times. Typically over time this leads to fragmentation of the physical memory, so that the unused sections of memory can be scattered about the address range in small pieces. This can lead to the situation where an allocate request for a contiguous address range cannot be satisfied by any of the available unused contiguous address ranges, despite the fact that the unused memory in total would be sufficient to satisfy the request. Virtual memory makes it possible to handle this situation, by exporting a contiguous logical address range that satisfies the allocation request. The logical to physical mapping allows the logical address range to be supported by numerous unused physical pages that are not at physically contiguous addresses.

Another advantage of virtual memory is the ability to be able to support situations where the memory requirements exceed the size of the physical memory available (which therefore could not be supported with a simple physical memory management). Virtual memory allows the physical memory to be treated as a fast cache, which can be used in combination with host memory and even host disk space to support memory requirements much greater than just the cache size.

The only disadvantage of virtual memory is the possible impact the memory address translation may have on the memory access performance. Any performance impact is obviously implementation dependant.

2.3 Address Translation Without Page Faulting

This section considers the use of simply enabling address translation on P10 to achieve the benefit of improved physical memory management efficiency. In this case the logical address space can be considered to be of the same range as the physical address space and the page faulting mechanism is not used.

2.3.1 Address Translation Initialisation

A number of control registers manage the behaviour of the memory controller. For a full definition of these control registers refer to the *P10 Reference Guide* volume 2, : Control Registers. All registers are on 64 bit boundaries except the fifo registers which are packed to allow bursts. The register definitions show addresses in multiples of 32 bits.

Firstly, the memory should be idle before any changes are made, which can be tested by checking the busy flag in the **MemoryControl** register. Then the following registers need to be set up:

2.3.1.1 MemoryPageTableLower and MemoryPageTableUpper

The type bitfield is set to video if the page tables are being held in local memory (which is the preferred option for best address translation performance).

The address at which the page table entries are to be defined is given as a page address, not byte address, and can make use of a maximum of 52 bits. The lower 20bits of the

address are written appropriately in **MemoryPageTableLower** and the upper bits in **MemoryPageTableUpper**.

2.3.1.2 MemoryPageTableLimit

The size of the desired logical address range precisely defines the number of page table entries needed. One page table entry is needed for every 4K of logical address range. The total size of the page table entries (in units of pages) is written to the MemoryPageTableLimit register.

Address translation can be enabled/disabled differently per memory access type using the **MemoryTranslationEnable** register:

- The Bypass bit controls bypass read/write accesses.
- The Texture bit controls texture read accesses.
- The GraphicsProcessor bit controls graphics core read/write accesses, apart from texture accesses.
- The CommandProcessor bit controls the GPIO read accesses.
- The Video bits control the read accesses made for video refresh.

A typical scenario will be to enable translation consistently for all accesses, apart perhaps from the video accesses.

2.3.1.3 MemoryPageTableLimit exceeded

If a virtual (byte) address exceeds the virtual address range implied by the Pager Table limit then page 0 of the Page Table is examined and defines how the “Out of Range” access is handled. Later sections (below) describe how pagetable entries can be initialised to report page faults.

Page Table Format

Each table entry allows for a 64bit byte-aligned physical address, and to achieve this each entry takes 64 bits and is packed on 64 bit boundaries. The format of a page-table entry is:

Bit	Name	Description
0-1	State	0 = Invalid 1 = Not Resident 2 = Read Only 3 = Read/Write
2	ContinueOnFault	0 = Stall until fault fixed 1 = Report fault and continue
3-5	Size	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
6-7	Type	0 = Video 1 = System (PCI) 2 = System (AGP) 3 = Reserved
8-11	Reserved	
12-63	PhysicalPage	Start address of the 4K page in physical memory

Table 2.1 Format for Page Table Entries

2.3.1.4 The *State* Field

Expanding the State bitfield in more detail:

- A value of 0 marks the page as invalid; accessing an invalid page is an error and will result in a page fault interrupt.
- A value of 1 marks the page as valid but not resident in physical memory and accessing the page will result in a page fault interrupt.
- A value of 2 marks the page as valid, resident and read only. Write access to this page will result in a page fault interrupt.
- A value of 3 marks the page as valid, resident and allows write as well as read access.

At start of day, prior to any allocations from the logical address space, the State field for all entries will be 0. Then, as allocations proceed, for a non-paging scheme the relevant entries will be updated to values of either 2 or 3 and only reset to 0 when the allocation is explicitly freed.

2.3.1.5 *ContinueOnFault* Field

The *ContinueOnFault* field allows P10 to support complex virtual texture paging operations.

The *ContinueOnFault* field is normally 0. As defined by the *State* field, a page access may result in a page fault which causes an interrupt and stalls the page access. This allows the device driver to either:

- signal an access error or
- repair the fault, change the page state, and restart the access.

However if *ContinueOnFault* = 1, the page access takes place regardless of whether a page fault interrupt has been signalled or not.

The *PhysicalPage* field in the Page Table must always be a valid address when *ContinueOnFault* is set Similarly when **MemoryTranslationEnable** enables *Bypass*, then the bypass accesses automatically act as if *ContinueOnFault* is set regardless of the Page Table entry, and the entry for bypass access must always be associated with a valid physical page.. This safeguard prevents the system locking out all bus accesses due to a bus access fault when bus access is needed to repair the fault.

*Note: Continue-on-Fault uses the physical address so the offending access is performed **before** the fault is flagged..*

2.3.1.6 *PhysicalPage* and *Size* Fields

Pages in memory are always 4K bytes, and each 4K page has its own page table entry, with the *PhysicalPage* field giving the start address of that 4K page in physical memory. The *Size* 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 *Size* field has the value “2” it indicates this 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 *Size* field allows address translation hardware to optimise reading of the page tables and reduce the number of TLB updates, although the hardware can choose to ignore this information and will still operate correctly (because the *PhysicalPage* field always contains the correct start address for each individual 4K page regardless of any Size information). Where possible buffers should be allocated with the largest page size possible. A 4K byte page holds 64 tiles which run horizontally across the screen. Moving vertically will result in a TLB miss for every tile if large pages are not used. A 512K byte page will hold 128 lines of a 1024 wide screen of 32 bit pixels.

2.3.1.7 *Type* Field

The *Type* bitfield is set to 0 if the physical memory is local graphics memory. Values of 1 or 2 are used when the memory has been allocated from host system memory and in addition has been locked down to a physical page of system memory. The DMA transfer protocol to access the system memory is either PCI (1) or AGP (2). If the system memory is cacheable memory then using PCI protocol will guarantee to give the right results (cache snooping occurs). If AGP protocol is wanted then the host software will need to take care of any cache flushing required explicitly. The Type field can be different for each page table entry, so in general the physical memory can be a mixture of local and system memory.

The *PhysicalPage* field normally holds the physical address of the page; this is a 64bit byte address to support extended addressing into system memory. The address is actually given in units of pages; hence the bit field is 52 bits.

2.4 Memory Management With Page Faulting

This section describes the use of the page faulting support on P10 in combination with address translation to achieve a system that can support large memory requirements (greater than the physical local memory available).

Note: This section assumes that the reader is familiar with P10's plain address translation support, described in section 2.3, Address Translation.

The basic method is to define a logical address range that can be much larger than the size of the local physical memory. Then the physical memory is treated as a fast cache that can be used in combination with system memory to support the logical address range. Every 4K of logical address range has a corresponding page table entry. A subset of these entries will map to physical memory. Some of the other entries are mapped instead to system virtual memory that acts as backing store. Any memory accesses within the address range of a page table that does not map to physical memory is arranged to cause a page fault interrupt and the graphics core is stalled.

The interrupt gives an opportunity for the system host to reallocate the mappings in the page table entries. First the host arranges for the entry that faulted to be given a page in physical memory. This may involve reusing a physical address that was previously allocated to another entry (the other entry is therefore updated to map to appropriate backing store and loses its physical mapping). The host then has to arrange for the physical address to be loaded with the correct data for the faulting entry. This will typically involve first locking down the system memory acting as backing store. A DMA controller is then available to copy the data from system physical memory to the physical memory allocated for the faulting page table entry. Once the physical page and its data are in place, the host finally signals for the hardware to proceed again.

A typical memory management scheme can provide two types of allocation from the logical address range. One type creates a mapping to physical addresses that are then permanently fixed to that allocation and never released for re-use until the whole allocation is explicitly freed. Allocations for graphics buffers such as front and back buffers and depth buffers are typical cases. The other type of allocation maps to physical addresses where these addresses are allowed to be reallocated as new allocation requests are invoked (i.e. allows the paging mechanism to operate). Texture maps are a typical example of use for this allocation type.

2.4.1 Page Table Format Revisited

The *State* bitfield is set to 1 for any entries that are being used for a logical allocation but do not currently map to a physical address. In this case the *PhysicalPage* field can be used by the host memory management to hold a system virtual address that points to the data. The Type field is set to 0 if the data is to be copied into a local memory physical address.

The *ContinueOnFault* field is typically set to 0, which causes the graphics core to stall until a page fault is signalled as fixed (i.e. correct data in a physical page is present).

All transactions that cause a fault must complete eventually and it is not possible to kill an operation that has been started. If it is important that, for example, a write to read-only page does not complete, the page should be remapped to a safe area of memory.

Note that transactions originating as slave PCI/AGP accesses are not suspended if they fault. Suspending bus transactions could cause deadlock, so the hardware behaviour in this case is that the access completes and the fault is reported to indicate there has been an error. These slave accesses correspond to bypass reads or writes, which should therefore always be to a page that is present and valid.

2.4.2 DMA Controller

When a page fault is detected data will normally have to be transferred from system memory to video memory. This may be done directly by the CPU reading system data and writing it directly to the chip, or by programming the DMA controller. The sequence of operations to fix a fault will usually be:

1. Fault detected.
2. Retrieve ID giving cause of fault retrieved from the PageControl fifo.
3. Remedy determined, list of pages to be paged out and paged in constructed.
4. DMA and table update commands sent to PageControl fifo.

Note: Page Control FIFO operations which cause a Page Table update use a logical page number which is not validated against the PageTableLimit register.

The cause of the page fault is communicated to the host via 4 dwords through the Page control fifo. These dwords are read from the fifo in order, while the FIFO Count !=0

FIFO Fault Word	Description	
Fault ID	The ID identifies the source of the address fault and is used to restart the correct part of the memory controller. The ID matches the bit field values defined for the **MemoryTranslation Enable** command (see below).	
Logical page address	The actual page that faulted.	
Table entry (1)	64bit page entry from the page table that caused the fault (differs from logical page address when **PageTable**.*Size* > 4	Low 32 bits
Table entry (2)		High 32 bits

Table 2.2 FIFO Page Fault Data

ID Value	Fault Location
0	Graphics Processor
1	VGA
2	Command Processor
3	Bypass
4	Reserved
5	Reserved
6	Video Processor Head 0
7	Video Processor Head 1

Table 2.3 Page Fault Location ID Values

From this information the backing store for the faulting address can be identified and the fault processed.

If the system memory used as source or destination of paging is not locked down then the CPU must handle the copy directly through the bypass. Table update commands should still be sent through the PageControl fifo to ensure correct ordering. All commands sent to the PageControl fifo take 4 dwords (1 AGP fast write). The format is:

Word	Bits	
1	0,1	Command 0 = Table update 1 = System to video DMA 2 = Video to system DMA 3 = Invalidate
	2	Interrupt on completion
	3..9	Reserved
	10..31	Page
Command = Table update		
2	0..31	Table entry
3	0..31	Table entry
Command = DMA (either type)		
2	0..5	Reserved
	6..7	Type 0 = Reserved 1 = PCI 2 = AGP
	8..11	Reserved
	12..31	System page
3	0..31	System page
Command = Invalidate		

2	0..31	Reserved
3	0..31	Reserved
All commands		
4	0..7	Restart
	8..15	Reserved
	16..23	Suspend
	24..31	Reserved

Table 2.4 Page Control FIFO Update Command Structure

The first word holds the type of command in the lower 2 bits. The options are to:

- modify a page table entry
- transfer data from system to video memory
- transfer data from video to system memory, or
- invalidate the entries in the TLB (translation look-aside buffer, a cache of page table entries).

An additional bit indicates that an interrupt should be raised when the command has completed. The upper bits of the first word hold a page number that this command will use.

The following 2 words are interpreted according to the command:

- When doing a page table update the second and third words hold the page table data that should be loaded into the table at the address given previously.
- If the command is a DMA then the page number specified in the first word is taken as the address in local memory. The following two words specify the system address (which is a page aligned 64 bit address).
- If the command is to invalidate the TLB, the second and third words are reserved.

The fourth word holds a mask of the possible source of memory transactions that should be suspended before this command starts, and restarted after it completes. This gives control over the accesses to areas of memory that are being updated to resolve a page fault.

When a page fault occurs, the memory controller halts all further processing on the source of the fault. As there are several sources, each with their own dedicated fifo into the memory controller, page translations can be done in parallel.

After the host has performed all the necessary transferring of data and page table updates to satisfy the page fault, the chip must be signaled to indicate that the faulting source can continue with its access request. The memory controller is told to continue with processing on a faulted source by raising the corresponding bit in the restart field of the fourth word. This not only allows processing to restart on that source, but also invalidates the associated TLB cache, thus causing the TLB entries that have been outdated by the page fault handling to be re-loaded.

When using the Restart mask in operations such as Table Update a suspend mask must also be used. The Suspend mask must must have bits set for every currently-suspended source.

2.4.2.1 Manipulating multiple 4K pages

Page faults that require the updating of several pages of memory require special consideration. For the entire duration of the updates, the relevant suspend bit in the fourth word must be held high. This ensures that the memory controller maintains the fault as blocking the associated source.

On the last page manipulation operation, the relevant restart bit is held high and the memory controller is finally restarted.

This mode of operation is required when dealing with several pages at a time. The host has the option to perform a number of page updates when a fault occurs, rather than fixing up only the page that faulted. This might be an advantageous for performance reasons if the host has knowledge that related pages are also very likely to be needed. Overhead costs such as the OS calls required to lock down system memory can be amortised over a larger number of page updates.

Also, the suspend bit is required to be held high when dealing with faults on pages that have sizes larger than 4K, because the DMA engine can only handle 4K of data at a time and so the data must be copied in 4K chunks.

2.4.3 Page Replacement Algorithms

When a new physical page is needed to satisfy a page fault it is possible that an existing page will have to be reused. There are three commonly used page replacement algorithms: least recently used (LRU), first-in first-out (FIFO), and random.

LRU is generally considered the best algorithm. It replaces the page that has been least recently used (i.e. was last used further back in time than any other). It relies on future activity being similar to past activity. Its draw back is the cost of maintaining a list of page usage. Most UNIX systems use modified LRU that replaces the full list with an approximation constructed periodically by a paging demon.

FIFO replaces the page that was loaded further back in time than any other. This is subtly different to LRU because it does not take usage into account. In this case, data which is frequently accessed over a long period of time can still get swapped out and then back in again. Windows NT uses this algorithm because the management overheads are much lower (per page fault instead of per access).

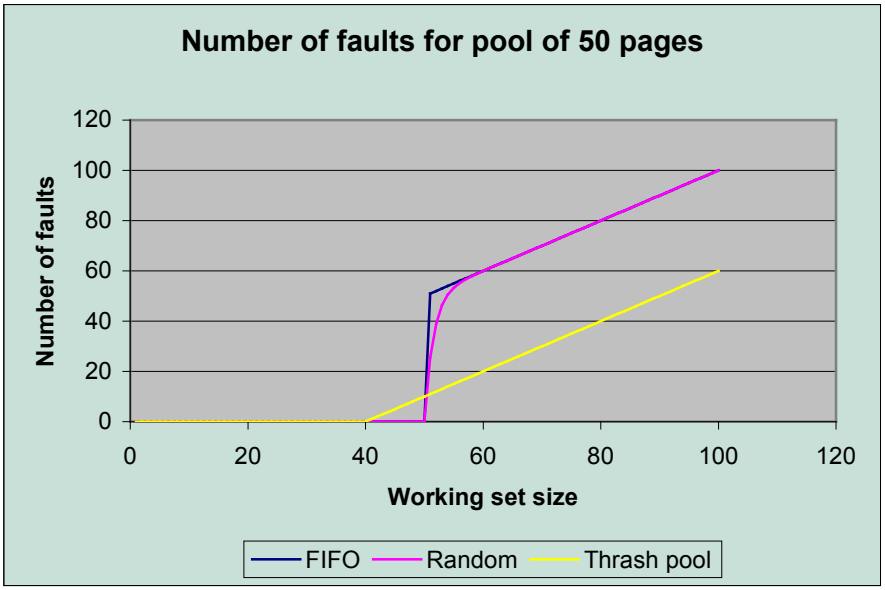
Random replacement simply picks the page to replace at random, and is normally the worst algorithm, although it can improve performance of some list processing with particularly random access patterns.

P10 hardware does not track page usage data (i.e. per access), so the LRU algorithm is not an option, but the FIFO algorithm is a recommended option. Consider the behaviour of the replacement algorithms for two common scenarios: thrashing and working set replacement.

Thrashing happens when the amount of texture needed for a given frame is greater than the pool of pages available. In this situation the LRU algorithm gives the worst performance. Simplifying the situation, assume that all texture is accessed linearly, each page being used once in each frame. The LRU algorithm replaces the oldest page, which means that by the end of the frame every page will have been swapped - i.e. there is no reuse. The performance of LRU is good up to the point where one more page than there is room for is needed, at which point it collapses and requires every page to be loaded

each frame. FIFO performance is similar. Random replacement is slightly better than FIFO because there is a (small) chance that useful data is retained.

The best way to handle thrashing appears to be to allocate a new 'thrash pool' which is separate to the normal working set pool. When thrashing is detected the working set pool is left as it is and all new pages are loaded into the thrash pool. This prevents useful pages being swapped out and only thrashes the overflow from the working set. The replacement algorithm used within the thrash pool can be the same as is normally used for the working set.



The chart shows steady state page replacement performance when the system is thrashing; it assumes a working pool of 50 pages and working set that ranges from zero to 100 pages. The chart shows that FIFO performance hits a cliff as soon as the working set is bigger than the pool. Random replacement is better because, at least with relatively few pages over the pool size, some useful data is retained. Reserving space for the thrash pool makes it start issuing faults earlier (10 pages are reserved), but as the working set size increases it is better at keeping useful data resident.

Thrashing can be detected by counting the number of faults reported in a frame. If it reaches the number of pages in the working pool the system is thrashing. There is no easy way to detect the end of thrashing after switching over to using the thrash pool, other than to occasionally switch back to working pool.

Working set replacement occurs when the scene drawn changes and some textures are no longer used and new ones need to be loaded. This situation favours LRU replacement. FIFO is a close approximation except for textures that are common across multiple scenes. These will age and be swapped out, only to be needed again within the same frame or the next one. Reloading common textures can be detected and minimized if the frame number that a page is swapped out is recorded in the replacement list. If it is reloaded in the same or next frame, then it can be treated as common and moved to a common pool that is kept resident. If the working pool starts to thrash then the common

pool is added to the working pool so that pages that are no longer common can be swapped out.

3 Input and Output

3.1 Where to store commands and data

3.1.1 Host memory

In general DMA buffers are written once by the CPU and are read only once by the graphics chip. For this reason it may be more efficient to store them in host memory where they can be accessed quickly by the CPU and then 'sucked' across the bus at AGP 4X rates.

Earlier 3Dlabs GLINT chips used this approach and it may still be the most efficient way for P10 in many cases. On legacy chips the overhead of starting a DMA was quite large which meant that performing many small DMA's was inefficient. P10 by contrast is very efficient for small DMAs.

3.1.1.1 Framebuffer vs. Host Memory

A problem with storing DMA buffers and other data structures in host memory is Host Memory Contention. This arises where the host AGP chipset can't get access to host memory because it is being used by another process (e.g. SCSI controller or CPU). (This problem is characteristic of low-end PC systems with built-in graphics controllers that partition part of host memory and use it as the framebuffer.) The latest Pentium 4-class systems offer much improved memory performance but memory contention is still likely to be a problem. For this reason items accessed frequently by the graphics chip should be stored in the framebuffer.

3.1.1.2 Framebuffer

Traditionally the framebuffer for a 3D graphics card consists of a front buffer, back-buffer and a cache of 3D textures, 2D bitmaps, fonts and various other 'surfaces'.

It makes sense for objects that need to be accessed frequently by the graphics chip (and infrequently by the host) to be stored in the framebuffer. Data structures such as **OpenGL Display Lists**, which are compiled once by the CPU and read multiple times by the graphics chip, used to be considered ideal candidates for framebuffer storage. Because of their size (100 Mbytes or more for high-end CAD applications) it often wasn't possible to fit them into a 32MB framebuffer. With P10's Logical Memory it is now possible to have a large virtual framebuffer (2 GB) and page in parts of data structures as needed.

There can also be a problem with **framebuffer memory contention** (video refresh etc) however it should be less noticeable than host memory contention because the memory interface of the graphics card is fast and optimized for the task whereas PC memory is generally slower and less specialized.

3.1.1.3 Host chipset/CPU implications

Traditionally graphics chips have been programmed by storing graphics chip commands and data in DMA buffers. These buffers tended to be physically contiguous, non-pageable memory and were either **cacheable** (when the card is plugged into a PCI bus) or **uncached (**when plugged into an AGP bus).[1]

Note: If P10 Logical Memory is enabled the DMA buffer no longer needs to be contiguous - the graphics chip can scatter-gather the pages from host memory. However the memory should be 'locked down' on the host side (to stop the O/S paging it out) and also be shown as non-pageable to P10 so that it does in turn does not try to page it out.

When Intel added **USWC** (i.e. write-combining) support into their X86 processor, this small CPU cache provided a significant throughput boost when copying data into an uncached DMA buffer. USWC works by grouping up writes to USWC memory in a small cache and sending them in a burst rather than in single transactions.

The framebuffer can also be marked as USWC. This means that any writes to it will have their AGP burst size optimized in the same way as writes to a host memory DMA buffer.

Intel further tweaked the USWC support in the Pentium 4 processor, making it more difficult to use efficiently. However it is now possible to use **cached AGP DMA** buffers and an efficient instruction to flush the specific CPU cache lines before starting a DMA. This is not "privileged" – it can be called from applications where previously the flush instruction cleared all cache lines but was only callable by the kernel.

Unlike earlier chips, P10 supports **AGP Fast Writes.** This feature improves performance when writing directly to the framebuffer and makes it possible to store the DMA buffer in the framebuffer itself.

Note: Storing the DMA buffer in the framebuffer may not improve performance but it does use valuable framebuffer memory.

3.2 Programed I/O vs. DMA

The simplest method of getting data into a piece of hardware is to write it, one word at a time, into some sort of input register. The problem with this is that a large FIFO buffer is required on this register in order to smooth over periods when the hardware can not consume the incoming data stream fast enough. Such large FIFOs are expensive in hardware terms and so undesirable. The host must also regularly check the amount of free space in this FIFO so as not to overflow it.

The alternative method is for the host to write its data stream into a normal memory buffer somewhere and let the hardware read it out via DMA as and when it is ready. Using standard system or graphics memory for this means there is no additional hardware cost for having very large buffers (e.g. 1MB or more). It also means that there can be an arbitrary number of independant buffers. So multiple processes can be filling in their own private buffers without fear of interference.

[1] PCI chipsets snoop the CPU cache which is why PCI supports cached memory but AGP does not.

3.2.1 The Input Message Port

Due to these performance limitations of the FIFO scheme, it was decided to include such a facility only as a debugging tool. The interface presented to the user is a very simple message port without any buffering (i.e. a 1-deep FIFO). The registers for controlling this are:

- **IMsgReady**
- **IMsgTag**
- **IMsgData0** -> **IMsgData3**

The **IMsgReady** register contains a single bit which indicates the status of the IMP. If it is set, then data can be written to the other registers. This must be checked before every message/command which is sent to this port.

IMsgTag contains the 10-bit tag value and a 2-bit size. The 1-based size field indicates how many of the **IMsgData#** registers will be used to provide the data field. Writing the last 32-bit data value will trigger the sending of the message. Messages are always sent with 128 bits of data, if fewer bits are provided by the user, the following defaults are used:

- IMsgData0: must be provided
- IMsgData1: IEEE float 0.0
- IMsgData2: IEEE float 0.0
- IMsgData3: IEEE float 1.0

There is one message port for each rendering channel through the P10 core (see section 3.5, “Dual Command Streams”).

3.2.2 The DMA Interface

The proper way of getting data into the chip is to use DMA.

Miranda implements a sophisticated circular buffer scheme in hardware. The basic idea is to make the whole process as efficient as possible. This means keeping the buffer management overheads to a minimum.

The scheme is based around a hardware managed buffer with two pointers in to it. One is the chip's read pointer, the other is the host's write pointer. Basic operation is that the host will write to the front of the buffer and increment the write pointer. The chip will then start reading this data, incrementing the read pointer as it goes. By having both of these pointers as hardware registers, the chip will always know how much data is available for reading, and the host can always find out exactly how much data has been processed.

The buffer is initialised at start of day by setting **CbufAddr,** the logical address to read from (the start of the buffer). This can be in host memory or in video memory (see earlier for the pros and cons of each).

- Byte swapping enables if required.
- *CBufEnableBusy*:
- The context address if required (see section 3.6, Multiple Contexts).

The read and write pointers (CBufWrPtr & CBufRdPtr) can then be used. These are given as 32-bit word offsets from the start of the buffer. This means that the user's code does not need to worry about where the buffer is mapped in to the hardware's logical address space.

3.3 Circular DMA Buffers

Because the read and write pointers are hardware managed, the buffer can be treated as a single, circular DMA buffer with a transfer granularity of one 32-bit word.

Wrapping around the end of the buffer is achieved by the simple operation of writing the write pointer back at the start. The DMA engine has no concept of where the end of the buffer is, it only knows what the highest write pointer value was and will read up to there. It does know what the buffer's start address value was though. So when the write pointer is set to a lower value than it was previously, this is interpreted as:

- Old (higher) value = end of buffer read from read pointer to here
- New (lower) value = 'current end' of next buffer read from start to here

The only restriction on the value put in the write pointer is that the engine must have finished reading any old data which was 'underneath' the new stuff. I.e., it must not overtake the read pointer. From a software point of view, this is a valid restriction anyway because if it happens, then the host will have been over writing data which had not yet been transferred.

3.3.1 Layout, Read/Write Pointers and Scheduling - Software Implementation:

In general, the host will want to send DMA data in chunks of one or more complete commands. At the very least, it must not split a tag from its associated data. Also, the DMA code does not want to be reading or writing across the AGP bus for every word of dta which is to be added to the buffer. This implies a scheme involving three basic operations:

- Wait for X words of space
- Queue some data
- Send all queued data

3.3.1.1 Wait For Space:

This should be called before any chunk of data is added to the DMA queue. It must reserve at least as much space as there is data to queue. However, there is no need to get the figure exactly right, there is no problem or penalty if too much space is reserved (as long as it is not hugely inaccurate) as the buffer itself will normally be several orders of magnitude larger than the request.

What this operation needs to do is to add the requested size to the current write pointer and compare this new, potential write pointer to the current hardware read pointer. If it has gone past, then the routine must block in some manner until the read pointer has moved on enough.

A Mental Block

There are a number of ways the blocking can be achieved depending upon the intended application, the level of OS support, etc. The most basic is to simply poll the read pointer continuously until it reaches a suitable value. To avoid hogging the AGP bus, this polling loop should contain some sort of delay even if it is only executing a processor no-op instruction several hundred (or thousand, according to CPU speed) times.

A more system friendly approach would be to use some sort of OS provided semaphore mechanism. Command IDs (see below) could be inserted into the DMA stream at strategic intervals (e.g. after any particular large render command, or just every X words). For safety, the wait routine should probably add its own id as well, just to make sure there is one in the buffer somewhere (although this means that the wait must always leave some space after it and never allow the buffer to fill completely).

The wait routine would then call the OS to block its process until a specific semaphore object has been signalled. The **CommandId**s would be set to generate an interrupt and the interrupt handler would signal this object and so wake the wait process.

Of course, care must be taken to ensure that the semaphore is cleared, signalled and waited on in a sensible order otherwise the **CommandId** could get missed and the system would deadlock.

Learning to Read:

Care must be taken when examining the read pointer because it can be in two different positions relative to the write pointer giving different calculations of how much fress space there is:

```
  |<-----free----->|<---pending--->|<-----free----->|
  |                |               |                |     Free = End - Write
  |----------------+---------------+----------------|
Start            Read           Write             End

  |<----pending--->|<-----free---->|<----pending--->|
  |                |               |                |     Free = Read - Write
  |----------------+---------------+----------------|
Start            Write           Read            End
```

Round and Round and Round and ...

If the new write pointer is actually outside of the DMA buffer, i.e. it has run off the end, then the code must wrap around to the start again. In theory this is simply a case of starting the free space test again with a base address of zero rather than the current write pointer.

In practice, there is a problem if the request is for more than half the buffer size. For example, a request is made for X words and does not fit in the remaing space, X is also more than half the buffer size:

```
                        |<-----------------X----------------->|
      |-----------------+-------+------------------------|
    Start             Write    Centre                      End

After wrapping, the picture is:

      |<-----------------X----------------->|
      |-----------------+-------+------------------------|
    Start              Last     Centre                      End
                      Write
```

The problem here is that the last write pointer, and hence the place the read pointer will eventually stop at, is not far enough in for the request to be satisfied no matter how long the routine waits for.

- There are a number of solutions to this problem:
- Do not allow requests of more than half the buffer size;
- Keep track of the last write pointer and treat the buffer as empty when the read pointer matches it;
- Always queue some fake data at the start of the buffer when wrapping (e.g. a CommandId tag which will be consumed by the DMA engine itself and so not interfere with any ongoing rendering). This will ensure that the read pointer always wraps along with the wait routine and so will present an empty (less two words for the CommandId tag) buffer.

3.3.1.2 Queue Some Data:

This would be done by having a software only copy of the write pointer which is incremented as data is written into the buffer at its position.

3.3.1.3 Send The Data:

This is simply a case of copying the above software write pointer into the hardware one.

3.4 DMA

DMA use in P10 is covered in previous sections on the DMA Interface (sect. 3.2.2) and Circular DMA (section 3.3) and in Isochronous DMA (section 3.5 below).

3.5 Dual Command Streams

P10 supports two independent input streams: Graphics Processing (3D and regular 2D) and Isochronous.

3.5.1 DMA stream

Input streams can be:

- Regular DMA
- Circular buffer (up to 16, automatically scheduled)
- From host memory or on-card memory
- At two levels of hierarchy

3.5.2 Vertex and Index Data Stream

The GPIO unit supports two forms of caching of vertex and index data. Data can be cached either in the unit itself or in a post-T+L form further down the pipeline. The latter method can only be used with indexed primitives. This is discussed further in Section 3.7.2, *Caching* Indexed Vertex Arrays can use 8, 16 and 32 bit indices.

- Two sets of indices
- Index caching to control vertex caches
- Independent lines, triangles and quads only

Vertex Array Elements may be:

- 1 to 4 words per parameter
- 1 to 16 parameters per element
- 1 to 16 disjoint parameter streams

…and are fully DX8 compatible

3.5.3 Output DMA

The Output DMA stream returns data to host memory or local memory. It supports Image uploads and State return

3.6 Multiple Contexts

In a modern system, there can be several different applications all wanting to draw their own private displays. The Miranda architecture provides dedicated support for up to sixteen completely independant rendering streams. As there is only one pipeline through the geometry and rasteriser engines, these different streams are multiplexed together at the DMA engine level.

Basically, there are sixteen different sets of read and write pointers, enables and base addresses. These operate entirely independantly and so can be written to or read from by multiple processes or threads (even if they are running concurrently on multiple processors) without fear of interference.

This is achieved via a DMA scheduler. It is the scheduler's job to arbitrate between the various active DMA contexts and to keep the rest of the chip informed as to which context it is processing work for.

3.6.1 Context Switching

In addition to issuing the DMA requests for each context as appropriate, the scheduler also tells the rest of the chip whether a given DMA request belongs to a different context from the previous one or not.?? In the case where the new request is for a new context, the old context's hardware state must be saved away and the new one's restored.

As previously mentioned, each context has an enable register which contains a context address. This is the logical tile address that the hardware state for that context is saved to. All impotnant state for the geometry and rasteriser units is saved and restored in this manner. The only exceptions are the internal registers of some of the programmable units. See the sections on each unit for details on programming for safe context switching.

3.6.2 User-induced and Isochronous Switches

TBA

3.6.2.1 Time Stamps

TBA

3.6.3 Context Scheduling

The scheduler is based around a FIFO queing mechanism. Whenever a given context's write pointer is updated by the host, that context is added to the FIFO. If the context is already in the FIFO, the two requests are simply concatenated. Thus the FIFO can never overflow no matter how much work the host is queueing up.

As long as the output pipe from the scheduler is unblocked, the context at the head of the FIFO will be removed and whatever work it has outstanding will be forwarded to the DMA engine proper to be processed.

There is also a timeout mechanism which can be used to prevent context thrashing. For example, if two contexts are continuously being given tiny amounts of work to do then the scheduler can end up switching at the same rate and so issuing very inefficient, tiny DMAs with a full context save and restore on either side. If a time out value is set, then although the work will be issued for the context at the head of the queue as and when it comes in, that context will not be removed until after the time out expires. This will hold the next context off for a while and let it build up a reasonable amount of work which will subsequently be issued in a single, larger, more efficient lump. It also removes the majority of the context save and restores.

In practice, the timeout mechanism may not be necessary. If the chip can keep up with the requests no matter how inefficient they are, then there is no problem. If it can not, then it will back up behind which ever unit is busy. When the blockage reaches the scheduler, it will cease removing entries from its FIFO and begin concatenating requests. As soon as the blockage is gone, these larger requests will be issued. The pipeline between the scheduler and the DMA engine proper is very short so inefficient DMA requests will quickly stall the scheduler. Thus the whole susyem is fairly well self-regulating

3.6.4 Driver Controlled Scheduling:

Under certain circumstances, it may be necessary for the driver to exercise some control over the order in which the various contexts are processed.

The simple level of control is the **SuspendUsr** register. This is a bit field of suspend flags, one for each of the sixteen user contexts. Any context whose bit is set will be blocked from placing its entry in the scheduler's FIFO. So this can be used to temporarily suspend one or more contexts until some particular condition has been met.

If more control is required, then the driver can take over the entire scheduling process and do everything manually. This is done by enabling the time out interrupt and the time out itself. The driver can then set the suspend mask to only allow one particular context to run. When the timeout interrupt occurs, the mask can be updated to allow a different context to run. In this way, the driver can schedule the various contexts in any order it sees fit in a completely deterministic manner irrespective of how the applications themselves are wanting to queue up their work..

3.6.5 Context Security

For reasons of efficiency it can be desirable to let a user-land process have direct access to the hardware read and write pointers. This would remove the need to switch from user land (where the application/driver which is generating the rendering command stream lives) to kernel land (where the driver has hardware access) to actually issue to DMA request. This transition can be very slow on some operating systems. So removing it is desirable, especially if the DMA request granularity is small and so the transition is a frequent occurance.

To this end, the registers for controlling the DMA buffers are split into two distinct regions. The kernel region contains all the setup registers that only the kernel land portion of the driver is allowed to touch. This regions should be mapped into memory such that only the kernel has access to it.

The second region is an array of sixteen read/write pointers and Command/Sync ids. Each set is placed in its own 4K page. The idea is that it should be possible to map the one specific page appropriate to an application into that application's address space in userland. A user land portion of the driver can then talk directly to the hardware in order to operate its own private DMA buffer.

This is a potentially dangerous thing to do because user land applications are not under the driver write's control and can have a tendancy to write random values to random locations. If such an error happens to hit the write pointer, then random data could get DMAd (and potentially from outside the DMA buffer which may not even be valid memory to read) and all sorts of nasty things could happen.

In order to prevent this, the write pointers have been fitted with a magic number field. The 'official' magic number is set by the kernel land driver at startup and is communicated to the user land driver when the a new context is created and handed over to the user land driver. If a write pointer update does not contain this magic number, then it will be discarded by the chip.

The odds are that any random error in the application itself will either not hit this one special location, or that it it does, it will not have the correct magic number in it. If the odds are not acceptable, of course, the driver can simply leave everything in kernel land only and just take the user -> kernel transition perforamnce hit.

3.6.5.1 Command Ids and Sync Ids:

These are two methods of synchronising the hardware with the host. The former is for keeping track of command DMA buffers, the latter for tracking actual rendering. Each context has its own private pair of ids which it is free to use at it likes. They are basically just user storage registers and whatever value is send in with the tag can be read back after the tag has been consumed by the appropriate unit.

The **CommandId** tag is consumed by the DMA engine and this is the last thing done in the DMA unit. So by the time a given CommandId value is put into the read back register and is visible to the driver, the driver knows that all commands in the DMA buffer before that id have been read. That is, the DMA itself has finished up to that point, the commands will not necessarily have been processed by any part of the graphics pipeline yet.

The Sync tag is consumed by the HostOut unit right at the back end of the graphics pipeline. This means that by the time a given sync value is read back by the driver, all rendering which was in the DMA buffer before the Sync tag will have been completed. The

Sync tag also has a the effect of causing all units to flush out any work they have batched up, sat in write caches, etc. Thus, when a sync value is read back by a given context, that context knows that all work it had queued up prior to the sync has fully completed.

3.7 Vertex and Index Buffers (GPIO)

The GPIO (Graphics Processor Input/Output) unit in P10 provides a very general mechanism for reading multiple streams of data into the P10 core. The obvious use for these data streams is to interpret them as vertex data (optionally indexed) and to pass that data on down the pipeline (to the Vertex Shader to be transformed and lit, for example). As an example of the generality of the GPIO unit, the indexing mechanism can also be used to perform palette expansion image downloads using 8 bit indices, where the “index” data is the source palette image and the “vertex” data is the palette. While the rest of the pipeline must play its part in providing the multitude of 3D processing functions on chip, the GPIO unit is the data gathering gateway to that on-chip functionality.

The principle use of the GPIO unit is to facilitate the drawing of 3D primitives. In particular, it is desirable to be able to render all 3D primitives without touching the index/vertex data on the host (providing the app and the OS/API allows that).

If the driver has to read the index/vertex data on the host in order to pre-process it or even just to copy it into a DMA buffer then substantial front-side bus bandwidth (in the host chipset) can be consumed and performance will be compromised. The GPIO unit assists in this matter by matching vertex data with tags according to a driver-supplied mapping. This mapping is commonly updated infrequently minimising managements overheads (the frequency is obviously app dependent but Direct3D apps tend to submit large batches of primitives because ISVs have been educated about the benefits of batching). An alternative would be for the driver to build a DMA buffer which contains tag/data pairs and send that to the chip - so the GPIO unit helps minimise the amount of data read by the chip as well as avoiding the driver reading the vertex data at all in a number of cases.

There are some situations in which the host cannot avoid touching the data e.g. OpenGL immediate mode where the app supplies the vertex data on the fly. Even in this case the GPIO unit can be useful – the driver can track the vertex data pattern, infer the vertex format, multi-buffer the vertex data in USWC memory and then issue GPIO requests to initiate rendering when a given threshold is reached. The advantage of this scheme is that the driver does not have to write tags and data into a DMA buffer – just the data is written.

The GPIO unit supports two forms of caching of vertex and index data. Data can be cached either in the unit itself or in a post-T+L form further down the pipeline. The latter method can only be used with indexed primitives. This is discussed further in Section 3.7.2, *Caching*.

3.7.1 Organizing data in memory

The AGP bus does not have enough bandwidth (even at 4x with 100% efficiency) to deliver all the vertex data that P10 can process (the AGP bus limit is approximately 32 million vertices/sec @ 32 bytes/vertex c.f. P10 vertex throughput of up to 50 million/sec). The bandwidth to P10 local video memory is much higher than AGP4x (16 times greater) and so P10 allows vertex data to be read from (and also wriiten to) local video memory, to take advantage of this higher bandwidth.

DirectX provides a mechanism for an application to allocate memory in which to store vertex data.- this memory is known as a vertex buffer. Indexed primitives require indices (up to 32 bits on P10, of which 24 are usable) and these can be stored in index buffers.

DirectX distinguishes two classes of vertex data – static and dynamic. Static vertex data is guaranteed to never be modified and so it is an excellent candidate for being placed in local video memory. Dynamic data may be modified and it is up to the driver to decide where best to place it. The vertex buffer could be in USWC system memory (allocated from the AGP heap, for example) where the write access for the app is fast but the transfer over the AGP bus is relatively slow. Alternatively, the vertex buffer could be allocated in local video memory. In the latter case, updating the vertex data will be relatively slow but the core will be able to read the data very quickly.

Index buffers can also be read from local video memory on P10, however the advantage of local video memory index buffers is less clear. Because the bandwidth requirements of index data are much lower than that of vertex data (say 6 bytes per triangle c.f. one new 32 byte vertex per triangle) it may be best to stream the index data over the AGP bus in parallel with vertex data reads over the local video memory bus.

3.7.2 Caching

The GPIO unit operates two forms of caching which improve vertex throughput. The first form of caching uses the address of a vertex as a key to avoid fetching a given vertex multiple times, thus reducing bus (either AGP or internal) traffic. The second form of caching uses the index as a key to avoid lighting and transforming a vertex multiple times. This second cache only operates for non-connected indexed primitives i.e indexed line lists and indexed triangle lists. Connected primitives, strips and fans, already have an implicit caching strategy which could in principle be augmented with index-keyed caching but for simplicity are not.

The address-keyed cache is invalidated when the enables for a buffer are changed . The index-keyed cache is enabled and invalidated via bits in the Begin command (see checklists in Drawing primitives below).

The index-keyed cache has 16 entries and so there is reasonable scope for an app to reorder its meshes to maximise the hit-rate in the cache. It is possible for a driver to re-order indexed meshes to improve the effectiveness of this cache but this is likely to only be a performance win for static meshes. The longer the program running in the Vertex Shader the larger the benefit of the index-keyed cache.

When caching is enabled it overrides the normal primitive assembly mechanism (which is usually transparent to the driver). Therefore the driver needs to re-synchronise the mechanism when switching between cached and non-cached rendering (see checklists in Drawing primitives below). In addition the current edge flag must be set.

3.7.3 Preparing to Draw Primitives

The GPIO unit can read data via up to two index buffers and from up to 16 data buffers. Up to 16 elements of 1, 2, 3 or 4 32-bit units can be defined. The GPIO unit can be programmed to assign a particular tag to each element or to skip an element if that particular element is not required to be processed by the pipeline. Skipped elements can arise during multi-pass rendering techniques when, for example, an app may generate a vertex with 12 sets of texture coordinates, reference eight of them in one pass and the remaining four in another pass. In addition to reading 2 index buffers automatically a

further 14 index buffers can be emulated by sending VertexIndex tags and data inline in the DMA stream.

It is useful to understand that the GPIO unit performs two operations simultaneously when processing vertex data. The first process gathers data from the enabled input data buffers (optionally indexed) in a round-robin fashion interleaving that data into one data stream inside the unit. The second process assigns meaning to vertex elements in that data stream by associating a tag with each element. As noted above it is possible to skip an element rather than associate a tag – the data for that element is discarded and is not seen by downstream units. There is no requirement to have the number of enabled data buffers match the number of parameters in the vertex – they are independent. For example, there may be two enabled data streams, one supplying two parameters (e.g. position and normal) and the other supplying three parameters (e.g. three sets of texture coordinates) to build a vertex with a total of five parameters.

The chip performs index checking as required by DirectX. Index checking ensures that a given index will not cause an out of range access into the corresponding data buffer.

So, to prepare for drawing primitives, it is necessary to define and enable the input buffers and define the association of tags to vertex elements. The following check-list expands on those steps:

- Send a **VertexDataBufferN** (N = [0-15]) command for each enabled data buffer – this is a 64-bit command that encodes the buffer address, the byte-swap mode, the data size and data stride for that buffer. The latter two fields are in 32-bit units minus one.
- Send a **VertexDataBufferEnable** command with a bit set for each enabled data buffer. This also invalidates the read-cache and may need to be issued for each primitive, depending on the location of the data.
- Send a **VertexIndexBufferN** (N = [0-2]) command for each enabled index buffer – this is a 64-bit command that encodes the buffer address, the byte-swap mode, the index size and an enable mask for each data buffer that should be indexed by the index buffer..
- Send a **VertexIndexBufferEnable** command with a bit set for each enabled index buffer.
- Send a **VertexIndexBounds** command to configure index checking – this is a 64-bit command that encodes a lower and upper bound. DirectX only supports one index buffer so there is only one copy of this register rather than the two you might expect.
- Send a VertexParameterMsgN (N = [0-15]) command for each parameter – this encodes the tag to associate with that parameter, the size of the parameter (in 32-bit units minus one) and whether to send or skip the parameter
- Send a **VertexParameterEnable** command with a bit set for each enabled parameter

3.7.4 Drawing Primitives

Checklist of the steps required to render a triangle fan in DirectX7.

- Send **Begin** command with TriangleFan primitive type
- Send **VertexDataBufferLookup** command. This is a 64-bit command containing the first vertex and number of vertices to read from the enabled data buffer(s)

- Send **End** command. This is used for closing line loops and triangle fans and so is not required for DirectX.

Checklist of the steps required to render an indexed triangle list in DirectX7.

- Send **VertexMachineState** command with data of 0 to reset the primitive assembly mechanism if using index-keyed caching.
- Send **EdgeFlag** command with data of the currentEdgeFlag to get edges drawn correctly when in wire-frame mode.
- Send **Begin** command with Triangle primitive type and the cache Invalidate and Enable bits set if index-keyed caching is required.
- Send **VertexIndexBufferLookup** command. This is a 64-bit command containing the first index and number of indices to read from the enabled index buffer(s). If using non 32-bit indices the start value can be used to offset the index lookup to cope with a non 32-bit aligned index buffer.
- Send **End** command. This is used for closing line loops and triangle fans and so is not required for DirectX.

3.8 Downloading Textures

Texture map downloads need to write the texture data into a memory location from which the core can subsequently access the data during rasterisation. This data can either be directly into video memory using the host or it can be downloaded through the core as per pixel data downloads For details refer to section 7.2, "Texture maps (download, MIPmap generation)"

3.9 DXVA Driver

Hardware support for video playback is a desirable component of a graphics accelerator. There is a family of video encode/decode standards known as MPEG which have become widely accepted. DirectX Video Acceleration (DXVA) is Microsoft's API to enable access to hardware accelerated MPEG decoding. There are multiple levels of DXVA acceleration ranging from simple motion compensation to performing IDCT (inverse discrete cosine transform) processing and raw bit-stream decompression.

In principle, P10 can support motion compensation and IDCT processing efficiently whereas the bit-stream decompression does not map well onto any of the various SIMD processors in the graphics core. Motion compensation (mocomp) is a form of multi-texturing whereas the IDCT could be implemented using the Vertex Shader unit. Due to the high performance of current host processors a hardware accelerated mocomp-only solution (MPEG 2) is sufficient for acceptable performance. Chapter 8 discusses the design of a HW mocomp accelerator for P10.

3.10 Video Port

The Video Port Unit – VPU – implements a VESA Video Interface Port (VIP) Version 2 Level II video port master.

The VPU supports the following:

- ITU-R BT.656 video stream – 8-bit @ 27MHz
- VIP1.1 video port – 8-bit @ 27Mhz
- VIP2 Level I video port – 8-bit @ 75MHz
- Proprietary VIP2 video port – 8-bit @ 150MHz

The VPU does not support the following:

- VIP1.1 or VIP2 host port
- VIP2 Level II video port – 16-bit @ 75MHz
- VIP2 Level III video port – 8-bit @ 75MHz input + 8-bit @ 80MHz output

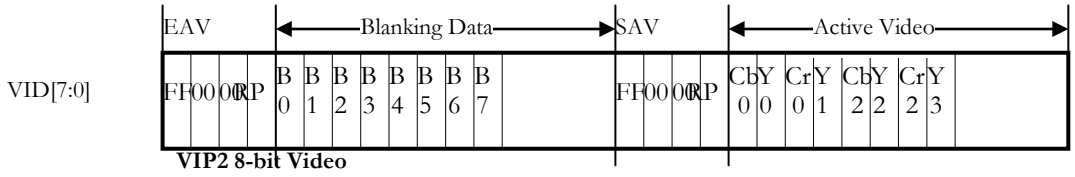
Familiarity with the following standards/recommendations is assumed:

- VESA Video Interface Port Standard, Version 2
- Recommendation ITU-R BT.656, Interfaces For Digital Component Video Signals In 525-Line And 625-Line Television Systems Operating At The 4:2:2 Level Of Recommendation ITU-R BT.601 (Part A)
- Recommendation ITU-R BT.601, Studio Encoding Parameters Of Digital Television For Standard 4:3 And Wide-Screen 16:9 Aspect Ratios

3.10.1 Video Stream Formats

Active video is formatted as 4:2:2 YCrCb, and is transmitted as a byte stream of Cb-Y-Cr-Y. The conversion of 4:2:2 YCrCb to RGB is described in ITU-R BT.601.

The VIP2 Level I video port provides 8-bit samples; control codes are interleaved with the data:



VIP2 8-bit Video

3.10.1.1 Empty Cycles

In VIP2, skip data (“00”) during active video is used to mark an empty cycle. In the VPU, empty cycles are optionally discarded. If the video stream is known to contain empty cycles, “00” bytes should be discarded. If the video stream conforms to ITU-R BT.656, or is known to contain out-of-range values, “00” bytes should be kept.

3.10.1.2 Blanking Data

VIP2 supports the transport of data other than video as ancillary data blocks during horizontal or vertical blanking intervals only.

The VPU does not interpret blanking data in any way. Blanking data is optionally stored in memory if needed by host software, otherwise it should be discarded to save memory bandwidth.

3.10.2 SAV and EAV Timing Reference Signals

The SAV and EAV timing reference signals are 4-byte sequences defined in ITU-R BT.656 and extended in VIP1.1 and VIP2.

The first 3 bytes are the fixed preamble “FF”, “00”, “00”.

The 4th byte is the reference byte (RP):

- Bit 7 = Task (T) bit. ‘1’ = Task A, ‘0’ = Task B. In ITU-R BT.656 this is fixed to ‘1’. In VIP1.1 and VIP2 this is used to distinguish two different tasks. In the VPU this is used to distinguish two different destinations in memory and frame interrupts (see later). The two tasks can be interleaved in Line or Field mode.
- Bit 6 = Field (F) bit. ‘0’ = Field 1, ‘1’ = Field 2.
- Bit 5 = Vertical blanking (V) bit. ‘0’ = active video, ‘1’ = vertical blanking interval.
- Bit 4 = Horizontal blanking (H) bit. ‘0’ = active video (in SAV), ‘1’ = horizontal blanking interval (in EAV).
- Bits 3:0 vary. In ITU-R BT.656 and VIP1.1 they are protection (P[3:0]) bits. In VIP2 they contain new video flags. In the VPU they are ignored.

The VPU uses the H bit from the SAV and EAV to control horizontal state, and the V, F and T bits from the EAV only to control vertical state.

3.10.3 DTV Display Formats

VIP2 DTV display formats are specified by the position of the SAV and EAV. These in turn identify the task, field, and blanking intervals.

In the VPU, horizontal samples are counted from 0 at the start of the horizontal blanking interval, and the first line of the frame is the first line of the vertical blanking interval. In interlaced video, with two vertical blanking intervals, the start field is also specified.

For example, this is how the 525-line System is formatted for the VPU:

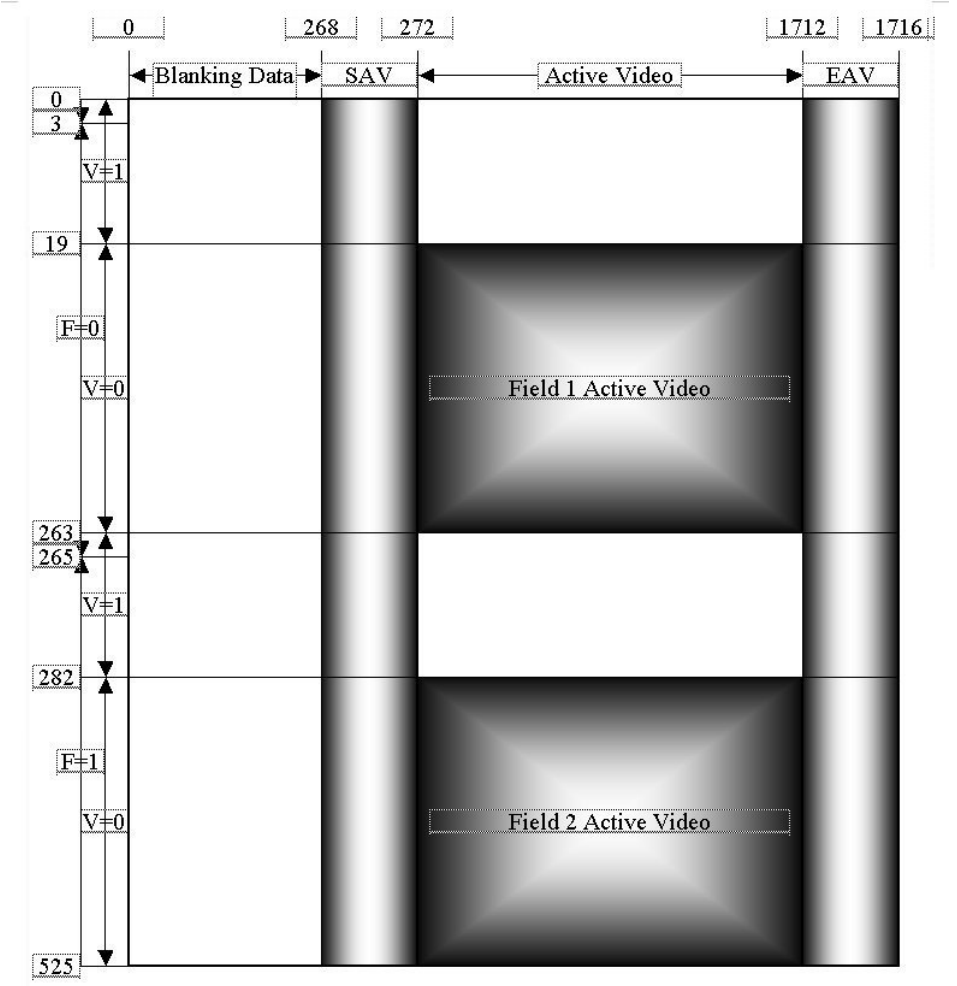


Figure 3.10 SAV and EAV Paramters for 525-line VPU

The positions of the SAV and EAV, the interlaced flag, and the start field needed to program the VPU for the example VIP2 DTV display formats are as follows:

DTV Display Format	SAV Pos	EAV Pos	Interlaced	Start Field
525-line System (720×487×30Hz)	268	1712	1	1
625-line System (720×576×25Hz)	280	1724	1	1
480P (704×480×60Hz)	300	1712	0	-
720I (1280×720×30Hz)	362	1646	1	1
720P (1280×720×60Hz)	362	1646	0	-
1080I (1920×1080×30Hz)	272	2196	1	1

Note: The SAV position is the width of the horizontal blanking interval. The EAV position + 4 is the width of the frame.

3.10.4 Fields and Frame

Frames are stored in individual buffers in memory. For non-interlaced video, the EAV Field (F) bit is ignored. For interlaced video, the EAV Field (F) bit is matched against a start field to determine the 1st field in the frame.

An interlaced video source can be stored as non-interlaced frames. This might be used to de-interlace video. A non-interlaced video source must never be stored as interlaced frames, because the EAV Field (F) bit never changes and the next frame will never be found.

3.10.5 Frames and Memory

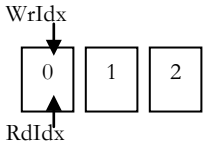
The VPU can store frames in 1, 2 or 3 buffers per task. Host software provides up to 3 buffer addresses per task, and the VPU cycles through the buffers in turn. Triple buffering allows the VPU to cope with mismatched input and output frame rates.

An indexing scheme allows the chip and host to signal which buffer each is currently processing. The chip maintains a 2-bit write index (WrIdx) per task, which indicates which buffer is currently being written. The host maintains a 2-bit read index (RdIdx) per task, which indicates which buffer is currently being read. When each side has finished processing its current buffer, it advances its index to the successive buffer.

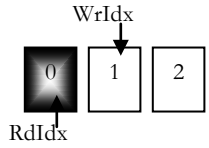
The protocol for their correct use is that each side must not reach the other's index. If either side would otherwise reach the other's index, it must preserve its current index and re-use the current buffer.

The maximum index is set by the host. The chip will wrap its index from the maximum back to 0, and the host should do likewise. If the host sets its read index past the maximum, the chip will continually write frames without ever waiting for the host. This might be useful in some diagnostic applications.

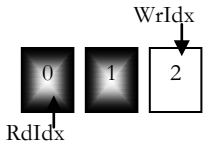
For example, this is how triple buffering might look:



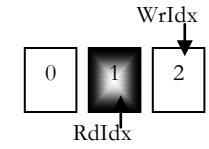
1. At reset



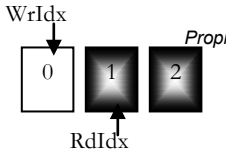
2. After chip write



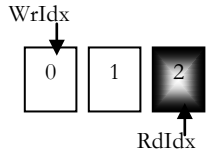
3. After chip write



4. After host read



5. After chip write



6. After host read

3.10.6 Frame Interrupts

As well as advancing its write index, the chip will also generate a frame interrupt when it has finished processing its current frame. The interrupt is generated even if the chip could not advance its index to the successive buffer. Host software detects dropped frames in its interrupt handler by finding the write index unchanged.

3.10.7 Programming Summary

Host software initiates the video port as follows:

1. Start IClk, by initiating the external video source.
2. Initialise VPUIClk, by writing the following registers:
   - Mode
   - SAVPos & EAVPos
   - BufAddr[0..1][0..2]
   - RdIdx
3. Take VPUIClk out of reset, by writing the Enable register.

Host software terminates the video port as follows:

1. Force VPUIClk into reset, by writing the Enable register.
2. Stop IClk, by terminating the external video source.

3.10.8 Programming Example

This is an example of how to initialise and terminate the VPU for the 525-line System.

The assumptions are that:

- Only Task A (‘1’) is present.
- The input and output frame rates are mismatched, so triple buffering is used.
- Host software will perform de-interlacing, so the video source is stored as non-interlaced frames.

```
struct Mode
{
    unsigned            : 1;
    unsigned XVideo     : 1;
    unsigned SwapData   : 1;
    unsigned SkipData   : 1;
    unsigned HBStore    : 1;
    unsigned VBStore    : 1;
    unsigned Interlaced : 1;
    unsigned StartField : 1;
    unsigned MaxIdx     : 2;
    unsigned            : 22;
```

```
    Mode& operator=(const unsigned i) { *((unsigned *)this) = i; return (*this); }
    operator unsigned(void) const { return (*((unsigned *)this)); }
};

extern void VPUWrite(
const unsigned Addr,
const unsigned Data);

void Init525(
const uint25 BufAddr)
{
        Mode M;
        const unsigned SAVPos = 268;
        const unsigned EAVPos = 1712;
        const unsigned Stride = (((EAVPos+4)+63)/64);
        uint25 BufAddr0, BufAddr1, BufAddr2;
        unsigned Data;

        //
        //      Initialise mode.
        //
        M = 0;
        M.XVideo = 0;           // 8-bit video
        M.SwapData = 0;         // (only applies to 16-bit video)
        M.SkipData = 1;         // Discard empty cycles
        M.HBStore = 1;          // Store horizontal blanking data
        M.VBStore = 1;          // Store vertical blanking data
        M.Interlaced = 0;       // Store video source as non-interlaced
       frames
        M.StartField = 0;       // (only applies to interlaced frames)
        M.MaxIdx = 2;           // Triple buffering
VPUWrite(eMode, M);

    //
    //    Initialise SAV/EAV positions.
    //
    VPUWrite(eSAVPos, SAVPos);
    VPUWrite(eEAVPos, EAVPos);

    //
    //    Initialise buffer 0, 1 & 2 addresses.
    //
    //    The 25-bit buffer address is a 28-bit tile address
    //    aligned to an 8-tile boundary (VPU has linear access to memory).
    //    525-line System Field 1 is 263 lines and Field 2 is 262 lines,
    //    so each buffer holds 263 lines rounded up to an 8-tile boundary.
    //
    BufAddr0 = BufAddr;
    BufAddr1 = BufAddr0 + (Stride * ((263+7)/8));
    BufAddr2 = BufAddr1 + (Stride * ((263+7)/8));
    VPUWrite(eBufAddr10, BufAddr0);
```

```
    VPUWrite(eBufAddr11, BufAddr1);
    VPUWrite(eBufAddr12, BufAddr2);

    //
    //    Initialise read index.
    //
    VPUWrite(eRdIdx, (0<<2));

    //
    //    Take VPUIClk out of reset.
    //
    VPUWrite(eEnable, 1);
}

void Term525(void)
{
    //
    //    Force VPUIClk into reset.
    //
    VPUWrite(eEnable, 0);
}
```

3.10.9 Register Interface

The 4-Kbyte region defines 32-bit registers on 64-bit boundaries as follows:

Offset	Name	Format
0x000	Enable	Enable
0x008	Mode	Mode
0x010	SAVPos	SAVPos & EAVPos
0x018	EAVPos	SAVPos & EAVPos
0x020	BufAddr00 (task 0, index 0)	BufAddr[0..1][0..2]
0x028	BufAddr01 (task 0, index 1)	BufAddr[0..1][0..2]
0x030	BufAddr02 (task 0, index 2)	BufAddr[0..1][0..2]
0x038	BufAddr10 (task 1, index 0)	BufAddr[0..1][0..2]
0x040	BufAddr11 (task 1, index 1)	BufAddr[0..1][0..2]
0x048	BufAddr12 (task 1, index 2)	BufAddr[0..1][0..2]
0x050	RdIdx	RdIdx
0x058	WrIdx	WrIdx

3.10.9.1 Enable

Reset value = 0.

Read-write access.

Offset	Size	Name	Description
0	1	Enable	When 0, VPUIClk is forced into reset. When 1, VPUIClk is taken out of reset.
1	31	-	Reserved.

3.10.9.2 Mode

Reset value = undefined.

Read-write access.

Offset	Size	Name	Description
0	1	-	Reserved.
1	1	XVideo	0 = 8-bit video. 1 = 16-bit video.
2	1	SwapData	0 = In 16-bit video, don't swap the Vid and XVid bytes during active video. 1 = In 16-bit video, swap the Vid and XVid bytes during active video.
3	1	SkipData	0 = Empty cycles during active video are processed. 1 = Empty cycles during active video are discarded.
4	1	HBStore	0 = Horizontal blanking data is discarded. 1 = Horizontal blanking data is stored.
5	1	VBStore	0 = Vertical blanking data is discarded. 1 = Vertical blanking data is stored.

6	1	Interlaced	0 = Store video source as non-interlaced frames. The video source can be non-interlaced or interlaced. 1 = Store video source as interlaced frames. The video source must be interlaced.
7	1	StartField	In interlaced video, this is matched against the EAV Field (F) bit to determine the 1st field in the frame.
8	2	MaxIdx	Maximum index: 0 = 1 buffer 1 = 2 buffers 2 = 3 buffers 3 = Reserved (3 buffers)
10	22	-	Reserved.

3.10.9.3 SAVPos & EAVPos

Reset value = undefined.

Read-write access.

Offset	Size	Name	Description
0	12	Pos	SAV or EAV position, counted from 0 at the start of the horizontal blanking interval.
12	20	-	Reserved.

3.10.9.4 BufAddr[0..1][0..2]

Reset value = undefined.

Read-write access.

Offset	Size	Name	Description
0	25	BufAddr	Buffer address aligned to an 8-tile boundary.
25	7	-	Reserved.

3.10.9.5 RdIdx

Reset value = undefined.

Read-write access.

Offset	Size	Name	Description
0	2	RdIdx0	Read index (task 0).
2	2	RdIdx1	Read index (task 1).
4	28	-	Reserved.

3.10.9.6 WrIdx

Reset value = undefined at chip reset, 0 at VPUIClk reset.

Read-only access.

Offset	Size	Name	Description
0	2	WrIdx0	Write index (task 0).
2	2	WrIdx1	Write index (task 1).
4	28	-	Reserved.

3.11 Upload Facilities

The upload facilities in P10 are provided by the GPIO Upload DMA Unit, in conjunction with the Host Out Unit and Pixel Unit.

The Upload DMA Unit can receive arbitrary pixel data from the Pixel Unit and packs it as 8-bit, 16-bit, 24-bit or 32-bit data for transfer to the host by the bus master. Data can be byte-swapped in any of four different methods as appropriate for the host. Data less than 8 bits in width must be packed by the graphics pipeline prior to reaching this unit.

The Host Out Unit allows upload data to be filtered out of the data stream before reaching the bus master.

4

Programming Overview

Programming P10 consists of setting up mode and state registers and defining the programs to run in a handful of programmable units. P10 uses fixed registers for specialised tasks where the need for dedicated functionality and efficiency outweighs the need for flexibility. These tend to be grouped into functional units of state registers controlled by one or two mode registers.

The principal functional groups, both fixed and programmable, and controlling registers, are described in the *Miranda P10 Reference Guide Volume I.* The sections following are concerned with programmability aspects only.

The graphics pipeline can be divided into two major sections which are mapped into the P10 hardware as shown in the block diagrams in the *Glint P10 Reference Guide volume I*, and described below. The major parts of the process are the

- **Transform and Lighting** group, and
- **Texture and Render** group.

4.1 Transformation and Lighting

4.1.1 GPIO

Also known as the Current Parameter Unit, This subsystem reads incoming commands and data on the input FIFO and implements various Vertex index, array and caching facilities, e.g. *glDrawElements()*, *glArrayElement()* and *glDrawArrays()*. The principal registers are **VertexParameter***, **VertexData***, **VertexIndexBuffer*** and **VertexCacheMode.** The group's main task is to allow a parameter such as a colour or a texture to be supplied for every vertex even when it is not included in a DMA buffer. This allows vertices in OpenGL to inherit previously defined parameters without being forced to supply them on every vertex.

4.1.2 Vertex Shading Unit

The Vertex Shading program:

- takes the typeless input parameters supplied for each vertex,
- combines them in some way with constant data (i.e. a transformation matrix, lighting parameters, etc.) ,
- writes the results to typed registers.

The Vertex Shader knows only about the vertex it is processing, so any operations which need topological information such as backface culling or clipping are not handled here.

Input data is ***typeless*** as the meaning is defined by the program. However the output registers are ***typed*** because a coordinate is processed differently to a texture or colour value, for example, in the rasteriser.

Output data includes:

- output coordinates which have been projected and viewport mapped and
- texture coordinates which have been divided through by the homogeneous w value.

The texture coordinates are written to the rasteriser as eight 4-component vectors (Vec4) and the interpretation of these 32 values is a function of the Texture Coordinate program.

Similarly colours are written to the rasteriser as eight 4 component vectors and the interpretation of these 32 values is a function of the Shading program.

The names “texture coordinates” and “colors” are misleading as they are not restricted to representing this type of data, but could be used to hold a fog value, Fresnel term, reflection vector, etc. The difference between the two data types are summarized in the following table:

Function	Texture Coordinates	Colors
Perspective correct interpolation	Yes	no
Clamped	No	yes in the range 0…1
Plane equation evaluation	floating point	fixed point
Index texture map	Yes	no

The coordinates are forwarded to the *Vertex Machine Unit* which monitors the incoming vertex data and issues geometry cammands to trigger primitive assembly. *Primitive Assembly* (Culling, Clipping) is handled by fixed function units described later. The Cull Unit caches the window coordinates for the 16 vertices and when a **Geom*** command arrives it uses the cached window coordinates to test clip against the viewing frustum and, for triangles, do a back face test.

The result is fed to the *Geometry Unit* for geometric clipping and rendering setup. The output is a series of triangles which are passed to the *Context Unit* and *Router* to be allocated to a texture pipe and/or *Depth and Stencil Test* unit.

4.2 Texture and Rendering

The Texture and Rendering functional group includes Texture Pipes, Depth and Stencil testing and Pixel Addressing facilities.

Each Texture Pipe contains a programmable *Texture Coordinate Unit*, a fixed-function *Texture Filter* and a programmable *Shader Unit*

4.2.1 Texture Coordinate Programme

Texturing is handled by a texture coordinate program. The inputs to the program are the 32 plane equations derived from the 32 texture coordinate values generated by the Vertex Shading program and 32 floating point values in the global registers. All computation is done using single precision floating point numbers.

The fragment's screen position is used together with the plane equations to compute the texture coordinate this fragment needs to access. The perspective division and level of detail calculation are also done and the results written to four output registers (which hold u, v, w, lod) to be used by the texture lookup and filtering hardware. Once these registers have been loaded the texture lookup and filtering is initiated via a command register and the results of the lookup and filtering operation are loaded into the texel registers in the Shading program. This process is repeated for all the textures used to colour a fragment and once the filtered texels have been loaded into the Shading program the Shading program will be run.

Computed values can be passed directly to the Shading program (to load up the texel registers) without causing a texture look up. The filtered texel value can also be returned to the texture coordinate program so feedback or dependent texture operations can be implemented (i.e. bump mapping).

Fragments can be terminated under program control.

Texture Lookups and Filtering are handled by fixed function units described later.

4.2.2 Shading

This is handled by a shading program. The inputs to the program are the 32 plane equations derived from the 32 colour values generated by the Vertex Shading program, the eight 32 bit texel values supplied via the texture coordinate program and 32 byte values stored in global registers. All the necessary input data is present when the program is started.

All computations are done using signed 4.8 fixed point numbers and any 8 bit values (such as a texel colour component) are converted using several different mappings. The results of a program run are written into the fragment register in the Pixel program. Typically 4 bytes are written into the fragment register representing R, G, B and A colour components, however up to 8 bytes of data can be written and it can be used in any way the Pixel program sees fit.

The shading program calculates the final colour of a fragment so will usually include Gouraud shading, texture application, fog and alpha testing.

4.2.3 Framebuffer Processing

The framebuffer processing is done in the Pixel program. The Pixel program takes the fragment colour calculated by the Shading program, data read from the framebuffer and 32 byte values stored in global registers and calculates the colour (or some other) value to write back into the framebuffer.

All calculations are done using an 8 bit ALU, one component at a time and will normally include such tasks as dithering, logical ops, alpha blending and plane masking.

The addresses of the framebuffer data to read and write is also under program control to allow a wide variety of addressing patterns to be implemented (e.g. convolution, multi sample AA, pattern filling, etc.).

All the programs are written to process a single vertex, fragment or pixel at a time and some general conventions must be adopted so the data flowing from one program to the next is assigned some consistent meaning.

The pixel and fragment programs tend to be quite short, especially the Pixel programs which are typically only 4 or so instructions long. The vertex programs tend to be the longest. All the handshaking and protocols between the programs (or indeed with the fixed function operations) are handled automatically and watchdog timers will prevent an erroneous program from locking up the chip.

4.2.4 How to draw a Gouraud-shaded triangle

For an example of typical Gouraud shading programs see the discussion of Gouraud Shading in section 8.

4.3 Fixed mode and state registers

Refer to the *Miranda P10 Reference Guide, volume I*

4.4 Programmable Units

4.4.1 Data flows among units

4.4.2 Vertex Shading Introduction

The Vertex Shading Unit (VSU) is responsible for all vertex level processing in the chip. This normally consists of transformation, vertex lighting and texture coordinate generation, but with the advent of DX8 and OpenGL's vertex program extensions may also include more exotic and user-defined operations, such as vertex blending, tessellation, skinning, etc.

The VSU is comprised of an array of Vertex Processors (VPs), each of which contain an ALU and work on a single vertex at a time. Each VP has access to an input buffer, an output buffer and internal storage containing all the data necessary to operate on that vertex.

4.4.2.1 Input Data

The VSU has 16 Vec4 (vectors consisting of 4 32-bit floating point values) input parameters. These parameters are typeless – that is their significance is defined by the program run in the VSU, rather than the name given to them.

Any number of these parameters may be loaded per vertex, and in any order, with the only restriction being that a defined trigger parameter is sent for each vertex. This trigger parameter is the switch that closes off the current vertex and starts execution of the VSU program using the input data already received. The trigger parameter again is typeless, but is typically the vertex position.

Typeless parameters for vertices, trigger parameter, inheritance, 16 vec 4s. [16 4-component vectors, typeless because their significance is defined by the invoking program rather than absolute position. Trigger param. User-defined, not necessarily Position as in OpenGL.]

4.4.2.2 Output Data

The VSU has 17 Vec4 output parameters, the first 8 are loosely defined as ColourA-H, the next as TextureA-H, with the Colour* parameters being passed downstream to the

Shading Unit, and the Texture* parameters being passed to the Texture Coordinate Unit. Again, these parameters are really typeless, with their significance being defined by the receiving unit.

The final output parameter contains the window coordinate parameter. This Vec4 contains the viewport-transformed window coordinate in the first 3 values and the homogenous W value in the 4th. Every VSU program should write this value, but there is no such requirement to output any of the other parameters.

4.4.2.3 Typed parameters

4.4.2.4 Special parameters

As well as the standard output parameters, the VSU outputs may be defined as special parameters. These outputs can then be interpreted as point sizes, eye-space vertex coordinates and user-defined clip plane outcodes by the rest of the geometry pipeline

4.4.2.5 Memory

The VSU has four types of internal storage; the first is instruction memory – enough for 256 instructions. Next is 256 Vec4 (or 1024 floats) of constant, read-only Coefficient Memory shared between the VPs. Finally, each VP has a 16 Vec4 scratch pad of read-write memory and an address register which can be used for indirect addressing into the coefficient memory.

4.4.2.6 Programs act as transfer functions

A VSU program consists of a list of instructions telling the VPs how to operate. The program is executed by all VPs in parallel and begins when all of the VPs are ready to run. This usually means that all of them have received a trigger parameter, but execution may also be triggered by new state data (new program or coefficient data), by a synchronisation tag or if the output MFIFO is full.

Execution is controlled by a sequencer, which reads each instruction and broadcasts it to all VPs. The sequencer will continue to operate until it reaches a *Stop* instruction or until the watchdog timer of 16K instructions is reached – although the VSU only has 256 instruction registers, as loops and sub-routines are supported there would, without the watchdog, be potential for an invalid program to be loaded and lock-up the chip.

4.4.2.7 Instruction Set Summary

Each instruction is broken up into fields, defining the operation, the location of the input data and where to put the results. There are three possible inputs for each instruction, but only one from the input buffers, one from the coefficient memory and two from the scratch pad may be defined for a single instruction. The output may be the scratch registers, the output buffers or the address register.

The following is a table of the instruction format:

Bit No.	Name	Width	Description
0..4	OpCode	5	This field holds the ALU operation. See later for a description.

Bit No.	Name	Width	Description
5..6	VectorCount	2	This field holds the number of components in the vector. The options are: 0 = one component vector (i.e. a scalar) 1 = two component vector 2 = three component vector 3 = four component vector
7..16	CoeffAddr	10	This field selects the float to read from the coefficient memory. The address is modified by the CoeffAddrBase and CoeffDataType fields.
17..22	InVertexAddr	6	This field selects the float to read from the input vertex registers. The address is modified by the InVertexAddrBase and InVertexDataType fields.
23..28	ScrAddrA	6	This field selects the float to read from the scratch register file. This value is srcA data. The address is modified by the ScrAddrBaseA and ScrDataTypeA fields.
29..34	ScrAddrB	6	This field selects the float to read from the scratch register file. This value is srcB data. The address is modified by the ScrAddrBaseB and ScrDataTypeB fields.
35..36	ArgA	2	This field selects the argA input to the ALU. The options are: 0 = coeff data 1 = input vertex data 2 = srcA from the scratch register file 3 = srcB from the scratch register file
37..38	ArgB	2	This field selects the argB input to the ALU. The options are: 0 = coeff data 1 = input vertex data 2 = srcA from the scratch register file 3 = srcB from the scratch register file
39..40	ArgC	2	This field selects the argC input to the ALU. The options are: 0 = coeff data 1 = input vertex data 2 = srcA from the scratch register file 3 = srcB from the scratch register file
41..42	ModA	2	This field defines how argA is modified before going into the ALU. The options are: 0 = pass 1 = negate 2 = absolute 3 = clamp to zero if negative

Bit No.	Name	Width	Description
43..44	ModB	2	This field defines how argB is modified before going into the ALU. The options are: 0 = pass 1 = negate 2 = absolute 3 = clamp to zero if negative
45..46	CoeffAddrBase	2	This field defines how the coefficient address is generated. The options are: 0 = relative (i.e. base + CoeffAddr) 1 = absolute (i.e. CoeffAddr) 2 = indirect (i.e. addressReg + CoeffAddr) 3 = circular addr = coeffBase + coeffAddr if (addr > coeffEnd) addr = coeffOrigin + addr – coeffEnd
47	CoeffDataType	1	This field defines the data type. The options are: 0 = scalar 1 = vector
48	InVertexAddrBase	1	This field defines how the input vertex address is generated. The options are: 0 = relative (i.e. base + InVertexAddr) 1 = absolute (i.e. InVertexAddr)
49	InVertexDataType	1	This field defines the data type. The options are: 0 = scalar 1 = vector
50	SrcAddrBaseA	1	This field defines how the scratch regiseter A address is generated. The options are: 0 = relative (i.e. base + ScrAddrA) 1 = absolute (i.e. ScrAddrA)
51	ScrDataTypeA	1	This field defines the data type. The options are: 0 = scalar 1 = vector
52	SrcAddrBaseB	1	This field defines how the scratch regiseter B address is generated. The options are: 0 = relative (i.e. base + ScrAddrB) 1 = absolute (i.e. ScrAddrB)
53	ScrDataTypeB	1	This field defines the data type. The options are: 0 = scalar 1 = vector

Bit No.	Name	Width	Description
54..61	DestAddr	8	This field holds the address to update with the results of an ALU operation. The address (after modification by the DestAddrBase) is decoded into the following ranges: 0…63 = scratch register 64…95 = ColourA[r, g, b, a]…ColourH[r, g, b, a] 96…127 = TextureCoordA[s, t, r, q]…TextureCooredH[s, t, r, q] 128…130 = window coordinate 131 = homogenous W 132 = address register 133…256 = no write Note the ColourA…ColourH parameters are automatically clamped when used downstream. The interpretation of the ColourA…ColourH and TextureCoordA…TextureCoordH values is down to the programs running in the Texture Coordinate Unit and the Shading Unit.
62	DestAddrBase	1	This field defines how the destination address is generated. The options are: 0 = relative (i.e. base + DestAddr) 1 = absolute (i.e. DestAddr)
63	DestDataType	1	This field defines the data type. The options are: 0 = scalar 1 = vector
64..67	Sequencer	4	This field holds the sequencer operation. See later for a description.
68..76	SeqData	9	This field holds data mainly for sequencer related operations such as jump or subroutine addresses, loop counter values. It can also supply a value to be loaded or added to the base registers. Instruction addresses can be absolute (0) or relative (1) and the most significant bit controls this.

The ALU has 22 instructions as follows (d is destination, s0, s1 and s2 are the three sources):

Value	Name	Time	Description
0	Move	1	d = s0
1	Add	2	d = s0 + s1
2	MAdd	4	d = s0 * s1 + s2
3	Mul	2	d = s0 * s1
4	Min	2	d = Min (s0, s1)
5	Max	2	d = Max (s0, s1)

Value	Name	Time	Description
6	SLT	2	if (s0 < s1) d = 1.0 else d = 0.0
7	SGE	2	if (s0 >= s1) d = 1.0 else d = 0.0
8	Fract	2	d = fractional part of s0
9	Trunc	1	d = integer part of s0 (as a floating point number)
10	Dot	2/4	d = s0 * s1 for first component, else d = s0 * s1 + s2
11	ShiftSign	1	d = s0 << 1 | s1.sign allows user clip outcode to be build up
12	Recip	9	d = 1.0 / s0, returns maximum positive number if s0 = 0.0
13	Div	9	d = s1 / s0, returns maximum positive or negative number if s0 = 0.0
14	RSqrt	1	d = 1.0 / sqrt (s0) (10 bits precision)
15	ALog	2	d = 2^{s0} (10 bits precision)
16	Log	3	d = $\log_2$ (s0) (10 bit precision)
17	Exponent	3	d = IntToFloat (s0.e - 127)
18	Mantissa	3	d = IntToFloat (1.0 + s0.m)
19	IntToFloat	3	d = IntToFloat (s0)
20	FloatToInt	3	d = FloatToInt (s0)
21	HRecip	8	d = 1.0 / s0, returns 1.0 if |s0| < epsilon. epsilon = 2^{-120}

Each instruction takes a single cycle to execute however listed above are times associated with each instruction, which is the number of cycles before the result is available to subsequent instructions. Additional dependencies exist in that a Div, or Recip instruction will occupy the Multiplier for 6 cycles and an Add has to wait until after a MAdd has completed.

The Sequencer has 13 instructions as follows:

Value	Name	Description
0	Inc	The next sequencer address is the current sequencer address + 1.
1	Jump	The next sequencer address is the address in the seqData field. The address in the seqData field is an absolute address if the most significant bit is clear, or a relative address if it is set.
2 3	Loop0 Loop1	The loop counter 0 or 1 is loaded with the contents of the seqData field. The maximum loop count is 127 and is primarily intended for looping around lights. The next sequencer address is the current sequencer address + 1.
4 5	DJNZ0 DJNZ1	The loop counter 0 or 1 is decremented and if the result is zero the next sequencer address is the current sequencer address + 1, otherwise the address in the seqData field is used as the next sequencer address. The seqData address can be absolute or relative.
6	Call	The current address + 1 is pushed on to the return stack and the next sequencer address is the address in the seqData field. The stack is only four deep and there is no protection against overflow. The seqData address can be absolute or relative.

Value	Name	Description
7	Return	The next sequencer address is taken from the return stack and the stack is popped. The stack is only four deep and there is no protection against underflow.
8	Stop	This terminates the program and implements the necessary handshaking to accept more vertex data and pass any results into the pipeline for culling and clipping.
9	IncCoeffBaseReg	The coeff address register used for relative addressing has the sequencer data field added to it. The sequencer data is sign extended before the addition. The next sequencer address is the current sequencer address + 1.
10	LoadCoeffBaseReg	The coeff address register used for relative addressing has the sequencer data field loaded into it. The sequencer data is first multiplied by 2 before loading. The next sequencer address is the current sequencer address + 1.
11	LoadCoeffOriginReg	The coeff origin register used for circular addressing has the sequencer data field loaded into it. The sequencer data is first multiplied by 2 before loading. The next sequencer address is the current sequencer address + 1.
12	LoadCoeffEndReg	The coeff end register used for circular addressing has the sequencer data field loaded into it. The sequencer data is first multiplied by 2 before loading. The next sequencer address is the current sequencer address + 1.

The ALU is a scalar processor, and addresses the memory locations in terms of floats (rather than Vec4s), vector operations are performed simply by setting the vector count field in the instruction and any relevant DataType fields, e.g.:

Reg[4+] = Mul3(Coeff[36+], A[8]);

…multiplies the values in coefficient 36, 37 and 38 by the value in scratch register 8 and places the results in scratch registers 4, 5 and 6. This instruction is really 3 instructions, controlled by the sequencer and will take the same amount of time to run as three separate Mul instructions.

4.4.2.8 Vertex Transformation

There are four basic transformations. They are:

Transforming by a 4x4 matrix (e.g. by the ModelView matrix to get the eye-space coordinate of the vertex or by the ModelViewProjection matrix to get the projected vertex):

```
reg[eyeVertex+] = Mul4 (in[pos_x], coeff[MVMat0+]);
reg[eyeVertex+] = MAdd4 (in[pos_y], coeff[MVMat4+],
reg[eyeVertex+]);
reg[eyeVertex+] = MAdd4 (in[pos_z], coeff[MVMat8+],
reg[eyeVertex+]);
```

```
reg[eyeVertex+] = MAdd4 (in[pos_w], coeff[MVMat12+],
reg[eyeVertex+]);
```

The 3-space eye vertex is required for local lighting and is derived by:

```
reg[oneOverEyeW] = Recip (reg[eyeVertex_w]);
reg[eyeVertex3+] = Mul3 (reg[eyeVertex+],
reg[oneOverEyeW]);
```

Similarly the projected vertex is the input position multiplied by the ModelViewProjection matrix. The 1/w value can be found from the projected vertex by:

```
reg[oneOverW] = HRecip (reg[projVertex_w]);
```

(see later why *HRecip* is used instead of *Recip*)

Transforming the normal by the Normal matrix. This is done the by code:

```
reg[normal+) = Mul3 (in[normal_x], coeff[NMat0+]);
reg[normal+) = MAdd3 (in[normal_y], coeff[NMat3+],
reg[normal+]);
reg[normal+) = MAdd3 (in[normal_z], coeff[NMat6+],
reg[normal+]);
```

If the normal needs to be normalised then the code to do this is:

```
reg[magSquared] = Dot3 (reg[normal+], reg[normal+]);
reg[invMag] = RSqrt (reg[magSquared]);
reg[normal+] = Mul3 (reg[normal+], reg[invMag]);
```

The ***RSqrt*** instruction just returns the value from the seed table and this can be refined with one iteration of the Newton Raphson formula, which extends the approximation result from 10 to 20 bits:

```
r = (3.0 - r * r * x) * r * 0.5;
```

The more accurate normalisation is:

```
reg[magSquared] = Dot3 (reg[normal+], reg[normal+]);
reg[r] = RSqrt (reg[magSquared]);
reg[rr] = Mul (reg[r], reg[r]);
reg[r5] = Mul (reg[r], coeff[ConstantHalf]);
reg[rrx] = Mul (reg[rr], reg[magSquared]);
reg[rrx] = Add (coeff[ConstantThree], -reg[rrx]);
reg[invMag] = Mul (reg[rrx], reg[r5]);
reg[normal+] = Mul3 (reg[normal+], reg[invMag]);
```

4.4.3 Texture Co-ordinate Unit (Introduction)

This section describes the assembly language and programming features of the Texture Coordinate Unit. Refer to Texture coordinate generation (s.8.5.1) for details on the exact register setup to load programs into this unit.

The Texture Coordinate Unit is in charge of producing the texture coordinate values so the given texel can be fetched from texture memory and passed on to the rest of the texture pipeline.

For such a task, it receives the per vertex interpolated parameters (**OutTexture CoordA..F**), calculated in the Vertex Shading Unit. For each fragment of the current primitive, the same preloaded program is triggered and executed. The program can last for 2048 core cycles and have up to 128 instructions (a watchdog safety mechanism aborts programs running for more than 2048 cycles).

Although the main purpose of this unit is to generate perspective corrected texture coordinates, its programmable nature allows for more advanced techniques such as:

- Dependent textures Bump Environment Mapping (s.8.5.3)
- Shadow textures
- Procedural textures
- Texture format conversion or automatic mipmap generation at texture download time MIPmap generation (s.7.2)
- Multitap/anisotropic texture filtering

4.4.3.1 Input Data

The unit receives external data from several different sources:

- Feedback data (be it an external stream or data coming from the loopback channel).
- 32 read only Global Registers.
- Interpolated data like the sample position inside the plane (X,Y), plane equations (PStart) and derivatives (dPdx, dPdy).
- Texture scale data (PlaneScale register)

The unit works natively with 32 bit floating point numbers, so any kind of integer data will have to be converted with the **`IntToFloat`** operation, prior to doing any calculation with it (note that the *scale* operand is assumed to be integer and doesn't need to be converted).

Feedback data

The **Feedback** register allows to read data either from an external stream coming from the host (through the **RasteriseRectangle** command with host-supplied data) or from a loopback channel that allows the Texture Coordinate Unit to consume texels from requests originated by the program. Those program requests are done by means of the **`Command`** special field of an instruction with the *enable-feedback* argument set to 1.

You can see the **Feedback** register as a read-only 32bit datum. To read it you must supply as arguments:

- The size of the data to read `8`, `16`, `24` or `32` bits.
- The position of the data inside the register from `0`, `8`, `16` or `24`.
- Whether to signextend the data or not using the keywords `Sign` or `Zero`, respectively.

For example **Feedback(8,24,Zero)** will use as source operand the 8 bits of the last feedback transfer beginning on bit 24 and zero-extending them. `Feedback` operand is normally used in conjunction with **`IntToFloat`** instruction.

The Feedback register is automatically written when data is received. In order the chip not to overwrite feedback data until the program has finished consuming it, or the program not to read the feedback data before it has actually been received, the sequencers **FinishedWithFeedbackData** or **WaitForFeedbackData** can be appended to the instruction to signal the consumption or the need to wait for feedback data.

Global registers

The global register set is composed by 16 pairs of read-only 32bit floating-point numbers. Only registers from the same pair can be accessed in the same operation, so it may be worth planning the arrangement of these global registers with regard to their usage in the program.

The global registers are represented as **Global0[n]** or **Global1[n]** in the assembler syntax, the first form referring to the lower 32 bit word of the pair and the second to the upper 32 bit word of the pair, and being `n` a number from 0 to 15.

Global registers can be addressed relative to the *global register base* (as set by the special instruction field **GlobalBase** or **PlaneGlobalBase)**. In this case, the assembler syntax is to precede the register index with `@`. For example **Global0[@1]** will use the low 32 bit word register at the index given by the *global register base* plus 1.

This relative addressing allows easy subrutinisation and parameterization of code. The interpolated plane equations and derivatives offer a similar facility.

The global registers can also be used to store the *tex-id* (texture ID looked up) and *dest-reg* (Shading Unit registers loaded with the texel data) to use in a Command **`cfifo –register`** field of an instruction.

Interpolated data

The architecture uses a plane equation approach to calculate any interpolated parameter. Some registers in the Texture Coordinate Unit allow accessing that interpolation data.

A parameter to be bilineally interpolated (in this case, texture coordinates) can be seen as a plane equation

$$p = a \cdot x + b \cdot y + c$$

where

- x and y are the current sample point of the plane, accessible from the registers `X` and `Y`.
- a and b are the plane partial derivatives respect to x and y, stored in the registers `dPdx[n]` and `dPdy[n]`
- c is a constant starting value, stored in the register `PStart[n]`.

All those registers are read-only and can be used as source operands for Texture Coordinate Unit operations.

Note that `PStart[n]`, `dPdx[n]` and `dPdy[n]` are arrays of values, `n` ranging from 0 to 7, one for each OutTextureCoord parameter as generated in the Vertex Shading Unit for the current primitive being rasterised.

A **PlaneBase** register allows relative addressing of the plane equation derivatives, much in the same way it's done for global registers. This base register can be loaded with the **PlaneBase** or **PlaneGlobalBase** special fields of the instruction. The derivatives can then be addressed by prepending a @ to the number, i.e dPdx[@1], and the net result will be that of addressing the plane equation given by the number plus the current PlaneBase value.

The usefulness of this relative addressing is evidenced when using subroutines. A good example of this is the *Perspective corrected & mipmapped 2D texture coordinate function* that can be found in section Perspective corrected 2Dtexture Coordinate example (s.4.3.3.6)

Whenever a derivative is used as the source operand for any operation, we can store both derivatives of the plane being addressed in the special registers `dPdxSaved` and `dPdySaved` by means of the *save* instruction field.

Texture scale register

Although normally the calculation of texture coordinates is independent of the size of the texture, for LOD calculation purposes the size of the texture is needed.

This size can be supplied in the 4bit read-only register set **`PlaneScale[n]`**, `n` ranging from 0 to 31 (thus having four scales for each texture ID).

Note that if LOD is to be used, the texture must be power of two, so what this register normally holds is the logarithm of the dimension.

The register set must be loaded beforehand by the programmer and actually nothing stops the programmer from loading other kind of integer data in this register and using it for other purposes than scaling the LOD calculations.

This register set shares the same relative addressing scheme as the plane derivatives interpolation parameters, i.e. setting **`PlaneBase`** or **`PlaneGlobalBase`** values allows to use relative addressing in the form of `PlaneScale[@n]`.

4.4.3.2 Output Data (Texture coordinates, shading parameters)

This unit can either:

- Output the texture coordinates to the texture fetch units (*filtertexture* mode) in order to initiate a texel lookup. The texel conveniently formatted and filtered will eventually be written into the texture input registers of the Shading Unit.
- Pass the calculated data directly to the Shading Unit (*passthrough* mode) useful, for example, for procedurally generated textures.

The output mode is selected by the *command* parameter of the **`Command`** ***cfifo-register*** instruction field present in the program.

The unit communicates to other units through a fifo, represented by the `C[n]` registers in the assembler syntax, *n* going from 0 to 3. For maximum efficiency each of these registers should be only written once. Note that no external action (texture lookup or output passthrough) is triggered until the `Command` instruction is issued.

Filtertexture mode

In this mode the four outputs `C[0..3]` are used as texture coordinates by the Texture Index Unit to perform a texture lookup:

- `C[0]` holds the S coordinate
- `C[1]` holds the T coordinate
- `C[2]` holds the R coordinate (for three dimensional textures)
- `C[3]` holds the LOD value and cubemap face (both are initialised to 0 by default and can be loaded by writing to the **LOD** pseudo register and using the **CubeSort** instruction, respectively).

`C[0..2]` must store *wrapped* values (as opposed to true floating point values). The wrapped value, suitable for texture indexing, is computed by using the `Wrap` operation when writing to the `C[0..2]` register.

To store the cubemap and LOD from their specific registers into `C[3]`, suffices writing the `LOD` register into `C[3]` (the write of the cubemap face is implicit in that instruction).

Only the needed outputs must be written, i.e., if the Texture Index Unit is setup to access non mipmapped 1D textures (as shown In Texture Pipe Method (s.7.1.4.1), there's no need to write to outputs `C[1..3]` in the program.

Passthrough mode

In this mode the four outputs `C[0..3]` are passed directly to the Shading Unit, so care must be taken of converting the floating point numbers to integer numbers with the **`FloatToInt`** operation, as expected by the Shading Unit.

The exact Shading Unit set of four contiguous texture registers that will receive the four outputs is specified by the *dest-reg* parameter of the `Command` instruction.

4.4.3.3 Temporary storage

The Texture Coordinate Unit was designed to be flexible for future uses but also to be efficient for the current usage.

It's because of the latter, that this unit has a lot of specific-purpose registers, in order to efficiently calculate texture coordinates the classic way.

Constants

The only floating point constant values allowed in an operation are 0.0 and 1.0, represented in the assembler syntax by the keywords `Zero` and `One`, respectively.

If other constants are needed by the program, they may be preloaded into global registers.

Working registers

A set of 16 32bit floating point read-write registers allows storing temporary results.

These are represented in the assembler syntax as **`W[n]`** when they appear as destination registers or as **`A[n]`** or **`B[n]`** when they appear as source registers, n ranging from 0 to 15.

The reasoning behind the `A` and `B` naming convention is that the working register set has two input ports, so you must tell the assembler what input port to use when reading a

register from this set. The implication is that as many as two different working registers can be read in the same operation, one appearing as `A` and the other as `B` (although you can read multiple times the same register as long as the operation format supports it).

A register can act both as source and destination of an operation, in that case the source value is the value of the register before the operation took place.

The first nine registers support a form of indirect addressing by preceding the register number with the `%` sign. This feature is normally used for cubemapping and setup by calling the **`CubeSort`** operation (see Bump Environment Mapping (s.8.5.3) for an example). If in a given operation one working register is used with indirect addressing, all the working registers must be used with indirect addressing as well.

Level of detail related registers

Two registers allow fast calculation of analytic level of detail or LOD (refer to Texture Coordinate Generation (s.8.5.1) for the theoretical explanation of LOD calculation).

- **LOD**: This register is written by assigning values to the *LoadMax*, *LoadMag*, *MergeMax* and *MergeMag* special fields of the instruction. This register can be read by using `LOD` as the source of an operation or can be written to `C[3]` using the **C-Fifo** register *instruction* field. This register is read as a 10.8 signed fixed point.
- **Q2**: This register stores a 32bit floating point value that will be used in the calculation implied when writing to the *LoadMax, LoadMag, MergeMax, MergeMag special* fields of the instruction. Note that this register must be loaded (by simply writing to `Q2`) before writing to those fields. This register cannot be read directly.

Flow control registers

- Two 8-bit counters, loaded via `LoadCounter` sequencer and decremented via `DJNZ` sequencer. These registers are addressed by using its index `0` or `1` as parameter to the aforementioned sequencers. These registers cannot be read directly.
- The **flag** register is updated whenever the *update-flag* field is present in the instruction, allowing to overwrite its value or to combined its value with the result of the current operation being positive or zero. The flag can also be used to mask out fragments from the current block with the `DoneAnd` or `KillFragments` sequencers. This register cannot be read directly.

Miscellaneous registers

There are two more specific purpose registers:

- **ScaleReg** is a 5bit read-write sign=, which makes the integer log2 of the operation's result to be written into this register.
- **DivResult** 32bit floating point read-write register that implicitly stores the result of the last division performed. Can be loaded with user-defined values via **`LoadDivResult`** operation.

4.4.3.4 Programs act as transfer functions (data types – bytes, multi-byte arithmetic)

The unit works as a SIMD machine simultaneously on 4x4 fragment regions of the current primitive. A fragment mask disables the output of any of the 16 processors calculating results for fragments in the current 4x4 block but outside the primitive.

- The texture coordinate unit program can also modify that fragment mask by setting the mask out value in the flag register and using ***KillTex*** or ***DoneAnd*** sequencers, thus allowing per pixel and per texture masking effects (the Shading Unit has a similar mechanism as explained in Programs act as transfer functions (s.4.3.4.4).

The SIMD nature of this unit remains hidden to the programmer most of the time but for conditional sequencers (`JumpTrue`, `JumpFalse`), where the need of a unique SIMD flow of execution for all the fragments in the block becomes evident.

4.4.3.5 Instruction Set Summary

This section describes the assembler syntax used by the tools. For a more precise BNF notation of the syntax and the codification of each instruction into bitfields, refer to the Texture Coordinate Unit Assembler & Disassembler Manual.

One program instruction is composed of several fields:
{c-fifo register} {special} {update-flag} {local register} {operation} {sequencer} {save} ";"

Only the operation field is mandatory, all the other fields are optional.

The instruction set is not completely orthogonal, for example some combinations of *operation* and *special* fields are not allowed. Those singularities are explained in the assembler documentation and are intrinsic to the bitfield instruction format, which explanation is beyond the scope of this programming guide.

An explanation of each field follows.

C-Fifo register

This field determines if there will be any write to the c-fifo or if any command will be sent to the texture fetch units or to the Shading Unit.

The possible values of this field are:

Expression	Meaning
C[0] =	The result value of the operation is written into the fifo entry. This is normally the S coordinate for filtertexture commands.
C[1] =	The result value of the operation is written into the fifo entry. This is normally the T coordinate for filtertexture commands.
C[2] =	The result value of the operation is written into the fifo entry. This is normally the R coordinate for filtertexture commands.
C[3] = LOD	The LOD value and the cubemap face are written into the fifo entry.
Command(*command args*)	Triggers the texture lookup or direct passthrough of c-fifo contents to the Shading Unit (see below for full explanation on arguments)

command args are in the form

command, tex-id, dest-reg, load-shading, enable-feedback, prog

where each field can take the following values:

- *command*: This field selects the command that will be executed from:
    - `FilterTexture`: The texel lookup on texture *tex-id* will occur and the RGBA results of the lookup will be copied into the Shading Unit texture registers, as specified by *dest-reg* and *load-shading*.
    - `PassThru2`: The 16 least significative bits of `C[0]` are copied into the two first Shading Unit texture registers of the *dest-reg* set (least significative byte of `C[0]` into the first register, most significative into the second), and the 16 least significative bits of `C[1]` into the two second (least significative byte of `C[1]` into the third register, most significative into the fourth).
    - `PassThru4`: Each Shading Unit texture register of the set given by *dest-reg* is loaded with the lower byte of each c-fifo register.
- *tex-id*: integer from 0 to 7 that selects the texture ID that will be looked up.
- *dest-reg*: integer from 0 to 7 that selects the set of four consecutive texture Shading Unit registers that will receive the texel or passthrough data.
- *load-shading*: `0` or `1` indicating whether to send the effects of the *command* (texel lookup or c-fifo contents) to the Shading Unit texture registers or not.
- *enable-feedback*: `0` or `1` indicating whether to loopback the texture lookup back to the Texture Coordinate Unit. In that case, the texel can be read in the `Feedback` register.
- *prog*: This field specifies the Shading Unit program entry point that will be used (see Start Addresses for Tile Programs (s.5.5.6) for an explanation on multi entry point programs). The possible values are `Default`, `First`, `Middle` or `Last`.

A variant of *command args* that allows reading *tex-id* and *dest-reg* from a global register exists in the form of

command, global, load-shading, enable-feedback, prog

where

- *global*: `Global0`[n]: *tex-id* and *dest-reg* are read from bits 0..2 and 3..5 respectively of the lower word of the given global register.

Finally, if the texture given by *tex-id* is a 3D texture and mipmap linear (linear minification filtering) is used, *dest-reg* least significative bit selects between the high level of detail mipmap and the low level of detail mipmap. See *Mipmap linear 3D textures* for an example of this usage.

Special

Expression	Meaning
GlobalBase(gbase)	Loads the global registers' base register for relative global register addressing. *gbase* is a number from 0 to 31.
LoadMag =	For an operation result `r`, it loads $\left(\log_2(\sqrt{r}) - 2 * \log_2(Q2)\right)$ into the `LOD` register
LoadMax =	For an operation result `r`, it loads $\left(\log_2(r) - 2 * \log_2(Q2)\right)$ into the `LOD` register
MergeMag =	Loads the **LOD** register with the maximum between the current `LOD` register value and $\left(\log_2(\sqrt{r}) - 2 * \log_2(Q2)\right)$.
MergeMax =	Loads the **LOD** register with the maximum between the current `LOD` register value and $\left(\log_2(r) - 2 * \log_2(Q2)\right)$.
PlaneBase(pbase)	Loads the plane base register for relative `PStart`, `Pdx`, `Pdy` and `PlaneScale` addressing. *pbase* is a number from 0 to 31
PlaneGlobalBase(pbase,gbase)	Loads both global register and plane/derivatives base register for relative addressing. *pbase* and *gbase* are numbers from 0 to 31.
Q2 =	Loads the result of the operation into the `Q2` register. Typically used as preliminary step to **LOD** calculation (either using magnitude or maximum techniques).
ScaleReg =	Loads the scale register with the exponent of the result value. Typically used with a `FloatToInt` operation.

Update-flag

By default the flag holds its previous value. The following expressions allow updating the flag depending on the result of the current operation (zero or positive) and the current value of the flag.

Expression	Meaning

Flag = P	Sets the flag to true if the result of this instruction is positive (zero or greater than zero).
Flag = Z	Sets the flag to true if the result of this instruction is zero.
Flag &= P	Ands the current value of the flag with the result of the operation being positive.
Flag &= Z	Ands the current value of the flag with the result of the operation being zero.
Flag |= P	Ors the current value of the flag with the result of the operation being positive.
Flag |= Z	Ors the current value of the flag with the result of the operation being zero.

Local register

The result of an operation can be stored in a local register.

Expression	Meaning
W[uint4] =	Loads the result value of the operation into the given working register.
W[uint4.0] =	Conditionally loads the result value into the given working register if the current flag value is 0.
W[uint4.1] =	Conditionally loads the result value into the given working register if the current flag value is 1.

Operations

The operation field determines the result that will be calculated.

Some of the operations take more than one cycle to be executed, in that case if the next instruction depends on the result of the current instruction and stores its result in a working register, the next instruction will stall until the current has finished calculating the result.

Expression	Cycles	Meaning
AMax(A,B)	1	if (|A| > |B|) r = A else r = B

AnisoRatio(A,B)	1	if (A/B <= 4.0) r = 2.0
		if (A/B < 8.0) r = 4.0
		if (A/B > 8.0) r = 8.0

CubeSort(A,B,C)	1	Sort (A,B,C) and set up face number and indirect addressing. r = 0.0

Expression	Cycles	Meaning
Div(A)	7	divResult = A/B, accurate to 24 bits, r = 0
DivLP(A)	5	divResult = A/B, accurate to 14 bits, r = 0
FloatToInt(A)	1	r = integer(A)
Fract(A)	1	r = Fraction of (A)
IntToFloat(A)	1	r = Float(A)
LoadDivResult(A)	1	divResult = A, r = 0
MAdd(A,B,C,D)	3	r = A*B + C*D
Max(A,B)	1	if (A>B) r = A else r = B
Min(A,B)	1	if (A>B) r = B else r = A
MSub(A,B,C,D)	3	r = A*B – C*D
Select(A,B)	1	if (flag) r = A else r = B
Wrap(A,B)	1	r = Fract(A*B) set flags if: (A*B) > 1.0 (A*B) is odd (A*B) is < 0.0

All the operations have a final optional *scale* parameter, in order to scale them by an integer power of two.

Operands

Each operand *A*, *B*, *C*, *D* and *scale* is restricted to the following values:

- A : `A[n]`, `Zero`, `One`, `dPdx[n]`, `dPdx[n]`, `dPdy[n]`, `PStart[n]`, `DivResult`, `Feedback`(*size*, *position*, *signextend*)
- B: `A[n]`, `B[n]`, `Zero`, `One`, `Global0[n]`, `Global1[n]`, `X`
- C: `A[n]`, `B[n]`, `dPdx[n]`, `dPdy[n]`, `dPdxSaved`, `dPdySaved`
- D: `B[n]`, `Zero`, `One`, `Global0[n]`, `Global1[n]`, `Y`, `LOD`
- Scale: `PlaneScale[n]`, *sint5*, `ScaleReg`, `-ScaleReg`

sint5 stands for 5bit integer number with sign.

Sequence control

The sequencer field determines which instruction will be executed after this one.

Expression	Meaning

Expression	Meaning
Call(addr)	The current instruction address + 1 is written to the stack. The next instruction will be the one at *addr*. The address stack can be popped with the Return sequencer.
DJNZ(loop-id, addr)	Decrements the given counter. The next instruction is the given address if the counter is not zero or the current address + 1 otherwise.
Done	Signals the end of the program. The unit will reset its internal state and prepare to execute the first instruction of the program on the next batch of data.
DoneAnd	Signals the end of the program and masks the tilemask with the current flag value.
FinishedWithFeedbackData	Indicates that the program no longer uses the current feedback data and that the feedback registers can be loaded with more data.
Increment	Go to the current instruction + 1 (default value)
Jump(addr)	Sets *addr* to be the next instruction.
JumpFalse(cc, addr)	If the And or the Or (as specified by *cc*) of all the fragment flags is false, sets *addr* to be the next instruction.
JumpTrue(cc, addr)	If the And or the Or (as specified by *cc*) of all the fragment flags is true, sets *addr* to be the next instruction.
KillFragments	Same as `DoneAnd`, but allowing the program to continue.
LoadCounter(loop-id, uint8)	Loads one of the two 8-bit counters with the *uint8* value.
Return(addr)	Pops the address in the stack and makes that the next instruction address. *addr* can be pushed onto the stack with the Call sequencer.
WaitForFeedbackData	Stall until the feedback registers have received the feedback data.

The instruction counter wraps around, so any increment beyond the maximum number of instructions will still be valid.

The addresses are labels or 8bit numbers. They can be absolute or relative (if preceded by `@`), in which case the destination address is the current instruction counter plus the specified address.

The *cc* operand can be *And* or *Or*.

The address stack is 4 entries deep and there is no underflow or overflow control.

Save

By appending `SavedP` to the instruction, the derivatives currently being accessed as operands in the current operation are saved away in the `dPdxSaved` and `dPdySaved` registers.

4.4.3.6 Examples

This section contains some commented examples ranging from the simplest 2D non-perspective corrected texture coordinate generation program, to the most advanced techniques.

The supplied assembler allows defining named constants so, throughout the examples and for clarity's sake, we refer to several constants defined as follows:

```
// Constants to access plane equation, derivatives,
// C-Fifo and working registers
Define(Q,3)
Define(S,0)
Define(T,1)
Define(R,2)

// Defines for "Command" instruction
Define(DoLoadShade, 1)
Define(NoLoadShade, 0)
Define(NoFeedback,0)
Define(DoFeedback, 1)
Define(LoResTexture, 0)
Define(HiResTexture, 1)

// Handy defines for accessing external plane equation and
// derivatives in multi stage programs
Define(S0,0)
Define(T0,1)
Define(R0,2)
Define(Q0,3)
Define(S1,4)
Define(T1,5)
Define(R1,6)
Define(Q1,7)
```

```
// Temporals for MIPMAPLINEAR 3D textures
// (we can overwrite Q,S,T,R, dsdx, dsdy, dtdx, dtdy)
Define(HiRed, 0)        // High level of detail texture
Define(HiGreen,1)
Define(HiBlue,2)
Define(HiAlpha,3)
Define(FifoRed, 0)
Define(FifoGreen, 1)
Define(FifoBlue, 2)
Define(FifoAlpha, 3)
Define(LoRed,4)                 // Low level of detail texture
Define(LoGreen,5)
Define(LoBlue,6)
Define(LoAlpha,7)
Define(NegativeLOD, 8)   // Temporaries to extract the lod
Define(PositiveLOD, 9)
Define(Temp, 10)
```

Non-perspective corrected 2D texture coordinate program

One of the simplest programs we can write is the one that produces non-perspective corrected 2D texture coordinates:

```
// Program without perspective correction
// If used, must be given non-perspective corrected coords!
Program(TCU_VANILLA,0x00)
    // W[S] will hold S texture coordinate
    W[S]= MAdd(dPdx[S],X,dPdy[S],Y);
    // W[T] will hold T texture coordinate
    W[T]= MAdd(dPdx[T],X,dPdy[T],Y);

    // Because the next instruction accesses W[S], there
// will be a one cycle stall here
    W[S]= MAdd(PStart[S],One,B[S],One);
    W[T]= MAdd(PStart[T],One,B[T],One);

    // Convert the coordinates to wrapped mode
// and write to fifo
// There's no stall here because we write to the fifo
    C[S]= Wrap(A[S],One);
    C[T]= Wrap(A[T],One);
    // FloatToInt(One) acts as a NOP
    Command(FilterTexture,0,0,DoLoadShade,NoFeedback,Default) FloatToInt(One);
```

The net result of this program is that

- (PStart[S] + dPdx[S]*X + dPdy[S]*Y) is used as S texture coordinate
- (PStart[T] + dPdx[T]*X + dPdy[T]*Y) is used as T texture coordinate

- A lookup of texture ID 0 is issued using those texture coordinates, and its results are sent to the first register set of the Shading Unit (registers `T[0..3]`).

Filtering of the texel is performed automatically by the Texture Index Unit, as specified in the TextureIndexMode register and described in @@@

The `Command` instruction is a bit peculiar: because the *operation* field is mandatory and there's no NOP operation in the instruction set, we use a dummy `FloatToInt(One)` operation that acts as a NOP. Ideally, we would pair the `Command` instruction with a useful operation, but in this case we cannot do it, as it has to be the last instruction of this block.

Note how texture coordinate calculations are interleaved to minimize the number of stalls.

Perspective corrected 2D texture coordinate program

Next in the step of complexity is the perspective corrected version: Because 3D to 2D space conversion implies a division by the distance from the primitive to the viewer, lineally interpolating the texture coordinates is wrong and produces well-known texture *swimming* effects (unless the polygon is completely facing the viewer).

What we need is a *hyperbolic* interpolation. This is achieved by dividing the texture coordinates by the Q texture coordinate, in what is normally called perspective correction.

```
Program(TCU_PERSP_CORRECT_2D,0x00)
W[Q] = MAdd(dPdx[Q], X, dPdy[Q], Y);
        W[S] = MAdd(dPdx[S], X, dPdy[S], Y);
        W[T] = MAdd(dPdx[T], X, dPdy[T], Y);

        W[Q] = MAdd(PStart[Q], One, B[Q], One);
        W[S] = MAdd(PStart[S], One, B[S], One);
        W[T] = MAdd(PStart[T], One, B[T], One);

        // Begin to do the async division
        DivLP(One, B[Q]);

        // Perspective correct
        // Note there's no stall when accessing DivResult
// because the result is written to c-fifo
        C[S] = Wrap(DivResult, B[S]);
        C[T] = Wrap(DivResult, B[T]);
        // Send to the texel fetch units
Command(FilterTexture,0,0,DoLoadShade,NoFeedback,Default) FloatToInt(One);
```

There are three changes from the previous program:

- We now calculate the value of the Q coordinate, the same way we do for S and T. Note that interleaving the calculation of Q coordinate with S and T, has removed the stall (we now use that cycle to calculate the Q value).
- We now calculate the reciprocal of Q and store into the **DivResult** register.
- When using the `Wrap` operation, we now factor **DivResult** into the wrapping.

Perspective corrected & mipmapped 2D texture coordinate program

See Texture coord generation (s.8.5.1) for the theory behind the LOD calculation:

```
Program(TCU_PERSP_CORRECT_2D_MIPMAP_MAX, 0x00)
    W[Q] = MAdd(dPdx[Q], X, dPdy[Q], Y) SavedP;
    W[S] = MAdd(dPdx[S], X, dPdy[S], Y);
    W[T] = MAdd(dPdx[T], X, dPdy[T], Y);

    Q2 = W[Q] = MAdd(PStart[Q], One, B[Q], One);
    W[S] = MAdd(PStart[S], One, B[S], One);
    W[T] = MAdd(PStart[T], One, B[T], One);

    // Begin to do the async division
    DivLP(One, B[Q]);

    // Calculate derivatives for LOD.
    LoadMax = MSub(dPdx[S], A[Q], dPdxSaved, B[S], PlaneScale[S]);
    MergeMax = MSub(dPdy[T], A[Q], dPdySaved, B[T], PlaneScale[T]);
    MergeMax = MSub(dPdx[S], A[Q], dPdxSaved, B[S], PlaneScale[S]);
    MergeMax = MSub(dPdy[T], A[Q], dPdySaved, B[T], PlaneScale[T]);

    // Perspective correct
    C[S] = Wrap(DivResult, B[S]);
    C[T] = Wrap(DivResult, B[T]);
    // Load the LOD value into the fifo
    C[3] = LOD FloatToInt(One);
Command(FilterTexture,0,0,DoLoadShade,NoFeedback,Default) FloatToInt(One) Done;
```

In this case, the max technique is used for LOD calculation. The PlaneScale registers scale the LOD calculation by the width or height of the texture.

Two stage perspective corrected & mipmapped 2D texture coordinate program

For using two texture stages, we just have to replicate the program, taking care of accessing the right external interpolated sources for each stage (`PStart`, `dPdx` and `dPdy`) and the right texture scale register (`PlaneScale`).

```
Program(TCU_PERSP_CORRECT_2D_MIPMAP_STAGES_01, 0x00)
    // First texture stage, note we access Q0,S0,T0
    W[Q] = MAdd(dPdx[Q0], X, dPdy[Q0], Y) SavedP;
    W[S] = MAdd(dPdx[S0], X, dPdy[S0], Y);
    W[T] = MAdd(dPdx[T0], X, dPdy[T0], Y);

    Q2 = W[Q] = MAdd(PStart[Q0], One, B[Q], One);
    W[S] = MAdd(PStart[S0], One, B[S], One);
    W[T] = MAdd(PStart[T0], One, B[T], One);
```

```
// Begin to do the async division
DivLP(One, B[Q]);

// Calculate derivatives for LOD.
LoadMax = MSub(dPdx[S0], A[Q0], dPdxSaved, B[S0], PlaneScale[S0]);
MergeMax = MSub(dPdy[T0], A[Q0], dPdySaved, B[T0], PlaneScale[T0]);
MergeMax = MSub(dPdx[S0], A[Q0], dPdxSaved, B[S0], PlaneScale[S0]);
MergeMax = MSub(dPdy[T0], A[Q0], dPdySaved, B[T0], PlaneScale[T0]);

// Perspective correct
C[S] = Wrap(DivResult, B[S]);
C[T] = Wrap(DivResult, B[T]);
C[3] = LOD FloatToInt(One);
// Send filtertexture command for texture id 0
Command(FilterTexture,0,0,DoLoadShade,NoFeedback,Default) FloatToInt(One);

// Second texture stage, note we access Q1,S1,T1
W[Q] = MAdd(dPdx[Q1], X, dPdy[Q1], Y) SavedP;
W[S] = MAdd(dPdx[S1], X, dPdy[S1], Y);
W[T] = MAdd(dPdx[T1], X, dPdy[T1], Y);

Q2 = W[Q] = MAdd(PStart[Q1], One, B[Q], One);
W[S] = MAdd(PStart[S1], One, B[S], One);
W[T] = MAdd(PStart[T1], One, B[T], One);

// Begin to do the async division
DivLP(One, B[Q]);

// Calculate derivatives for LOD.
LoadMax = MSub(dPdx[S1], A[Q], dPdxSaved, B[S], PlaneScale[S1]);
MergeMax = MSub(dPdy[T0], A[Q], dPdySaved, B[T], PlaneScale[T1]);
MergeMax = MSub(dPdx[S0], A[Q], dPdxSaved, B[S], PlaneScale[S1]);
MergeMax = MSub(dPdy[T0], A[Q], dPdySaved, B[T], PlaneScale[T1]);

// Perspective correct
C[S] = Wrap(DivResult, B[S]);
C[T] = Wrap(DivResult, B[T]);
C[3] = LOD FloatToInt(One);
// Send filtertexture command for texture id 1
Command(FilterTexture,1,1,DoLoadShade,NoFeedback,Default) FloatToInt(One) Done;
```

Note how the second `Command` instruction gets the texel data from the second texture and targets it to the second Shading Unit register set.

With this code, Shading Unit registers T[0..3] (register set 0) will receive the RGBA values of the lookup of texture with ID 0, while Shading Unit registers T[4..7] (register set 1) will receive the RGBA values of the lookup of texture with ID 1.

Perspective corrected & mipmapped 2D texture coordinate function

In the previous example, the program length grows lineally with each texture stage we add. We can achieve the same result using functions and function calls:

```
Program(TCU_PERSP_CORRECT_2D_MIPMAP_FUNCTION, 0x00)
    W[Q] = MAdd(dPdx[@Q], X, dPdy[@Q], Y) SavedP;
    W[S] = MAdd(dPdx[@S], X, dPdy[@S], Y);
    W[T] = MAdd(dPdx[@T], X, dPdy[@T], Y);


    Q2 = W[Q] = MAdd(PStart[@Q], One, B[Q], One);
    W[S] = MAdd(PStart[@S], One, B[S], One);
    W[T] = MAdd(PStart[@T], One, B[T], One);

    // Begin to do the async division
    Div(One, B[Q]);

    // Calculate derivatives for LOD and store them in temporaries
    W[dsdx] = MSub(dPdx[@S], A[Q], dPdxSaved, B[S], PlaneScale[@S]);
    W[dtdx] = MSub(dPdx[@T], A[Q], dPdxSaved, B[T], PlaneScale[@T]);
    W[dsdy] = MSub(dPdy[@S], A[Q], dPdySaved, B[S], PlaneScale[@S]);
    W[dtdy] = MSub(dPdy[@T], A[Q], dPdySaved, B[T], PlaneScale[@T]);

    // Calculate the LOD approximation using the magnitude
    LoadMag = MAdd(A[dsdx], A[dsdx], B[dtdx], B[dtdx]);
    MergeMag = MAdd(A[dsdy], A[dsdy], B[dtdy], B[dtdy]);

    // Perspective correct
    C[S] = Wrap(DivResult, B[S]);
    C[T] = Wrap(DivResult, B[T]);
    // Write the level of detail value. FloatToInt(One) acts as a NOP
    C[3] = LOD FloatToInt(One) Return;
```

Because the function doesn't know which interpolated parameters or texture scale register needs to access, it uses those registers with relative addressing. At function call time, the right base value will be stored in `PlaneBase` register.

The `Command` instruction will also have to be issued after calling the function.

The previous function is called for two texture stages with the following code:

```
Program(TCU_FUNCTION_CALL,0x00)

        // Setup plane base and call function for first stage
    // "Min(A[0], B[0])" acts as a dummy (NOP)
    // instruction that doesn't use SourceA bit field
        PlaneBase(0) Call(TCU_PERSP_CORRECT_2D_MIPMAP_FUNCTION) Min(A[0], B[0]);
```

```
// Send FilterTexture command for first texture stage
Command(FilterTexture,0,0,DoLoadShade,NoFeedback,Default) FloatToInt(One);

// Setup plane base and call function for second stage
PlaneBase(4) Call(TCU_PERSP_CORRECT_2D_MIPMAP_FUNCTION) Min(A[0], B[0]);
// Send FilterTexture command for second texture stage
Command(FilterTexture,1,1,DoLoadShade,NoFeedback,Default) FloatToInt(One) Done;
```

On each function call, we setup the `PlaneBase` register so the function accesses the right set of registers for the given texture.

Note that because of the function call setup, this program is two execution cycles longer, although much shorter in program space: only two more instructions are needed for each extra texture stage.

The `Command` instruction could be moved inside the function if the *global* field version of the instruction was used: You could preload the lower word of global registers 0 and 4 with values 0x0 and 0x9 respectively, and append the line.

```
Command(FilterTexture,Global0[@0],DoLoadShade,NoFeedback,Default) FloatToInt(One)
        Return;
```

to the function (obviously removing the `return` sequencer from the former last line of the function). The function call would be a simple

```
Program(TCU_FUNCTION_CALL,0x00)
                        // Setup plane base and call function for first stage
// "Min(A[0], B[0])" acts as a dummy (NOP)
// instruction that doesn't use SourceA bit field
        PlaneBase(0) Call(TCU_PERSP_CORRECT_2D_MIPMAP_FUNCTION) Min(A[0],
        B[0]);
        // Setup plane base and call function for second stage
        PlaneBase(4) Call(TCU_PERSP_CORRECT_2D_MIPMAP_FUNCTION) Min(A[0],
        B[0]);
```

Using this approach, each new texture stage requires only one extra line.

Mipmap linear 3D textures

The texture fetch mechanism can filter up to 8 texels at the same time. This is normally achieved using

- Trilinear filtering on 2D textures: using four texels from one mipmap level and four texels from the next mipmap level.
- Trilinear filtering on 3D textures: using four texels from one 3D slice and four texels from the next 3D slice. Note that in this case we don’t use the mipmap level for the

third linear interpolation, but we interpolate between bilinear filters of adjacent depth slices.

Due to this limitation, to do mipmap linear filtering on 3D textures it's necessary to use a special texture coordinate unit program that:

- Sets the fifo entries to the necessary S, T, R and LOD values.
- Fetches the texel through for the high level of detail 3D texture and receives it using the feedback mechanism.
- Fetches the texel through feedback for the low level of detail 3D texture, using the .
- Calculates the interpolation between one and the other with regard to the computed LOD value.
- Uses passthrough to send the interpolated RGBA values to the Shading Unit.

This is by far the most complicated program of this section and fully demonstrates the flexibility of the Texture Coordinate Unit. Other advanced programs can be found in chapter 10.

```
Program(TCU_PERSP_CORRECT_3D_MIPMAP_FUNCTION, 0x00)
        W[Q] = MAdd(dPdx[@Q], X, dPdy[@Q], Y) SavedP;
        W[S] = MAdd(dPdx[@S], X, dPdy[@S], Y);
        W[T] = MAdd(dPdx[@T], X, dPdy[@T], Y);
        W[R] = MAdd(dPdx[@R], X, dPdy[@R], Y);

        Q2 = W[Q] = MAdd(PStart[@Q], One, B[Q], One);
        W[S] = MAdd(PStart[@S], One, B[S], One);
        W[T] = MAdd(PStart[@T], One, B[T], One);
        W[R] = MAdd(PStart[@R], One, B[R], One);

        // Begin to do the async division
        Div(One, B[Q]);

        // Calculate derivatives for LOD
        W[dsdx] = MSub(dPdx[@S], A[Q], dPdxSaved, B[S], PlaneScale[@S]);
        W[dtdx] = MSub(dPdx[@T], A[Q], dPdxSaved, B[T], PlaneScale[@T]);
        W[drdx] = MSub(dPdx[@R], A[Q], dPdxSaved, B[R], PlaneScale[@R]);
        W[dsdy] = MSub(dPdy[@S], A[Q], dPdySaved, B[S], PlaneScale[@S]);
        W[dtdy] = MSub(dPdy[@T], A[Q], dPdySaved, B[T], PlaneScale[@T]);
        W[drdy] = MSub(dPdy[@R], A[Q], dPdySaved, B[R], PlaneScale[@R]);

        W[temp] = MAdd(A[dsdx], A[dsdx], B[dtdx], B[dtdx]);
        LoadMag = MAdd(A[temp],One, B[drdx], B[drdx]);
        W[temp] = MAdd(A[dsdy], A[dsdy], B[dtdy], B[dtdy]);
        MergeMag = MAdd(A[temp], One, B[drdy], B[drdy]);

        // Perspective correct
        C[S] = Wrap(DivResult, B[S]);
```

```
        C[T] = Wrap(DivResult, B[T]);
        C[R] = Wrap(DivResult, B[R]);
        C[Q] = LOD FloatToInt(One) Return;

Program(TCU_3D_MIPMAP_LINEAR_FUNCTION_CALL,0x00)
        // Call the function to setup the texture coordinates
        PlaneGlobalBase(0,0) Call(TCU_PERSP_CORRECT_3D_MIPMAP_FUNCTION)
        Min(A[0], B[0]);

        // Get the texels from hi res texture
        Command(FilterTexture, 0, HiResTexture, NoLoadShade, DoFeedback, Default)
        FloatToInt(One);

        // Store LOD in parallel with first feedback
        Flag = P W[Temp] = MAdd(Zero,Zero, One, LOD);
        W[NegativeLOD] = MSub(A[Temp],One, One, One);
        W[PositiveLOD] = MAdd(A[Temp],One,One,One);
        W[Temp] = Select(A[PositiveLOD],B[NegativeLOD],14);
        W[Temp] = Fract(A[Temp]) WaitForFeedbackData;

        W[HiRed] = IntToFloat(Feedback(8,0,Zero));
        W[HiGreen] = IntToFloat(Feedback(8,8,Zero));
        W[HiBlue] = IntToFloat(Feedback(8,16,Zero));
        W[HiAlpha] = IntToFloat(Feedback(8,24,Zero)) FinishedWithFeedbackData;

        // Get the texels from lo res texture, in parallel
        // with the previousCommand(FilterTexture, 0, LoResTexture,
        //NoLoadShade, DoFeedback, Default) FloatToInt(One);

        // Do the interpolation for Hi colors, and work back to back with second feedback
        W[HiRed] = MSub(A[HiRed], One, A[HiRed],B[Temp]);
        W[HiGreen] = MSub(A[HiGreen], One, A[HiGreen],B[Temp]);
        W[HiBlue] = MSub(A[HiBlue], One, A[HiBlue], B[Temp]);
        W[HiAlpha] = MSub(A[HiAlpha], One, A[HiAlpha], B[Temp])
        WaitForFeedbackData;

        W[LoRed] = IntToFloat(Feedback(8,0,Zero));
        W[LoGreen] = IntToFloat(Feedback(8,8,Zero));
        W[LoBlue] = IntToFloat(Feedback(8,16,Zero));
        W[LoAlpha] = IntToFloat(Feedback(8,24,Zero)) FinishedWithFeedbackData;

        // Finish the interpolation
        W[LoRed] = MAdd(A[LoRed],B[Temp],Zero, Zero);
        W[LoGreen] = MAdd(A[LoGreen],B[Temp],Zero, Zero);
        W[LoBlue] = MAdd(A[LoBlue],B[Temp],Zero, Zero);
        W[LoAlpha] = MAdd(A[LoAlpha],B[Temp],Zero, Zero);

        W[HiRed] = MAdd(A[LoRed], One, B[HiRed], One);
```

```
W[HiGreen] = MAdd(A[LoGreen], One, B[HiGreen], One);
W[HiBlue] = MAdd(A[LoBlue], One, B[HiBlue], One);
W[HiAlpha] = MAdd(A[LoAlpha], One, B[HiAlpha], One);

C[FifoRed] = FloatToInt(A[HiRed]);
C[FifoGreen] = FloatToInt(A[HiGreen]);
C[FifoBlue] = FloatToInt(A[HiBlue]);
C[FifoAlpha] = FloatToInt(A[HiAlpha]);

// Pass through to shading unit
Command(PassThru4, 0,0, DoLoadShade, NoFeedback, Default) FloatToInt(One)
Done;
```

Note that this is not the only way of achieving mipmap linear 3D textures. You could:

- Send the hi level of detail and the low level of detail texel values to the Shading Unit (either using multiple programs and the same texture register set or two texture register sets).
- Send the interpolation value (LOD) to the Shading Unit using passthrough mode.
- Interpolate the texels in the Shading Unit.

Tricks of the trade

Programming the Texture Coordinate Unit is always a tradeoff between program space, program execution cycle count and image quality.

Some tips worth taking into account when coding programs follow:

- Use **`DivLP`** (low precision divide) and LOD calculation approximations (magnitude method) if you don't require high texture quality. The effect of these approximations is normally a too high frequency mipmap on very anisotropic primitives, or texture *swimming* on high valued texture coordinates (textures with a high repeat factor).
- Pair instructions whenever possible by interleaving different coordinate/texture stage calculations to avoid stalls.
- Don't calculate the LOD value if not needed: some textures work just well without using mipmaps (for example lightmaps) and when combining several texture stages, not using mipmaps allows better pairing between the calculations from different stages. Recall, though, that even if imipmaps are not used, LOD calculation is necessary if the minification filter differs from the magnification filter.
- Sharing 1/Q value among texture stages, don't perspective correct if not needed (particles/billboards).
- Z-Buffer values are comparable although not compatible with texture coordinate floating point numbers. This means that you can map the z-buffer
- Accessing LOD for bias.

4.4.4 Introduction to Shading

This section describes the basic operation of the Shading Unit, a programmable unit designed for per-pixel shading calculations.

By shading we mean the calculation of a colour value for each pixel, or fragment, making up a primitive. By per-pixel we mean those operations that are performed independently

for each pixel, as opposed to only once for, say, each vertex or primitive. The functions performed by the Shading Unit are roughly those specified by the texture stage pipeline, in OpenGL 1.2 and DirectX7, and the arithmetic instructions of a pixel shader program, in DirectX8.

In addition, the Shading Unit is typically used to implement fog and alpha test.

Typically, some aspects of shading are performed independently for each pixel, while other, more expensive, calculations are performed per-vertex or per-primitive and approximated at the pixel level by means of linear interpolation. The per-vertex aspects of shading calculations are performed by the Vertex Shading Unit (See *Section 4.3.2*) and are not discussed here.

Although per-pixel shading is conceptually performed for each fragment independently, the Shading Unit has sixteen fragment processors in a SIMD array and so processes sixteen fragments in parallel. Due to its SIMD architecture, the same operations are performed for every fragment.

Note: The values calculated typically differ from one fragment to the next. This requires some degree of separation of the fragments, and resources in the Shading Unit can be usefully characterised as being either shared globally or else replicated locally for each fragment.

Although the Shading Unit is a multi-processor unit, it is sometimes useful to describe its operation in terms of only a single fragment; for example we might say that there are thirty-two local registers in the Shading Unit, when what we mean is that there are thirty-two local registers for each fragment.

4.4.4.1 Input Data (plane equations, textures)

Input data to per-pixel shading operations consists of interpolated data arising from per-vertex calculations, as well as sampled and filtered texture data arising from texture lookups. In addition calculations may make use of per-primitive constants loaded directly by the driver, which are described in Section 4.3.4.3.

The results of per-vertex shading calculations are linearly interpolated by means of plane equation evaluators to produce interpolated values for each fragment. Specifically, the Colour[A...H] outputs of the Vertex Shading Unit are interpolated and made available for use in per-pixel shading calculations by means of the Shading Unit plane equation registers.

There are thirty-two 8-bit plane equation registers per fragment. However they serve functionally as eight four-component 32-bit registers, since they are loaded with the interpolated four-component values derived from the eight Vertex Shading Unit Colour outputs.

The interpolated values contained in the plane equation registers are typically thought of as colours, but their use is arbitrary and dictated entirely by their function within the supplied Shading Unit program. Other possible uses include interpolated per-pixel normals and matrix coefficients for per-pixel lighting calculations.

As well as interpolated values arising from per-vertex calculations, per-pixel shading calculations may make use of filtered texture data derived from per-fragment texture lookups. The calculation of texture coordinates for texture lookups is performed by the Texture Coordinate Unit (See Section 4.3.3), and is not described here. The filtered

texture data arising from these lookups is made available for use in shading calculations by means of the Shading Unit texture registers.

There are thirty-two 8-bit texture registers per fragment, but functionally they serve as eight four-component 32-bit registers. In this view texture register n is loaded with the filtered RGBA texture data for texture n.

4.4.4.2 Output Data (fragment colors)

Since the function of the Shading Unit is to perform per-pixel shading calculations, its output is simply an RGBA colour value per-fragment that is forwarded for use as the colour for that fragment. The RGBA output value consists of four 8-bit components.

The output colours are written (under program control) to the C FIFO, which forms the interface to the next unit. Because the C FIFO is a queue rather than a register set, each component of the output colour must be written to the C FIFO exactly once. However the components may be written in any order.

4.4.4.3 Meory (instructions, global data, temporaries)

Storage for intermediate results is provided in the form of the Shading Unit local registers. There are thirty-two local registers per fragment, and each local register is signed 12-bit precision with a fixed-point s3.8 format. This gives the ability to represent signed values within the range [-0x8.00, 0x7.FF] during program calculations.

Storage is also provided for thirty-two global registers, which are shared across all fragments rather than being instantiated for each. Each global register is of 8-bit unsigned precision. Global registers are loaded by the driver by means of the **ShadeGlobal**[8] registers. The **ShadeGlobal** registers load the global registers in four-register-aligned four-register groups, such that the global registers are notionally grouped into eight 32-bit four-component registers, from the point of the view of the driver.

Finally, storage is provided for the Shading Unit program, which is the sequence of Shading Unit instructions that the unit will execute. The single program buffer is shared by all fragment processors, in keeping with the SIMD nature of the unit. This buffer provides storage for 128 instructions. The driver loads the instructions of the program using the **ShadeProgramData** command. The instructions are loaded at the current load address, which is auto-incremented after each load. The **ShadeProgramAddr** command sets the current load address.

4.4.4.4 Programs act as transfer functions (data types – bytes, multi-byte arithmetic)

The program executed by the Shading Unit computes a single output RGBA colour value from a range of available inputs. The program alone defines the function that is computed.

The instruction opcodes that are provided are targetted at the following calculations.

- Combining colour values: addition, modulation, subtraction, etc.
- Interpolation between colours.
- Comparing values.
- Selecting between two computed values depending on comparison results.
- Performing multi-word arithmetic using carry.

Dot products between pairs of vectors may be computed by means of sequences of multiplies and adds.

Internal results are stored in the local registers and hence are represented in s3.8 signed fixed-point format. This permits signed calculations on quantities such as vectors.

In addition the program is able to set or clear fragment flags, marking fragments as valid or invalid. This can be used to implement alpha test and texkill.

4.4.4.5 Argument Formats

Note that while the internal representation of the Shading Unit local registers is s3.8 signed fixed-point, the texture and plane equation registers are only 8-bit unsigned. This is because the values stored in the latter register types are derived from sources such as interpolated vertex data and textures, which have historically represented 8-bit colour components.

For this reason some means is required of representing signed values with ranges other than [0, 255] in the 8-bit registers, together with a means of ensuring that those values are correctly interpreted on being read from such registers.

It is now common to represent signed floating point values in the unsigned 8-bit format by means of scaling and biasing. For example values in the range [-1, 1] can be represented as follows:

- Scale by 0.5 to produce [-0.5, 0.5] range.
- Bias by +0.5 to produce [0, 1] range.
- Scale by 255 to produce [0, 255] unsigned 8-bit range.

Signed values represented in this way may safely be interpolated and texture filtered. Likewise it is common to represent numbers in the floating point range [0, 1] by scaling them by 255 to produce the 8-bit unsigned range [0, 255].

The Shading Unit instruction format includes intrinsic argument formats intended for use in interpreting scaled and biased values sourced from texture and plane equation registers. These formats are mapping modes that dictate how unsigned 8-bit values should be promoted to the internal s3.8 format. The following are the most useful argument formats:

- *ZeroExtend.* In this format 8-bit unsigned values are simply zero-extended with a zero (positive) sign bit and three zero integer bits, converting the unsigned 8-bit integer range [0,255] to the fixed point range [0x0.00, 0x0.FF].
- *MapToOne.* This format is provided for mapping the unsigned 8-bit integer range [0,255] to the fixed point range [0x0.00, 0x1.00], ensuring that values of 255 get interpeted as 1.0.
- *Bias2.* This format deducts a bias of 0.5 and scales by two. The initial 8-bit value is assumed to be a biased-and-scaled representation of a number in the range [-1, 1].

4.4.4.6 Multi-word Arithmetic

If the internal s3.8 numeric representation is not of sufficient range, it is possible to perform multi-word arithmetic, at some considerable performance cost. Various constructs are provided in the Shading Unit for this purpose:

- All addition and subtraction operations generate a carry.
- Versions of the addition and subtraction opcodes that take into account a previously generated carry.
- Upper-word and lower-word versions of the multiplication opcode, for calculating the upper and lower word of a multi-word multiplication product.

- Subroutining with relative addressing, which may be useful for the coding of general-purpose multi-word arithmetic routines.

4.4.4.7 Saturation

The Shading Unit comes equipped with versions of the arithmetic opcodes that saturate to the internal range of [-0x8.00, 0x7.FF]. However it is commonly desirable to saturate the results of internal computations to some range other than the internal range. For example DirectX7 requires that the results of each texture stage should be clamped to [0, 1] range between stages. Likewise DirectX8 requires that the results of pixel shader instructions should be clamped to the numerical range exported by the hardware, which is typically [-1, 1]. Finally DirectX8 specifies a saturate instruction modifier which saturates the result of an instruction to [0, 1].

The Shading Unit has a ***Saturate*** opcode which saturates its input to either [0, 1] or [-1, 1], depending on the value of the Div2 instruction field. Thus saturation can be performed whenever required, at the cost of a single instruction. Ideally it is possible to track the potential numerical range of results at program generation time (given the limited range of the inputs and the potential range of earlier results) and so avoid generating explicit saturation instructions except when absolutely necessary.

Saturation to other imaginable ranges is not immediately possible but could be accomplished at some expense by scaling, saturating, and de-scaling.

4.4.4.8 Subroutines and Relative Addressing

The limited size of the program buffer is the greatest limitation on program usefulness. In the worst case, Shading Unit programs generated naively from DX8 pixel shader programs may be too long to fit in the available space. As a solution the Shading Unit instruction format provides constructs for subroutining and two distinct methods of relative addressing.

A useful technique is to define standard subroutines for the various operations defined by the API and to call these for each set of operands as appropriate. Another is to call the same program code for each component of a vector or colour in turn, possibly using a second level of subroutines.

Subroutining requires generality of operands to be useful, and relative addressing is the chief means of achieving this.

The Shading Unit has four addressing modes, which are the cartesian product of two independent mechanisms:

- Absolute
- AbsoluteComponent
- ArgRelative
- ArgRelativeComponent

The first mechanism is component relative addressing, in which the least significant two bits of the register address provided in the instruction field are replaced with a 2-bit component offset explicitly set on the calling instruction (which implies it is only useful within subroutines). The intended use of this mechanism is the implementation of subroutines which operate on single colour or vector components and can be called for each relevant component in turn.

The second mechanism is the more traditional arg relative addressing, in which the address provided in the instruction field is treated as an offset to be added to one of four

possible base addresses. The four base addresses are labelled A, B, C and D, and any one may be used orthogonally on any register reference. The A, B and C base address registers are loaded automatically on any call instruction with the effective addresses of the two source arguments and destination, respectively. The D base address register may be loaded explicitly using an Arg opcode. The previous set of base addresses is pushed onto an internal stack on a subroutine call, and popped off the stack on the return, allowing nesting of subroutine calls with arg relative addressing.

The intended use of the second relative addressing mechanism is the implementation of a library of subroutines that perform standard operations. The call instruction (which may be the inlined first instruction of the routine) is responsible for loading the base address registers with the addresses of the parameters to be read and written in the routine.

4.4.4.9 Opcodes

The following instruction opcodes are supported:

Opcode	Description	Notes
Add	Q = A + B	Basic add.
AddC	Q = A + B + carry	Add with carry for multi-word arithmetic.
AddS	Q = A + B Q = min(Q, 0x7.ff) Q = max(Q, -0x8.00)	Saturating add.
AddSC	Q = A + B + carry Q = min(Q, 0x7.ff) Q = max(Q, -0x8.00)	Saturating add with carry.
Sub	Q = A – B	Basic subtract.
SubC	Q = A – B – carry	Subtract with carry for multi-word arithmetic.
SubS	Q = A – B Q = min(Q, 0x7.ff) Q = max(Q, -0x8.00)	Saturating subtract.
SubSC	Q = A – B – carry Q = min(Q, 0x7.ff) Q = max(Q, -0x8.00)	Saturating subtract with carry.
MultU	Q = (a * b) >> 12	Upper word of multi-word multiply result.
MultL	Q = (a * b)	Lower word of multi-word multiply result.
MultS	Q = (a * b) >> 12 Q = min(Q, 0x7.ff) Q = max(Q, -0x8.00)	Modulate with saturation.
PassA	Q = A	
SelectA	Q = flag ? A : B	Conditional. No branches so evaluate both paths and select the correct result.
SelectB	Q = flag ? B : A	Inverse conditional.

Saturate	If (Div2)
	Q = min(A, 0x0.00)
	Q = max(Q, 0x1.00)
	else
	Q = min(A, -0x1.00)
	Q = max(Q, 0x1.00)
		Saturate to [0, 1] if *Div2* field is set, otherwise to [-1, 1].

Opcode	Description	Notes
Arg	Nop	Load base address D from the *WEMode*, *FlagMode* and *Div2* fields. No result written.

5 Initialization

5.1 Memory Allocation (typical positions for LB, FB)

TBA

5.2 Page Tables

Page Table Initialization and management is described in detail in section 2.3.1, Address Translation Initialization.

5.3 Context Record

TBA

5.4 Registers

5.5 Programs

5.5.1 Program Initialization

The P10 architecture contains five programmable units. These are the Pixel Address unit, the Pixel unit, the Texture Coordinate unit, the Vertex Shader unit and the Pixel Shader unit. Each of these units contains a program store that holds the instructions that can be executed by the unit.

Programs can be an arbitrary number of instructions in length. The last instruction in each program includes information that allows the unit to know when it has reached the end of the program. This feature allows programs to be placed anywhere within the program store and if necessary the programs can even wrap around the end. It is therefore only necessary for the programmer to download the program into the program store, set the program start address, and then run the program.

The program store can be considered simply as an array of instructions, as shown in Figure 1. The address of the program is the location of the program within the program store. Note that the size of the program store varies from unit to unit, for example the program store within the pixel address unit can hold 32 instructions whereas the pixel shader unit holds 128 instructions.

	Address	Instruction
First program start	0	Add(r0, A0, tileX);
	1	Add(r0, A0, tileY);
	2	LoadXY(r0, r1);
	3	SendSourceAddr(buf4, puReg1);
	4	SendDestAddrAndTile(buf1,puReg0,First););
	5	SendSourceAddr(buf4, puReg1);
	6	LoadXYFromTile();
	7	SendDestAddrAndTile(buf0, puReg0, Last);
Second program	8	SendDestAddrAndTile(buf0, puReg0, Only);
Third program	9	SendDestAddrAndTile(buf4, puReg0, Only);

Figure 1. An example program store containing three pixel address unit programs.

5.5.2 Specifying program start addresses

The P10 program assembler translates each program into an array of 32 bit instruction data. The assembler also allows the programmer to specify the address of the program but this is only necessary if the program includes absolute jump instructions. In all other cases it makes more sense to set the program address to zero and use relative jumps because this allows the program to be placed anywhere within the program store of the unit.

5.5.3 Downloading programs

The mechanism for downloading programs is similar for each unit with the one exception of the pixel address unit. The pixel address unit uses a slightly different approach because the program store is quite small, only 32 instructions, and the pixel address unit instructions are only 16 bits in size.

5.5.4 Downloading pixel address unit programs

Pixel address unit instructions are 16 bits in size. Consequently pairs of instructions are mapped into the program store as 32 bit registers within the unit. The registers map to even addresses in the program store so **FBProg0** holds instructions 0 and 1, **FBProg1** holds instructions 2 and 3, and so on.

Pseudo code for downloading the pixel address unit programs from Figure 1:

```
extern ULONG pixelAddressProg1[4];
extern ULONG pixelAddressProg2[1];
extern ULONG pixelAddressProg3[1];

FBProg0= pixelAddressProg1[0];
FBProg1= pixelAddressProg1[1];
FBProg2= pixelAddressProg1[2];
FBProg3= pixelAddressProg1[3];
FBProg4= pixelAddressProg2[0] | (pixelAddressProg3[0] << 16);
```

Note: The pixel address unit only supports absolute jump instructions. If you use jumps within a pixel address unit program then you must download the program to the same address that you specified within the source code.

5.5.5 Downloading other unit programs

The other programmable units require that you set a program address register and then download the program. The following example is for the pixel unit but the procedure for the other units is exactly the same.

Pseudo code for downloading two pixel unit programs:

```
extern ULONG pixelProg1[8];
extern ULONG pixelProg2[4];


PixelProgramAddr= 0;
PixelProgramData= pixelProg1;

PixelProgramAddr= 4;
PixelProgramData= pixelProg2;
```

Note that the **PixelProgramData** register should be written to using a **PixelProgramData** hold tag that indicates the number of instructions that are to be downloaded. The 32bit pixel program data then follows.

5.5.6 Setting program start addresses for tile programs

Each programmable unit has a mode register that contains the start address of the program to run when a tile register is received. Some units can run different programs depending on the program id within the tile.

Pseudo code for setting the first, middle, and last program addresses in the pixel shader unit for programs at address 0, 5 and 15:

ShaderMode= TileEnable | TileAddrFirst(0) | TileAddrMiddle(5) | TileAddrLast(15);

Once the program, or programs, have been downloaded and the start addresses have been set it is then only necessary to perform the drawing operation which will generate the tile commands that will cause the program to run.

5.5.7 Running programs

Sometimes it is necessary to run programs manually. One example of this is to allow the context unit to restore the values of local registers after a context switch. To do this download a program that initialises local registers and run it by writing to the **RunProgram** register in the unit. After a context switch the program will automatically be re-run, thereby restoring the state of the local registers.

5.6 Video Output

This section describes how to program the Video Unit to a given video mode. P10 supports analog and digital output. It has two DACs, enabling it to drive two monitors independently. The Video Unit's registers are duplicated to permit dual head configurations. The primary head's registers are at hex offset 2000 from the base address of the control region; the secondary head's registers are at hex offset 8000. The remainder of this section concentrates on programming the Video Unit to drive a single monitor. Section 5.6.3 describes the differences between single and dual head configurations.

Most of the Video Unit's registers are indexed. To access the registers in this mode, set VideoIndexControl.AutoIncrement to 1, then write the register's index to the **VideoIndexLow** and **VideoIndexHigh** registers. The indexed register can then be read / written via the **VideoIndexData** register. By setting **VideoIndexControl.***AutoIncrement* to 1, each read/write increments the register index, allowing consecutive bytes to be accessed in turn.

The indexed registers of the Video Unit can also be accessed directly by forming an address from the base address of the control region plus the offset of the Video Unit registers plus the index of the register. The P10 decodes the byte-enables for read / write accesses, enabling the programmer to access 8, 16, or 32 bits at a time.

When writing to an indexed register, it is possible to OR/AND the contents with the value written. To OR the value with a register, add hex 400 to its index. To AND the value with a register, add hex 800 to its index.

5.6.1 Programming the Video Mode, RAMDAC and LUTs

Programming the video mode can be broken down into the following steps:

- disabling the current video mode;
- initializing the video channels;
- initializing the LUT(s);
- programming the video timing registers; and
- enabling the new mode.

5.6.1.1 Disabling the current video mode.

Video output is disabled by setting **VideoTiming.***TimingEnable* to 0. If the VGA is running, it must be disabled by setting **VGAControlReg.***EnableVGADisplay* to 0, to allow video output to be driven by the Video Unit.

5.6.1.2 Initializing the video channels.

The Video Unit output is composed from four layered channels, from the lowest layer upwards, these are: Underlay, Main, Overlay and Cursor. Each channel is independent of every other. At initialization time only the Main channel is likely to be enabled. To initialize the Main channel:

- set **VideoMainAddress** to the offset of the screen memory, from the start of the framebuffer;
- set **VideoMainXStart** & **VideoMainXEnd** to the width (in pixels) of the desired video mode; similarly, set **VideoMainYStart** & **VideoMainYEnd** to the desired height;
- set **VideoMainStride** to the byte stride between consecutive scanlines required for the desired video mode. The stride is independent of the screen dimensions to enable scanlines to be aligned optimally;
- set **VideoMainFormat** to the pixel format required for the mode;
- ensure **VideoMainPan** is set to 0;
- ensure that color keying and logic ops are disabled in **VideoMainKeyTest** and **VideoMainKeyOp**;
- ensure that **VideoBufferControl.Main** is set to 0 (Single Buffer);
- set **VideoTiming.MainEnable** to 1. The main channel won't actually be enabled until **VideoTiming.TimingEnable** is set to 1.

Finally, set **VideoUpdate.MainReg** to 1 to inform the Video Unit that the Main channel's registers have been updated.

5.6.1.3 Initializing the LUT(s).

There are two LUTs. In 8bpp color index modes, the most common configuration will be to load the palette into LUT0 and the Gamma Correction table into LUT1. For RGB modes, LUT0 will not generally be used (though see the register definition for alternative uses for LUT0). LUT data is accessed via the **VideoPaletteWriteAddress** and **VideoPaletteData** registers. Both LUTs are accessed using the same registers, the appropriate LUT is selected by setting **VideoControl1.***AccessLUT* to either 0 (LUT0) or 1 (LUT1).

To load the color index palette, set **VideoLUT0**'s *Mode* to 1 (ColorIndex), it's *MainEnable* to 1 and *Width* to the width of each color component. Next, load the palette data into the LUT. Six and eight bit color component's are written as one byte per R, G, B component; these are scaled up to ten bit values by P10, if **VideoControl1.***ExtendLUT* is set to 1. Ten bit component's are written as two bytes each, only the bottom ten bits are significant. Set VideoPixelMask to 0xff (this is only applicable to LUT0) to indicate that the index to the palette isn't masked (assuming that all 256 entries of the LUT have been programmed).

To load the Gamma Correction table, repeat the process described above, this time for LUT1 (**VideoLUT1.***Mode* should be set to 0).

5.6.1.4 Programming the Video Timing Registers.

This part of the initialization comprises three steps: setting-up the dot clock, setting-up the video timing counters and setting-up other DAC control registers.

The dot clock ticks once for each pixel read by the DAC. Before it can be re-programmed it must be stopped by setting **DClk0Control.***State* to 0 (DriveLow). A PLL must be chosen for the clock source. Any PLL can be chosen to drive the clock, but this example uses PLL1:

1. Set **PLL1Control.***Enable* to 0 (Disable).
2. Next, calculate the values of the PLL *Prescale*, *FeedbackScale* and *PostScale* variables required to generate a PLL output at the desired dot clock frequency, write these to **PLL1PreScale**, **PLL1FeedbackScale** and **PLL1PostScale**.
3. Set **PLL1Control.***Enable* to 1 (Enable) and **PLL1Control.***Ref* to 0 (Internal). The PLL internal reference clock runs at 14.3182MHz.
4. Wait for PLL1 to lock to the desired frequency by waiting for **PLL1Control.***Lock* to go to 1.
5. Finally, it is safe to enable the DClk by setting **DClk0Control.***Src* to 4 (PLL1) and **DClk0Control.***State* to 2 (Run).

.

Now set-up the video timing counters. The horizontal (pixel) counters are: **VideoHSyncStart**, **VideoHSyncEnd**, **VideoHBlankEnd** and **VideoHTotal**. The vertical (scanline) counters are: **VideoVSyncStart**, **VideoVSync**, **VideoVBlankEnd** and **VideoVTotal**. These values are derived from the VESA timings.

Other DAC control registers required for setting-up the video timing include:

- **VideoDACControl**, normally only the *Pedestal* bit will be set to 1;
- **VideoDACSyncControl**, normally, only **HSyncCtl** & **VSyncCtl** need to be set to the correct polarity of the sync signals.

5.6.1.5 Enabling the New Video Mode

To enable the new mode, set **VideoTiming.***TimingEnable* to 1.

5.6.2 Using Video Scaling

Video scaling is not normally required when programming the video, however, some low resolutions may require scaling if their X and / or Y dimensions are too small to be supported well (or at all) by the monitor.

The simplest solution is to double the horizontal and / or vertical mode timing counters, in turn requiring the dot clock to be doubled or quadrupled. If the horizontal timings have doubled, set **VideoScale.***HScale* to 8 in order to output the same pixel twice. This means that, although the output resolution is twice what it should be, it still looks correct to the viewer. To enable horizontal scaling, **VideoControl0.***PixelScale* must also be set 1. Similarly, if the vertical timings have doubled, set **VideoScale.***Vscale* to 8 in order to output the same line twice. To enable vertical scaling, **VideoControl0.***LineScale* must also be set to 1.

Video scaling can also be used to enable support for additional modes for digital monitors that don't provide native scaling.

5.6.3 Dual Head Video Output

Configuring the second head is almost identical to the procedure for configuring the first, described in 5.6.1, this section outlines the major differences.

Note: ***The second head has it's own set of Video Unit registers beginning at hex offset 8000 from the start of the control region.***

5.6.3.1 Disabling the current video mode

The second head doesn't drive the VGA so it is not necessary to program the **VGAControlReg** register.

5.6.3.2 Initializing the video channels.

The second head shares the framebuffer with the primary head, which means that the main channel's **VideoMainAddress** must point to a different region of the framebuffer. The second head is independent of the primary and does not need to be programmed to the same mode.

5.6.3.3 Initializing the LUT(s).

This step is identical for both heads.

5.6.3.4 Programming the Video Timing Registers.

This part of the initialization comprises three steps:

- setting-up the dot clock
- setting-up the video timing counters
- setting-up other DAC control registers.

Setting-up the dot clock for the secondary head differs from the primary because the clock and PLL control registers are shared by both heads. Therefore the second head must use DClk1 and PLL0 (PLL1 is used by the primary head in 5.6.1, we can assume that PLL2 and PLL3 are being used to drive MClk and KClk).

DClk1Control will be set-up identically to **DClk0Control**, except for the **Src** field, will must be set to 3 (PLL0). PLL0 is a little different from the other PLLs as it has four sets of registers. **PLL0Select** determines which set of registers are used. Sets A and B are reserved for the VGA, leaving C and D available. Apart from selecting the register set for PLL0 and programming that set's PLL registers, setting-up PLL0 is otherwise identical to setting up PLL1.

5.6.3.5 Genlocking

When configuring dual head output, the best visual quality is achieved by *Genlocking* the heads so that they remain in sync. This is especially important when adding support for stereo glasses. Genlocking is available for heads that share the same vertical refresh rate, but works better when they also share the same horizontal refresh rate (which will be true if both heads have the same resolution). To configure genlocking, each head's **VideoGenlock** register needs setting-up. The following fields of the register are common to both heads:

- **Mode**. Set to 2 (Internal), because both heads are on the same P10;
- **LockStereo**. Set to 1 if the heads are configured for stereo output; and
- **VOnly**. Set to 1 if the heads have different resolutions.

In addition, the **Head** field must be set to the index of the other head, i.e. **VideoGenLock[0].Head** = 1 and **VideoGenLock[1].Head** = 0.

On some boards, it might not be possible to exactly match the refresh rates on both heads, typically due to board layout issues. It is possible to correct any small descrepencies using the genlock specific video timing counters: **VideoGenlockH** and **VideoGenlockV**. These are also defined per head.

Note: P10 does not support Sync on Green.

5.6.4 Digital Video Output

Section 5.6.1 described how to program the video mode for an analog monitor. To enable digital video output, only the DAC control registers need be programmed differently.

In **VideoDPSyncControl** (the digital port version of **VideoDACSyncControl**), the **VSyncCtl** & **HSyncCtl** fields should be set to the correct polarity of the sync signals. If analog video output is not required, it can be disabled by setting **VSyncOverride** and **HSyncOverride** in **VideoDACSyncControl** to 2 (ForceLow) to force the analog monitor into DPMS 'off' mode.

In **VideoDPMode**, set **Mode** to enable output from the digital port. Normally, this will be set to 1 (SinglePixel), but for very high resolution modes outside the frequency range of the PLLs, **Mode** can be set to 2 (DoublePixel), allowing the dot clock to run at half the frequency otherwise required. The **StrobeDelay** and **StrobeInvert** fields are specific to the board design and are not dealt with here.

5.6.4.1 The Digital Port.

The **VideoDigitalPortControl** register must also be set-up to enable digital output. This register controls the behavior of the *digital port* and its set-up is dependent on the design of the board. The digital port has two input pipes (In0 and In1) which aren't discussed here as they are only important when interleaving the output of two P10 chips. The digital port also has two output pipes (Out0 and Out1). In addition, the output from the two DACs are inputs to the digital port.

On boards with two digital outputs DAC0's RGB data is routed to Out0 and DAC1's RGB data is routed to Out1. For boards that have only a single digital output, DAC1 and Out1 are not used. The DAC inputs to the digital port and the output pipes are all 24 bits wide. Internally, the digital port has two 12 bit channels which can be configured as one 24 bit channel when only one digital output is in use. The table on the next page illustrates the possible configurations of the digital port and their effect on the **VideoDigitalPortControl** register.

Configuration*	Out0	Out1	VideoDigital PortControl
(A) Single head board, or dual head board when only the primary head is enabled	Channel0 and Channel1 co-operate to pass through DAC0 RGB data	Unused	Mode = 3 (Out0)
(B) Dual head board when only the secondary head is enabled	Unused	Channel0 and Channel1 co-operate to pass through DAC1 RGB data	Mode = 5 (Out1)
(C) Dual head board, where each head is independent	Channel0 passes through DAC0 RGB data at double data rate (to maintain 24 bit output)	Channel1 passes through DAC1 RGB data at double data rate (to maintain 24 bit output)	Mode = 1 (Shared) DoubleEdge=1 (On)
(D) Dual head board, where the secondary head is a clone of the primary	Channel0 passes through DAC0 RGB data at double data rate (to maintain 24 bit output)	Channel0 passes through DAC0 RGB data at double data rate (to maintain 24 bit output)	Mode = 6 (DualOut0) DoubleEdge = 1 (On)
(E) Dual head board, where the primary head is a clone of the secondary	Channel1 passes through DAC1 RGB data at double data rate (to maintain 24 bit output)	Channel1 passes through DAC1 RGB data at double data rate (to maintain 24 bit output)	Mode = 7 (DualOut1) DoubleEdge = 1 (On)

* Configuration is specified only in terms of the digital port. For instance if the primary head is analog and the secondary head is digital, configuration (B) is applicable.

6
Synchronization

6.1 Synchronization with Core and with VTG

6.1.1 Synchronizing Video Channel Updates with Video Output

The **VideoUpdate** register synchronizes each video channel's registers with video output. The **VideoUpdate** register has two flags for each channel, e.g. for the Main channel:-

6.1.2 VideoUpdate.MainBuffer

Set after updating the channel's base address registers: VideoMainAddress and VideoMainStereoAddress.

6.1.3 VideoUpdate.MainReg

Set after updating the other channel-specific registers: VideoMainPan, VideoMainStride, VideoMainFormat, VideoMainXStart, VideoMainYStart, VideoMainXEnd, VideoMainYEnd, VideoMainKeyTest, VideoMainKeyOp, VideoMainKeyRGBA and VideoMainBlend.

After writing the update flags, the **VideoUpdate** register can be read back to determine when the registers have been updated. The VideoBufferControl determines when a channel's registers are updated. If the channel is in single buffer mode, the updates are applied immediately, otherwise (double or triple buffered mode) the updates are applied during frame blank.

Synchronizing Video Channels Updates Between Channels

The **VideoLock** registers can be used to ensure synchronization between channels. Two independent locks can be configured per head using **VideoLock0** and **VideoLock1**. A Lock ensures that video updates to a locked channel are only latched when updates are available to the other channels specified in the Lock register. This could be used, for example, to ensure that buffer updates to the Main and Overlay channels are applied during the same frame blank.

6.1.4 Synchronizing the Core with Video Output

The **SyncToVTG** command is used to stall the core units until a VTG sync point is reached. The sync point is specified in the **VideoGPEvent** register and can be set to occur when any of the video channel's is updated. Generally, the **SyncToVTG** command will be sent when the programmer needs to stall the command stream until the next frame blank.

6.2 Invalidating Caches

The **CacheControl** command provides the following bit field controls:

- bit 0 = Flush LB Cache
- bit 1 = Invalidate LB Cache
- bit 2 = Flush Pixel Cache
- bit 3 = Invalidate Pixel Cache
- bit 4 = Invalidate Texture Primary Cache
- bit 5 = Invalidate Texture Secondary Cache

6.2.1 Texture Cache Control

When simply binding to a new texture (i.e. updating the **TextureBaseAddress** register state) a **CacheControl** command that only invalidates the Texture Primary Cache is sufficient.

If the data for a currently bound texture is being modified, for example by a texture download operation, or a partial sub-texture update, or an update to the texture border colour state, then both the primary and the secondary texture cache need to be invalidated.

An optimised method for invalidating only a part of the texture secondary cache is available and useful when only a small amount of the current texture data has been altered. The **InvalidateSecondaryTextureCache** command is sent with the start address of the data to be invalidated. The address is a byte address but the bottom 2 bits are ignored. The address will be incremented by 4 to get to the next tile to invalidate. The **InvalidateSecondaryCacheCount** command is then sent with a count of the number of texture cache entries to invalidate (in units of 32bit groups).

6.2.2 Pixel and Local Buffer Cache Control

One example of a situation where it can be necessary to flush and invalidate the pixel and/or local buffer cache is when the host uses the bypass mechanism to update local memory directly rather than via the normal command stream.

Another example is an implementation of a depth buffer clear by temporarily setting one of the **FBBuffer** registers to operate on a local buffer region rather than its normal pixel buffer region. In this case the safest option is to flush and invalidate the pixel and local buffer caches before and after implementation of the clear operation.

6.3 Interrupts

6.3.1 Interrupts & Synchronization

6.3.1.1 CommandIDApi Interrupts

CommandIDApi commands can be used to determine how much of a DMA buffer the graphics chip has read in. When the graphics chip encounters a **CommandIDApi** command it takes the software-definable 30-bit *CommandID* field and stores it in the

CommandIDUsr register for the current context, it will additionally generate an interrupt if the *Intr* bit is set.

CommandIDApi with interrupts is very useful because it can be 'sprinkled' throughout the command stream. When a DMA buffer is full the software can go to sleep and wait for an interrupt to occur rather than poll the **CBufWrPtrUsr** to check for free space. Polling is generally inefficient on CPU resources and the reading of the **CBufWrPtrUsr** register causes AGP bus traffic, which is also inefficient.

6.3.1.2 Example code for a rendering routine

```
...
if (My DMA buffer is almost full)
{
                // Send a CommandID, enable interrupt, then go to sleep
MyCommandIDAPI.CommandID = 0x10;          // Set up commandID
MyCommandIDAPI.Intr      = 1;             // Generate an interrupt
AddCommandToDMABuffer(CommandIDAPITag, MyCommandIDAPI);
SleepUntilEvent (MyCommandIDEvent);       // Wait for interrupt
                // It's now safe to start adding more things to the DMA buffer
}
...
```

Ensure that there is enough room at the end of the DMA buffer for a command ID tag/data pair and never sprinkle one in the middle of a large operation such as a download, as these operation need to be atomic.

6.3.1.3 Example code for an interrupt routine:

```
...

if (UserRegs.CommIntrMask & MY_CONTEXT_BIT)          //Handle MY interrupt

{
    Assert (UserRegs.CommandID[MY_CONTEXT] == 0x10, "Bad CommandID");
    UserRegs.CommIntrMask = MY_CONTEXT;              // Clear interrupt
WakeupEvent (MyCommandIDEvent);
    }
...
```

Note: Ensure that Command interrupts are enabled in the Interrupts register.

6.3.1.4 Sync Interrupts

Sometimes, rather than know how far through a DMA buffer a graphics chip has read it is important to know, additionally, whether the graphics chip has actually processed (i.e. rendered) the commands in the DMA buffer. This is what **Sync** commands are for, like **CommandIDApi** commands they also have a 30-bit software-definable *SyncID* field and an *Intr* bit.

Additionally, **Sync** commands also have a *flush* bit which, when enabled, causes the command to stall the pipeline until all pending AGP writes have been flushed across the bus.

Once again it is possible to determine whether a DMA buffer has been processed by inserting a **Sync** command and then polling the **SyncIDUsr** until the value of the SyncID changes, however, as with **CommandIDApi** it's much more efficient and system-friendly for the software to set the *Intr* bit and go to sleep until woken by an interrupt.

*Note: The **Sync** commands inserts a bubble in the command stream because it travels all the way through the graphics chip's pipeline to the rasterizer and back out again. This is expensive in performance terms so it is preferable to use **CommandIDApi** wherever possible.*

7

Image Download (How to, Setup)

7.1 Pixel Data

The **PixelData** tag is used to supply up to 32 bits of data for every pixel processed during the operation of a **DrawRectangle2D** command, and is most frequently of use in downloading image data from host memory into the framebuffer.

PixelData tags are 32 bits wide and can either contain unpacked data for a single pixel in each tag, or packed data of 4-, 8-, 16-, 24- or 32-bits per pixel. The Rasteriser Unit unpacks data from a series of tags and forwards it to the Pixel Unit where it is accessed by the relevant fragment processor.

Whenever pixel data is provided to the rasteriser in this way, it is accepted in scanline order (each scanline with increasing X), meaning that the tiled nature of the P10 is transparent to the host.

Image downloads (native or translating) performed this way use the following main units:

- Rasteriser
- Pixel Address Unit
- Pixel Unit

Additionally, image downloads requiring a palettised translation can be achieved using either the texture pipe or the GPIO subsystem:

- Texture Coordinate Unit
- Texture Address Unit
- Texture Index Unit
- Shading Unit

Or:

- Vertex Index Unit
- Vertex Data Unit

7.1.1 Native download setup

For native downloads, only the Pixel Address Unit and Pixel Unit require programs to be loaded. The Pixel Address Unit program simply needs to send the destination address for the buffer being used:

```
Program( pixelAddressDownloadProg, 0 )
    SendDestAddrAndTile( buf0, puReg0, Only);
```

The Pixel Unit program should read the pixel data from the per-fragment data and output it to the framebuffer:

```
Program( pixelDownload24bppNative, 0)
    E = Fragment[ 0 ]  C[ 0 ] = PassA( E );
    E = Fragment[ 1 ]  C[ 1 ] = PassA( E );
    E = Fragment[ 2 ]  C[ 2 ] = PassA( E )  Done;
```

This program could also be used to perform raster operartions combining the downloaded data with the destination and/or a solid colour (loaded into the global registers) or brush pattern (pre-loaded into the local registers). For example:

```
Program( pixelDownload24bppNativeXor, 0)
    E = Fragment[ 0 ]  C[ 0 ] = Xor( P[ 0 ], E );
    E = Fragment[ 1 ]  C[ 1 ] = Xor( P[ 1 ], E );
    E = Fragment[ 2 ]  C[ 2 ] = Xor( P[ 2 ], E )  Done;
```

After loading the programs and setting the **PixelMode** and **FBMode** tags appropriately, the following tags need to be sent to complete the setup:

Tag	Requirements
FBBuffer*N*	Set *Width, PixelBytePitch, PixelSize, SubFieldStartByte* & *SubFieldStartCount* as appropriate for source. Set *ReadEnable* bit if the Pixel Unit program reads the destination buffer (or specify with **FBBufferReadEnables** tag).
FBBaseAddr*N*	Set to buffer base address.
FBBufferEnables	Enable buffer *N* only.

Additionally, the **RasterMode** tag should be sent. The important fields to specify which determine how the **PixelData** tags will be interpreted are:

Field	Requirements
ByteSwap	Set according to host memory layout: 0=ABCD (no swap), 1=BADC, 2=CDAB, 3=DCBA .
Mirror	Set to enable mirroring of bits in the download data (bit 0 will be swapped with bit 31, bit 1 with bit 30 etc).
PixelSize	Specify size of packed download data (use 32bits if unpacked): 0=4 bits, 1=8 bits, 2=16 bits, 3=24 bits, 4=32 bits.
Invert	Set to enable inversion of each bit in the downloaded data.
NibbleSwap	Set if downloading 4-bit packed data to swap the order of nibbles within each byte.

7.1.2 Native download operation

The download is performed by rasterising a 2D rectangle of the required dimensions in the destination buffer, followed by sufficient **PixelData** tags to provide image download data for each pixel in the rectangle.

The origin of the rectangle is specified using the **RectanglePosition** tag, which has the following fields:

Field	Requirements
X	2's complement x-coordinate (14 bits)
Y	2's complement y-coordinate (14 bits)

The rectangle is then defined with the **DrawRectangle2D** tag:

Field	Requirements
Width	Width of rectangle (0…8191)
Height	Height of rectangle (0…8191)
Operation	Set to 1 (*SyncOnHostData*). This infers the *IncreasingX* and *IncreasingY* fields.
PixelsPerScanline	Set to 1 (process 8 pixels per scanline per tile).
PackedBitMask	Set to 0.

Following the definition of the rectangle, the image data is sent as a series of **PixelData** tags. Each scanline should be sent in turn with sufficient data to cover the entire scanline (if this is not the case, the Rasteriser will stall indefinitely waiting for the remainder). When sending packed image data, note that the Rasteriser will discard any remainder at the end of each scanline, so each new scanline should start with a new **PixelData** tag.

To minimise the volume of data sent from the host, it is recommended that the number of **PixelData** tags required for a scanline is calculated, and a hold tag sent followed by the raw data for the entire scanline.

7.1.3 Translating downloads

For colour translations which do not use a palette, the only alteration to the method already outlined is to the Pixel Unit program, which can be modified to carry out the translation on the image data before outputting to the framebuffer.

An example program, to translate 24bpp image data to a 15bpp (555) framebuffer is shown below:

```
Define (scr_0, 0)
Define (scr_1, 1)
Define (scr_2, 2)
Define (scr_3, 3)
Define (img_0, 4)
Define (img_1, 5)
Define (img_2, 6)

Program(pixelDownloadTrans24bppto15bpp, 0x00)
E = Fragment[ 0 ]     W[img_0] = PassA( E );          // W[img_0]=BBBBBBBB
E = Fragment[ 1 ]     W[img_1] = PassA( E );          // W[img_1]=GGGGGGGG
E = Fragment[ 2 ]     W[img_2] = PassA( E );          // W[img_2]=RRRRRRRR
E = 0x20 W[scr_0] = MultU( A[img_0], E );             // W[scr_0]=000BBBBB
E = 0x04 W[scr_1] = MultL( A[img_1], E );             // W[scr_1]=gggggg00
```

```
E = 0xE0 W[scr_1] =   And( A[scr_1], E );          // W[scr_1]=ggg00000
E = 0x04 W[scr_2] = MultU( A[img_1], E );          // W[scr_2]=000000GG
E = 0xF8 W[scr_3] =   And( A[img_2], E );          // W[scr_3]=RRRRR000
E = 0x80 W[scr_3] = MultU( A[scr_3], E );          // W[scr_3]=0RRRRR00
        C[ 0 ] =    Or( A[scr_0], B[scr_1] );                 // gggBBBBB
        C[ 1 ] =    Or( A[scr_2], B[scr_3] )  Done;           // 0RRRRRGG
```

7.1.4 Palettised translating downloads

The Pixel Unit does not have suitable instructions or data storage to perform palettised downloads, so another mechanism must be employed. There are two basic methods, one using the texture pipe and the other using the GPIO subsystem.

The method chosen depends on several factors. The advantages of the texture pipe method include:

- Ease of implementation
- Speed of host operation
- Low host resource usage
- Unrestricted palette size

The main disadvantage of this method is the speed of P10 operation; the texture subsystem splits each tile into four sub-tiles for processing, and hence only 4 pixels can be processed at a time, rather than 8 as with the GPIO method.

7.1.4.1 Texture pipe method

The basic premise of this method is to first download a look-up table (LUT) to an area of offscreen memory (using the standard native download method) and then map this to a texture. Image download data is then used as an index into the texture map, with the result passed to the Pixel Unit for combination with the framebuffer in the usual way.

The LUT can be downloaded to a 1D or 2D texture map as desired. The following discussion uses a 1D map for simplicity.

After downloading the LUT to a suitable area of offscreen memory (using the standard native download method described previously), a **WaitForCompletion** tag should be sent with data value 0, to ensure the LUT is fully downloaded prior to being accessed by the texture subsystem. Setup for the translation is then as follows.

The Texture Coordinate Program takes the downloaded image data, uses this as an index into the texture and initiates the Shading Unit program. This example is suitable for 8-bit or 4-bit palette indices, depending on the value loaded into Global0[0]:

```
Define (PaletteTex, N)   // using texture N
Define (ShadeLoad, 1)
Define (NoFeedback, 0)

Program(TC_PaletteLookup8bppSource, 0 )
    W[0] = IntToFloat( Feedback(8, 0, Zero) );          // W[0]=downloaded data
    C[0] = Wrap( A[0], Global0[0] );                    // Global0[0]=1/256 or 1/16
    Command(FilterTexture, PaletteTex, 0, ShadeLoad, NoFeedback, Default)
    Fract( One)  Done;
```

The Shading Unit program just passes the texture value onto the Pixel Unit:

```
Program( SU_PaletteLookup24bppDest, 0 )
    C[ 0 ] = PassA( T[ 0 ]_Z );
    C[ 1 ] = PassA( T[ 1 ]_Z );
    C[ 2 ] = PassA( T[ 2 ]_Z )  Done;
```

The Pixel Address Unit program again simply needs to send the destination address for the buffer being used:

```
Program( pixelAddressDownloadProg, 0 )
    SendDestAddrAndTile( buf0, puReg0, Only);
```

The Pixel Unit program should read the pixel data from the shading data (rather than the per-fragment data as previously) and output it to the framebuffer:

```
Program( pixelDownload24bppPalette, 0)
    C[ 0 ] = PassB( F[ 0 ] );
    C[ 1 ] = PassB( F[ 1 ] );
    C[ 2 ] = PassB( F[ 2 ] )  Done;
```

After loading the programs and setting the **TextureCoordMode, ShadeMode, PixelMode** and **FBMode** tags appropriately together with the **FBBuffer*N***, **FBBaseAddr*N***, **FBBufferEnables** and **RasterMode** tags (as specified in paragraph 7.1.1), the following additional tags need to be sent to complete the setup:

Tag	Requirements
CacheControl	Set *InvalidateTexturePrimaryCache* and *InvalidateTextureSecondaryCache* bits.
TextureBaseAddress*N*	Set to LUT tile base address.
TextureAddressMode*N*	Set *MapType* to 0 (1D), *Width* to LUTsize, *PowerOfTwoTexture* to 1, *Format* to 10 (8888) and *Pitch* according to pixelsize of LUT entries: 0=8bpp, 1=16bpp, 2=24bpp, 3=32bpp.
TextureIndexMode*N*	Set *MapType* to 0 (1D), *Width* to log2(rectwidth) and *WrapU* to 1.
TextureGlobal0	Set to 1/LUTsize (used in example TCU program).

LUTsize should be specified as the lowest power of two large enough to encompass the entire LUT (eg 256 for 8bpp indices and 16 for 4bpp indices).

Once this setup has been completed, download can proceed exactly as described in paragraph 7.1.2.

7.1.4.2 GPIO method

This method is really suitable only for palettes with 8-bit indices, as the Vertex Index Unit does not support 4-bit indices. Additionally, the LUT must be supplied as unpacked 32-bit data (destinations lower than 32bpp can still be supported, since they will just discard unused portions of the data).

The basic premise of the method is to treat each scanline of image data as a sequence of vertex indices. The Vertex Index Unit sends each 8-bit index to the Vertex Data Unit; this

then accesses the appropriate value in the LUT and forwards it on to the Rasteriser as a **PixelData** tag.

The first stage of this method is to transfer the LUT into host memory mapped into the P10 address space, as an array of 32-bit values as described above. The LUT must start on a 32-bit aligned address. Additionally, each scanline's packed 8-bit image data must be transferred into host memory mapped into the P10 address space. It is not necessary for each scanline to start on a 32-bit aligned address, however.

Next, appropriate programs are loaded into the Pixel Address Unit and Pixel Unit; these are identical to those used for native downloads, since all the translation occurs prior to the Rasteriser.

After loading the programs and setting the **PixelMode** and **FBMode** tags appropriately, the **FBBuffer*N***, **FBBaseAddr*N***, **FBBufferEnables** and **RasterMode** tags should be set as specified in paragraph 7.1.1, with the *Pixel**Size*** field in **RasterMode** set to 4 (since the GPIO subsystem forwards **PixelData** to the Rasteriser as unpacked 32-bit data regardless of the destination depth). Then the Vertex Index Unit and Vertex Data Unit can be set up as follows:

Tag	Requirements
VertexCacheMode	Set to 0.
VertexIndexBounds	Set *Base* to 0, *Count* to 256.
VertexParameterMsg*Q*	Set *Tag* to **P10_PixelData_Tag**, *Size* to 0, *Send* to 1.
VertexDataBuffer*M*	Set *Addr* to address of LUT (in 32-bit words), *DataSize* to 0, *DataStride* to 0, and *ByteSwap* according to host memory layout: 0=ABCD (no swap), 1=BADC, 2=CDAB, 3=DCBA
VertexIndexBuffer*N*	Set *Addr* to address of image data (in 32-bit words), *IndexSize* to 0, *Enable* to enable data buffer *M* only, and *ByteSwap* according to host memory layout: 0=ABCD (no swap), 1=BADC, 2=CDAB, 3=DCBA
VertexParameterEnable	Set *Enable* to enable parameter *Q* only.
VertexDataBufferEnable	Set *Enable* to enable data buffer *M* only, *CacheMode* to 0.
VertexIndexBufferEnable	Set *Enable* to enable index buffer *N* only.

Following setup, each scanline is downloaded by sending a **RectanglePosition/ DrawRectangle2D** pair as specified in paragraph 7.1.2 followed by a **VertexIndexBufferLookup** tag with the following fields:

Field	Requirements
First	Set to byte offset of the first index in the scanline from the start of the image data.
Count	Set to the number of indices in the scanline (ie scanline width in pixels).

Finally, a **CommandID** tag can be sent with a unique ID to identify the download; receipt of this ID value indicates that the GPIO subsystem has completed its part of the download

operation, and the host memory used to store the image data and LUT may safely be reallocated.

7.1.5 Downloads with patterned brushes

Section 7.1.1 briefly mentioned that the Pixel Unit program could combine the source (present in the fragment data registers) with destination and/or solid or patterned brushes. Solid colour brushes are easily handled by loading the colour into the global registers; however, patterned brushes require different colours per fragment, and the per-fragment data registers are already being used to hold the image download data.

The solution is to run a short Pixel Unit program to pre-load the brush into the local registers of each fragment processor. A suitable Pixel Unit program to load a 24bpp brush into local registers 12-14 would be:

```
Program( pixelBrushDownload24bpp, 0 )
    E = Fragment[ 0 ]  W[ 12 ] = PassA( E );
    E = Fragment[ 1 ]  W[ 13 ] = PassA( E );
    E = Fragment[ 2 ]  W[ 14 ] = PassA( E )  Done;
```

After loading this program, the brush data can be loaded directly into the fragment registers using the UserFragData0 to UserFragData63 tags. The RunPixelProg tag can then be used to transfer the fragment data into the local registers of each fragment processor (only the RunAddress field need be specified, with the EnableRun bit set).

Once this operation has been performed, any of the download methods can include the brush by accessing the relevant local registers in the Pixel Unit program. The only point to note is that the brush must be reloaded after a context switch, since the local registers are not saved as part of the context.

For more about Patterned Downloads, see 2D logical operations – Color Pattern Operations

7.2 Texture maps (download, MIPmap generation)

Texture map downloads need to write the texture data into a memory location from which the core can subsequently access the data during rasterisation. This data can either be directly into video memory using the host or it can be downloaded through the core as per pixel data downloads.

Direct writing of the texture using the core requires no chip setup. Therefore, this section describes what chip setup is necessary to perform the download through the core.

To perform the download a frame buffer register is pointed at the download position. The Rasteriser Unit is setup to provide scanline conversion of the download area and to synchronize this with host provided data. Formatting of the downloaded data is then performed in the Pixel Unit before the data is finally written out to memory.

A simple example pixel unit program to perform a download of a RGBA texture would look something like this:

```
E=Fragment[0] C[0]= PassA(E) ;
E=Fragment[1] C[1]= PassA(E) ;
E=Fragment[2] C[2]= PassA(E) ;
E=Fragment[3] C[3]= PassA(E) Done;
```

In the example program, the data to be output by the pixel unit is received for each fragment using the “fragment” data registers and simply output to the corresponding channel of the texture memory. This pixel unit program assumes that a standard Pixel Address Unit program will have been loaded.

Apart from loading the correct programs into the pixel address unit the destination buffer needs to be set-up to receive the data. The **FBBufferAddr** Tag should be set to point to the download address and the **FBBuffer** tag should be sent to configure the buffer to the correct size and depth.

Tag	Requirements
FBBuffer*N*	See below.
FBBaseAddr*N*	Set to buffer base address.
FBBufferEnables	Enable buffer *N* only.
RectanglePosition	Set to the position of the texture (or sub-texture) update.
DrawRectangle2D	See below.

The **FBBufferN** tag should be set up as below:

Field	Requirements
Width	Width of the texture in tiles (if the texture is not a whole number of tiles wide then this value is the next)
PixelBytePitch	Pitch of the texture.
PixelSize	Pitch of the texture – 1.
SubFieldByte Count	Pitch of the texture – 1.

The **DrawRectangle2D** tag should be set up as below:

Field	Requirements
Width	Width of the texture (or sub-texture update) in pixels
Height	Height of the texture (or sub-texture update) in pixels
Operation	Sync on host data
IncreasingX	true
IncreasingY	True – sync on host data needs us to rasterise in a known order
PixelPerScanLine	“8” – for pixel unit downloads

For example, the psuedo-code for a typical texture download, size dwWidth * dwHeight * dwPitch, would look like this:

dwOffsetX = 0;

```
dwOffsetY       = 0;
dwWidthInTiles = (dwWidth + tileWidth – 1) / tileWidth;

SEND_TAG (FBBaseAddr0_Tag,  dwAddrInTiles);

SEND_TAG (FBBuffer0_Tag,                    FBBuffer(Width, dwWidthInTiles) |
      FBBuffer(PixelBytePitch, dwPitch) |
      FBBuffer(PixelSize, dwPitch - 1 ) |
      FBBuffer(SubFieldByteCount, dwPitch - 1 ) );

P10_DrawRectangle2D DrawRectangle2D ;
DrawRectangle2D.bits.Width                                          = dwWidth ;
DrawRectangle2D.bits.Height
            = dwHeight ;
DrawRectangle2D.bits.Operation
            = SyncOnHostData ;
DrawRectangle2D.bits.IncreasingX                                    = TRUE ;
DrawRectangle2D.bits.IncreasingY                                    = TRUE ;
DrawRectangle2D.bits.PixelsPerScanline                              = Pixels8 ;

// rasterise the rectangle
SEND_TAG (RectanglePosition_Tag, dwOffsetX | (dwOffsetY << 16));

SEND_TAG (DrawRectangle2D_Tag,                          DrawRectangle2D.word );

/* Send the HOLD tag */
SEND_HOLD_TAG(PixelData_Tag,                          (dwWidth * dwHeight));

for (j=0; j < dwHeight; j++) {
for (i=0; i < dwWidth; i++) {
DWORD dwData                                          = GetColourValue(i,j);

WRITE_32BIT_DATA(dwData);
}
}
```

To perform sub-texture updates, the above process is identical except that only the area being updated needs to be rasterised. The position of the drawrectangle2d command (dwOffsetX, dwOffsetY) needs to be set to the correct position and the width and height adjusted accordingly. Obviously, only the data for this new area need to be sent.

To perform 3D texture downloads, each slice needs to be downloaded as if it was a separate 2D texture download. So the process again is identical to the one above. However, each slice must be downloaded to a new tile aligned address. For example, between each slice download the download address (dwAddrInTiles) must be updated between iterations:

```
dwAddrInTiles += (dwWidthInTiles*dwHeightInTiles)*dwPitch;
```

Mip-maps are a collection of pre-scaled versions of a single texture. Having the textures pre-scaled saves the processing time that would be required to scale down larger textures to make them usable on a distant surface. During rendering, the closest level of detail in the mip-map is selected so the final scaling process takes less time and introduces fewer errors. P10 natively supports textures with mip-maps.

To produce the various levels of detail for a mip-mapped texture, the initial level of detail is scaled to produce an image that is exactly half the size of the previous level. The method for producing the scaled version depends upon the user, common methods include using a box filter but any filtering method can be applied.

This texture scaling is repeated until the mip-map reaches 1x1 in size.

When downloading mip-mapped textures to the P10 three things must be ensured:

- The mip-map levels must be stored in the correct order, largest to the smallest.
- The mip-map levels must be stored in contiguous memory.
- Each mip-map level must be downloaded to a tile aligned address.

7.3 Bitmask data

The **Bitmask** register can be used to perform conversions between monochrome source data and color pixels. Each write to the register passes thirty-two bits of monochrome data to be converted into an appropriate **TileMask** or **PixelMask** for the Pixel Unit. Since bitmask operations are always performed in scanline order, each write to the **Bitmask** register results in writes to one line of thirty-two pixels across multiple tiles.

There are three common uses for bitmask data: monochrome bitmap downloads to the framebuffer, font glyph downloads to an offscreen cache, and text rendering from host memory font glyphs.

7.3.1 Opaque Monochrome Bitmap Downloads

A typical Pixel Unit program for opaque monochrome bitmap downloads will take the foreground bitmap color in the **PixelGlobal0** register, the background bitmap color in **PixelGlobal1**, and use the **PixelMask** to determine the color of each pixel. Here is such a program for an eight bit destination, which can readily be expanded to greater color depths.

```
Program(MonoDownloadBltS8, 0x10)
    E = Global[0]        C[0] = PassA(E) Flag=PM;
    E = Global[4]        C[0]& = PassA(E) Done;
```

The first instruction sets the pixels in the tile to the color specified by the low byte of the **PixelGlobal0** register, and copies the **PixelMask** into the Flag register. The second instruction sets the pixels specified in the Flag register to the color specified in the low byte of the **PixelGlobal1** register. After configuring the Rasteriser to generate the **PixelMask** from the bitmask data, this program will expand the monochrome **Bitmask** register writes into the color destination.

With the pixel unit program loaded, you now need to set up the chip to perform the download. Typically the CPU will prefer aligned thirty-two bit reads from host memory, so downloading an unaligned portion of a bitmap would incur a large overhead of masking and shifting operations. If each scanline of the bitmap is thirty-two bit aligned, the simplest

solution is to download a larger rectangle, and enable clipping so that only the correct area is actually written to the framebuffer.

You need only clip excess pixels from the left side of the rectangle as **DrawRectangle2D** will automatically clip any to the right, and, by default, bitmask data will not be split across scanlines; any excess data at the end of a scanline is thrown away. You must use the **UserScissor** registers for clipping, as **VisRect** clipping does not advance the bitmask when it clips on the left and would therefore render incorrectly.

Note that if you often download monochrome bitmaps which include many lines of a solid color, you may wish to first scan through those bitmaps to find any such areas and render them as solid fills instead. While this will increase the host software overhead, solid fills are rendered much faster than monochrome downloads and you may see an overall performance gain.

Pseudo-code for a typical download of a bitmap size (w, h) pixels from source bitmap coordinates (bx, by) to screen coordinates (x, y):

```
int endbx, endsx;
dword *bptr;

LoadPixelUnitProgram (MonoDownloadBlt);
UserScissorMinXY = (y << 16 | x);
UserScissorMaxXY = (0x3fff3fff);
PixelGlobal0 = ForegroundColor;
PixelGlobal1 = BackgroundColor;

endbx = bx + w;     // Calculate bitmap end x coordinate
endsx  = x + w;      // Calculate screen end x coordinate
bx &= ~0x1f;        // Align start to 32 bits

w = endbx – bx;      // Calculate new width
x = endsx – w;       // Calculate new screen start coordinate
bptr = bitmap + (by * bitmap_stride) + (bx >> 5);   // Calculate pointer to first DWORD

RasterMode |= (GeneratePixelMask | UserScissorEnable);
RectanglePosition = (y << 16 | x);
DrawRectangle2D =  0xE000 | SyncOnBitmask
```

Finally, write each thirty-two bit dword of bitmap data to the Bitmask register.

For best performance in DMA operations, you should create a **Bitmask** hold tag to specify the number of dwords you will write to the **Bitmask** register (one tag per line or one per download, depending on the size of your DMA buffer), and you can then perform a simple memory copy of each scanline from the bitmap to the DMA buffer.

The pixel unit program can easily be extended to support logical operations by expanding the monochrome data into a temporary register and performing the logical operation between that register and the destination; you cannot write directly to the destination as the first instruction operates on every pixel regardless of the bitmask and the result would be incorrect. An example XOR program:

```
Program(MonoDownloadBltDSx8, 0x10)
    E = Global[0]       W[0] = PassA(E) Flag = PM
    E = Global[4]       W[0.1]= PassA(E)
        C[0] = Xor(P[0], B[0]) Done;
```

This program assumes a standard Pixel Address Unit program will have loaded the destination tile into P[0]. It then expands the monochrome data into temporary register W[0] and XORs the destination with that register.

7.3.2 Rendering Host Memory Font Glyphs/Transparent Downloads

Rendering font glyphs and monochrome downloads from host memory is relatively simple when the background is transparent so that only the foreground pixels should be set in the destination and the background pixels remain unchanged. You configure the Rasteriser to generate a **TileMask** rather than a **PixelMask**, and then use a normal solid fill program in the Pixel Unit; the data written to the **Bitmask** register will be converted into a **TileMask**, which will then be used to mask writes from the Pixel Unit to the destination. This will automatically ensure that only pixels specified by the bitmask data are set in the destination.

As with opaque monochrome downloads, for transparent monochrome downloads you will again probably choose to render from a thirty-two bit aligned pixel in your source and clip the appropriate pixels on the left side of the destination rectangle; the download operation is almost identical to opaque downloads other than clearing the GeneratePixelMask bit in the **RasterMode** register and loading the normal solid fill routine into the Pixel Unit rather than the opaque download program. See section 7.3.1 above for more details.

Pseudo-code for font glyph rendering, assuming all glyphs are thirty-two bit aligned in host memory:

```
LoadPixelUnitProgram (SolidFill);
PixelGlobal0 = TextColor;

RasterMode &= (~GeneratePixelMask);    // Generate a TileMask, not a PixelMask
RectanglePosition = (y << 16 | x);
Render2D =  0xE000 | SyncOnBitmask

Finally, write each thirty-two bit dword of the glyph to the Bitmask register.
```

Again, this operation can readily be expanded to support logical operations, this time by loading your normal solid color logical operation program into the Pixel Unit. See the 2D Logical Operations section for more details on logical operation programs.

7.3.3 Font Glyph Downloads To Offscreen Cache

For the best text-rendering performance, you should download font glyphs to an offscreen memory cache, rather than downloading the glyph every time you wish to render it. As the glyphs are stored in offscreen memory as monochrome bitmaps, this is a much more complicated operation than the standard color-expanding downloads.

Because the glyph download operation is integrated so closely with text rendering, it will be covered in detail in that section.

7.4 Performing uploads

Uploads are performed in a similar way to downloads, with a 2D rectangle being rendered to the appropriate position in a buffer. The role of the Pixel Unit in this case is to output fragment data to the Host Out Unit, where it is forwarded to the Upload DMA Unit for transfer to the host.

The main units involved are:

- Rasteriser
- Pixel Address Unit
- Pixel Unit
- Host Out Unit
- Upload DMA Unit

Additionally, monochrome uploads employ the texture pipe; in particular:

- Texture Coordinate Unit
- Texture Address Unit
- Texture Index Unit
- Shading Unit

7.4.1 Upload setup

For colour uploads, only the Pixel Address Unit and Pixel Unit require programs to be loaded. The Pixel Address Unit program simply needs to send the destination address for the buffer being used:

```
Program( pixelAddressUploadProg, 0 )
    SendDestAddrAndTile( buf0, puReg0, Only);
```

The Pixel Unit program should read the pixel data from the buffer and forward it onto the Host Out Unit:

```
Program( pixelUpload32bppNative, 0)
    C[ 0.HO ] = PassA( P[0] );
    C[ 1.HO ] = PassA( P[1] );
    C[ 2.HO ] = PassA( P[2] );
    C[ 3.HO ] = PassA( P[3] )  Done;
```

If a colour translation is required from source to destination format, this can also be performed by the Pixel Unit program.

It is important to note that the program *must* pass the correct amount of data to the Host Out Unit; if too much data is forwarded, then some will be discarded; if too little data is forwarded, then the Upload DMA unit will stall indefinitely awaiting the remainder. This is of particular note with 32bpp and 24bpp (packed) formats.

After loading the programs and setting the **PixelMode** and **FBMode** tags appropriately, the following tags need to be sent to complete the setup:

Tag	Requirements
FBBuffer*N*	Set *Width, PixelBytePitch, PixelSize, SubFieldStartByte* & *SubFieldStartCount* as appropriate for source. Set *ReadEnable* bit (or specify with **FBBufferReadEnables** tag).
FBBaseAddr*N*	Set to buffer base address.
FBBufferEnables	Enable buffer *N* only.
HostOutMode	Set *OutputUploadTag, OutputSyncTag* and *OutputUploadDMATags* bits.

7.4.2 Upload operation

The Upload DMA Unit supports only linear uploads, and therefore it is necessary to split a rectangular upload into a series of scanline uploads, each performed using a **DrawRectangle2D** of a single pixel in height.

For each scanline, the Upload DMA Unit is first prepared to receive the pixel data with the **UploadDMA** tag. The scanline is then rasterised with a **RectanglePosition/ DrawRectangle2D** pair. Finally a **Sync** tag is sent with a unique ID; receipt of this ID in the appropriate **SyncID** register will indicate the completion of the scanline upload to host memory (if desired, this final operation can be performed just once, after rasterising all the scanlines that comprise the upload rectangle).

The **UploadDMA** tag should be assembled with the following settings, with only the *Addr* field requiring alteration for successive scanlines:

Field	Requirements
ByteSwap	Set according to buffer and host memory layouts: 0=ABCD (no swap), 1=BADC, 2=CDAB, 3=DCBA .
PixelSize	Must match size of pixel data produced by Pixel Unit program: 0=8-bit, 1=16-bit, 2=24-bit, 3=32-bit.
Protocol	Selects bus protocol: 0=PCI, 1=AGP.
Enable	Set to 1.
Count	Upload count, in pixels - 1. Must agree with width in **DrawRectangle2D**.
Addr	Upload address, in bus address space. Performance will be optimised if this is aligned to a pixel boundary.

7.4.3 Monochrome uploads

Uploads to a monochrome bitmap can be efficiently achieved through the use of the texture subsystem. All units except the texture subsystem are configured as if for a native 8bpp upload, and programs in the Texture Coordinate Unit and Shading Unit assemble each byte from 8 horizontally consecutive source pixels.

A 2D texture is mapped onto the source rectangle, and for each scanline, a **RectanglePosition/DrawRectangle2D** pair is sent, with a horizontal width 1/8th of the true width of the rectangle (ie the number of bytes of data that will be uploaded). The x- and y-coordinates specify the offset of the first pixel to be uploaded from the top-left of the source rectangle (which must be tile-aligned). The program in the Texture Coordinate Unit uses these coordinates to access the appropriate 8 horizontally consecutive pixels in the

texture map which are passed to the Shading Unit and assembled into a single byte which is passed onto the Pixel Unit, and from there onto the Host Out Unit.

Since the smallest unit of data that can be uploaded is a byte, uploads must be performed as multiples of 8 pixels and so the host must deal with clipping of unwanted pixels at the left and right edges of each scanline.

7.4.3.1 Programs for monochrome uploads

At the heart of the method is the Texture Coordinate Unit program. The example program below causes bytes to be assembled with the rightmost pixel in each 8-pixel run placed in the least significant bit, and the leftmost pixel in the most significant bit. The Shading Unit programs will in this case be most efficient if the rightmost pixel is handled first, hence this program selects pixels in “reverse” order. Note also the optimised pairing of ALU and sequencer instructions:

```
Define(ShadeLoad, 1)
Define(DisableFeedback, 0)
Define(UpTex, N)        // using texture N

// Assumes Global0[0]=8, Global1[0]=7, Global0[1]=1/rectheight, Global1[1]=1/rectwidth

Program(TC_TranslateTo1bpp, 0)
    W[0] = MAdd(One, X, Zero, Zero);                // calculate x & y coordinates
    W[2] = IntToFloat( PStart[0] );
    W[3] = IntToFloat( PStart[1] );
    W[4] = MAdd(One, Global1[1], Zero, Zero);       // A[4]=X wrapping value
    W[0] = MAdd(A[0], Global0[0], One, B[2]);       // x=pstart(0)+8*X
    W[1] = MAdd(One, B[3], One, Y);                 // y=pstart(1)+Y
                                                    // y=pstart(1)+Y
    W[0] = MAdd(One, Global1[0], One, B[0]);        // x=pstart(0)+8*X+7

    C[0] = Wrap( A[0], B[4] );      // send horizontal index within texture map
                                    // convert to texture coords for 1st pixel
    C[1] = Wrap( A[1], Global0[1] );                // send vertical index within
    texture map
    Command(FilterTexture, UpTex, 0, ShadeLoad, DisableFeedback, First)   // send to first
    W[0] = MSub(A[0], One, One, One);       // & decrement X coord

    LoadCounter(0, 6)               // set up loop for middle six pixels
                                    // set up loop for the middle six pixels
                                    //
    C[0] = Wrap(A[0], B[4]);        // send horizontal index for 2nd pixel
                                    // & convert to texture coords for 2nd pixel loop:
    Command(FilterTexture, UpTex, 0, ShadeLoad, DisableFeedback, Middle)

                                                                    // send to middle
    W[0] = MSub(A[0], One, One, One);   // & decrement X coord
    C[0] = Wrap(A[0], B[4])             // send horizontal index for 3rd-8th pixels
                                        // convert to texture coords for 3rd-8th pixels
```

```
    DJNZ(0, loop);                  // & loop back
    Command(FilterTexture, UpTex, 0, ShadeLoad, DisableFeedback, Last)
                                    // send to last
    Fract( One)  Done;
```

As can be seen, this program initiates three different programs in the Shading Unit; the first handles initialisation of the upload byte with the first pixel, the middle (for the middle 6 pixels) incorporates each successive pixel into the upload byte, and the last adds in the final pixel and forwards the completed byte onto the Pixel Unit. Suitable programs for a 24bpp source are as follows:

```
Define (Scratch, 0)
Define (Bitmap, 1)
Define (Bitmask, 2)

// Assumes Global[0..2] loaded with 24bpp foreground colour for mono translation

Program(SU_Translate24bppTo1bpp_First, 0x00)
    W[Bitmask] = PassA( Const[128]_Z);      // set Bitmask=bit 7 (msb)      // initialise bitmask
    Flag =    Sub( T[0]_Z, Global[0]_Z, ==);  // test pixel against foreground colour
    Flag &= Sub( T[1]_Z, Global[1]_Z, ==);
    Flag &= Sub( T[2]_Z, Global[2]_Z, ==);
    W[Bitmap] = SelectA( A[Bitmask], Const[0]_Z);      // set initial pixel according to test
result
    W[Bitmask] = Add( A[Bitmask], B[Bitmask])          // Bitmask=bit 8 for middle/last
programs
    Done;                                              // because we'll shift right when adding
it in

Program(SU_Translate24bppTo1bpp_Middle, 0x08)
    Flag = Sub( T[0]_Z, Global[0]_Z, ==);              // test pixel against foreground colour
    Flag &= Sub( T[1]_Z, Global[1]_Z, ==);
    Flag &= Sub( T[2]_Z, Global[2]_Z, ==);
    W[Scratch] = SelectA( A[Bitmask], Const[0]_Z);     // select mono pixel value
    W[Bitmap] = Add( A[Bitmap], B[Scratch]) /2         // add in value and shift bitmask right
    Done;

Program(SU_Translate24bppTo1bpp_Last, 0x10)
    Flag =    Sub( T[0]_Z, Global[0]_Z, ==);           // test pixel against foreground colour
    Flag &= Sub( T[1]_Z, Global[1]_Z, ==);
    Flag &= Sub( T[2]_Z, Global[2]_Z, ==);
    W[Scratch] = SelectA( A[Bitmask], Const[0]_Z);     // select mono pixel value
    C[0] =    Add( A[Bitmap], B[Scratch]) /2   // add in value, shift bitmask right & send on
    Done;
```

The Pixel Address Unit program used is identical to the colour upload program (its only purpose being to initiate the Pixel Unit program, since this does not require access to data in the framebuffer).

```
Program( pixelAddressUploadProg, 0 )
SendDestAddrAndTile( buf0, puReg0, Only);
```

The Pixel Unit program simply passes on the single byte of data received from the Shading Unit:

```
Program( pixelUploadMono8pixels, 0 )
C[ 0.HO ] = PassB( F[0] )  Done;
```

7.4.3.2 Monochrome upload setup

After loading the programs and setting the **TextureCoordMode, ShadeMode, PixelMode** and **FBMode** tags appropriately (note that the *PlaneOriginAtZero* bit should be set in **TextureCoordMode**), the following tags need to be sent to complete the setup:

Tag	Requirements
FBBuffer*N*	Set *Width, PixelBytePitch, PixelSize, SubFieldStartByte* & *SubFieldStartCount* as appropriate for source. Set *ReadEnable* bit (or specify with **FBBufferReadEnables** tag).
FBBaseAddr*N*	Set to buffer base address.
FBBufferEnables	Enable buffer *N* only.
CacheControl	Set *InvalidateTexturePrimaryCache* and *InvalidateTextureSecondaryCache* bits.
HostOutMode	Set *OutputUploadTag, OutputSyncTag* and *OutputUploadDMATags* bits.
TextureGlobal0	Set to 8 (constant for example TCU program).
TextureGlobal1	Set to 7 (constant for example TCU program).
TextureGlobal2	Set to 1/rectheight (used by example TCU program).
TextureGlobal3	Set to 1/rectwidth (used by example TCU program).
ShadeGlobal0	Set to foreground colour (used by example SU program).
TextureIndexMode*N*	Set *MapType* to 1 (2D), *Width* to log2(rectwidth) and *Height* to log2(rectheight).
TextureBaseAddress*N*	Set to source rectangle tile base address
TextureAddressMode*N*	Set *MapType* to 1 (2D), *Format* to 10 (8888) and *Pitch* according to source pixel size: 0=8bpp, 1=16bpp, 2=24bpp, 3=32bpp. The combined *Width/Height/Depth* fields should be set to the source buffer width (in tiles). Set *PowerOfTwoTexture* to 0.
TexturePlaneStart0	Set low 32 bits to x-offset (in pixels) of left edge into source rectangle. Set high 32 bits to y-offset (in pixels) of top edge into source rectangle. These values are used in the example TCU program.

The values *rectheight* and *rectwidth* must be powers of two, and should be set to the lowest power of two large enough to encompass the source rectangle.

7.4.3.3 Monochrome upload operation

After the additional setup, the upload procedure for monochrome uploads is identical to that used for colour uploads as described in paragraph **7.4.2**, except that the upload count and rectangle width should be 1/8th of the true width, and the *PixelSize* field specified in the **UploadDMA** command should be set to 0 (8bpp).

8

Rendering

8.1 Program-to-program parameter consistency

8.2 Selecting the primitive type for the vertex stream target (triangles, polymode, 2D rectangles/clears)

8.3 Vertex Processing

This section looks at how to implement a standard geometry transformation and lighting pipeline in the Vertex Shading Unit (section 4.3.2). It is written against the OpenGL pipeline, but is equally applicable to other APIs.

For a good general discussion of the transformations involved in a geometry pipeline, see the *OpenGL Programming Guide,* chapter 3, “Viewing”. A useful analogy is setting up a camera to take a photograph. The steps are:

3. Positioning: point the camera at a scene, that is, at a viewable volume in the world to be modelled This is the ***Viewing Transformation***.
4. Composing: Arrange the scene by moving the components. This is the ***Modelling Transformation***
5. Choose a lens or Zoom setting. This is the ***Projection Transformation***.
6. Scale up or down to meet the size requirements of the display system – ***Viewport Transformation***.

The following code samples implement partial calculations, which may be pieced together based on state variables to implement a complete pipeline.

8.3.1 Transformation

There are four basic transformations, not all of which are always needed. They involve transforming the position by:

1. the ModelView matrix to get the eye vertex and
2. the ModelViewProjection matrix to get the projected vertex.

Both of these are 4x4 matrices so the code to do this (assuming the matrices are transposed) is:

```
reg[eyeVertex+] = Mul4 (in[pos_x], coeff[MVMat0+])
reg[eyeVertex+] = MAdd4 (in[pos_y], coeff[MVMat4+],
reg[eyeVertex+])
reg[eyeVertex+] = MAdd4 (in[pos_z], coeff[MVMat8+],
reg[eyeVertex+])
reg[eyeVertex+] = MAdd4 (in[pos_w], coeff[MVMat12+],
reg[eyeVertex+])
```

The 3-space eye vertex obtained in the routine above is needed for local lighting. The homogeneous eye vertex is also needed for the user clip planes so this is derived by:

```
reg[oneOverEyeW] = Recip (reg[eyeVertex_w])
reg[eyeVertex3+] = Mul3 (reg[eyeVertex+],
reg[oneOverEyeW])
```

3. Similarly the projected vertex is the input position multiplied by ModelViewProjection matrix. The 1/w value can be found from the projected vertex[2] by:

```
reg[oneOverW] = HRecip (reg[projVertex_w])
```

Transforming the normal by the Normal matrix. This is done the by code:

```
reg[normal+) = Mul3 (in[normal_x], coeff[NMat0+])
reg[normal+) = MAdd3 (in[normal_y], coeff[NMat3+],
reg[normal+])
reg[normal+) = MAdd3 (in[normal_z], coeff[NMat6+],
reg[normal+])
```

If the resulting normal needs to be normalized then the code to do this is:

```
reg[magSquared] = Dot3 (reg[normal+], reg[normal+])
reg[invMag] = InvSqrt (reg[magSquared])
reg[normal+] = Mul3 (reg[normal+], reg[invMag])
```

[2] See section @@@@ for a discussion of why ***HRecip*** and not ***Recip*** is used in this situation.

The ***InvSqrt*** instruction returns the value from the seed table. This can be refined with one iteration of the Newton Raphson formula (this takes the approximation result from 10 bits to 20 bits):

r = (3.0 - r * r * x) * r * 0.5;

A more accurate 20bit normalisation based on this approach is:

reg[magSquared] = Dot3 (reg[normal+], reg[normal+])
reg[r] = InvSqrt (reg[magSquared])
reg[rr] = Mul (reg[r], reg[r])
reg[r5] = Mul (reg[r], coeff[ConstandHalf])
reg[rrx] = Mul (reg[rr], reg[magSquared])
reg[rrx] = Add (coeff[ConstantThree], -reg[rrx])
reg[invMag] = Mul (reg[rrx], reg[r5])
reg[normal+] = Mul3 (reg[normal+], reg[invMag])

8.3.2 Texture Operation

The texture operation consists of selecting where each input coordinate is coming from and if it is from a TexGen operation actually doing the texture generation function. The final 4 component texture is then multiplied by the texture matrix as already covered. In these examples the texture value will be multiplied through by 1/w.

The four TexGen operations in OpenGL are:

- Using the corresponding input coordinate.
  reg[texture_x] = Move (in[texture_x])
- Generating a value using ObjectLinear. TexGen1 holds the coefficients. *reg[texture_y] = Dot4 (in[p*
- Generating a value using EyeLinear. TexGen2 holds the coefficients.
  reg[texture_z] = Dot4 (reg[eyeVertex+], coeff[TexGen2+], reg[texture_z])
- Sphere Map. The TexGen operation for SphereMap is given by:
  $\mathbf{r} = \mathbf{u} - 2(\mathbf{n} \cdot \mathbf{n} \cdot \mathbf{u})$
  $\mu = 2\sqrt{(r^2x + r^2y + (r_z + 1)^2)}$
  $\sigma = r_x / \mu + 1/2$
  $\tau = r_y / \mu + 1/2$

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 into eye space
• is a dot product

The code to implement this is:

```
// Normalise the eye vertex.
// See earlier code, result left in u.
// normal dot eye
reg[NdotU] = Dot3 (reg[normal+], reg[u+]);
                                        // r = u - (nn * (NdotU * 2.0));
reg[NdotU] = Mul (reg[NdotU], coeff[ConstantTwo])
reg[nn+] = Mul3 (reg[normal+], reg[NdotU])
reg[r] = Add3 (reg[u+], -r[nn+   // m = r.x * r.x + r.y * r.y + (r.z + 1.0) * (r.z + 1.0);
reg[r_z] = Add (reg[r_z], coeff[ConstantOne])
reg[m] = Dot3 (reg[r], reg[r])
//recipM = InverseSquareRoot (m) * 0.5;
reg[m] = InvSqrt (reg[m])                  // seed
reg[mm] = Mul (reg[m], reg[m])             // Newton Raphson approx
reg[m5] = Mul (reg[m], coeff[ConstandHalf])
reg[mmx] = Mul (reg[mm], reg[mm])
reg[rrx] = Add (coeff[ConstantThree], -reg[mmx])
reg[m] = Mul (reg[mmx], reg[m5])
reg[m] = Mul (reg[m], coeff[ConstantHalf])
reg[texture+] = MAdd2 (reg[r+], reg[m],
coeff[ConstantHalf])
```

In OpenGL all four TexGen operations can be mixed in any combination (sphere maps are only legal for s and t) so combinations of the above sets of 4 instructions are usually collected together. If no texture transformation is needed the results can be multiplied by 1/w and written to the output register.

The texture can be optionally transformed, but must be divided through by w. The reciprocal of w is available as it is needed for the perspective division just prior to veiwport mapping. If the texture is to be used as a projective texture then it will have a Q that is not unity and S/Q, T/Q and maybe R/Q will also be needed.

8.3.3 Fog

The fog calculation takes the vertex z value in eye coordinates and applies one of the selected equations:

$f = e^{-d*z}$ Exponential

$f = e^{-(d*z)2}$ Exponential squared

$f = (e - z)\ \lambda$ where $\lambda = 1/(\varepsilon - s)$ Linear

where:
d is the fog density
s is the fog start value

ε is the fog end value.

The values d , λ, and ε are held in the FogDensity, FogScale and FogEnd coefficient registers respectively.

The sections below describe the fog test implementations.

8.3.3.1 FogExp

p = fogDensity * eyeZ;

fog = Exponent (p);

```
reg[fog] = Mul (coeff[fogDensity], |reg[eyeVertex_z])      // p
                                                           // Exponential approximation.
reg[fog] = Mul (^reg[fog], coeff[ConstantLogOfE])
// ConstantLogofE is ~1.442695
reg[fog] = ALog (-reg[fog])
```

8.3.3.2 FogExp2

p = (fogDensity * fogDensity) * (eyeZ * eyeZ);

fog = Exponent (p);

```
reg[fog] = Mul (coeff[fogDensity], |reg[eyeVertex_z]) // p
// Exponential approximation.
reg[fog] = Mul (^reg[fog], ^reg[fog])
```

```
reg[fog] = Mul (reg[fog], coeff[ConstantLogOfE])
// ConstantLogofE is ~1.442695
reg[fog] = ALog (-reg[fog])
```

8.3.3.3 FogLinear

eyeZ = Abs (eyeVertex.z).

fog = (fogEnd - eyeZ) * fogScale

```
// We cannot do an abs and negate at the same time so juggle
// equation around.

reg[fog] = Add (-coeff[fogEnd], |reg[eyeVertex_z])
reg[fog] = Mul (-reg[fog], coeff[fogScale])
```

8.3.4 Lighting

8.3.4.1 Generalized light pipeline:

A generalised lighting pipeline for light i, is:

```
// eyeVertex, VP are Vec3.
    if (localLight | localViewer | spotLight | attenuation)
    {
    // Normalised vector from light to eyeVertex and light position
    // in 3-space.
    VP = eyeVertex - light[i].position; // Light position in 3D coords
    magVPSquared = Dot (VP, VP);
    invMagVP = InverseSquareRoot (magVPSquared)
    VP = VP * invMagVP;
    }
else
    {
    VP = light[i].position;
    }

if (spotlight)
    {
    // PDot clamps to zero if negative.
    spotDot = PDot3 (-VP, light[i].spotlightDirection);
    if (spotDot < light[i].cosSpotlightCutoffAngle)
    spotAttenuation = 0.0; // light adds no contribution
    else
    spotAttenuation = Power (spotDot,
    light[i].spotlightExponent);
    }
else
    spotAttenuation = 1.0;
```

```
if (attenuation)
    {
    // Attenuation factors stored as 1 over factor.
    distAttenuation = light[i].constantAttenuation
    light[i].linearAttenuation * invMagVP +
    light[i].quadraticAttenuation * invMagVP * invMagVP;
    }
else
    distanceAttenuation = 1.0;
    attenuation = spotAttenuation * distAttenuation;

if (localViewer)
    {
    halfVector = -eyeVertex;
    halfVector = Norm (halfVector);
    halfVector += VP;
    }
else if (localLight)
    halfVector = VP + Vec3 (0, 0, 1);
else
    halfVector = light[i].halfVector;

if (localLight)
    halfVector = NormQuick (halfVector);
    else if (localViewer)
    halfVector = Norm (halfVector);

    normalDotVP = PDot3 (normal, VP);
    normalDotHalfVector = PDot3 (normal, halfVector);

if (normalDotVP == 0.0)
    powerFactor = 0.0;
else
    powFactor = Power (normalDotHalfVector, material.specularExponent);
    ambientLight += light[i].ambientIntensity * attenuation;
    diffuseLight += light[i].diffuseIntensity * normalDotVP *
    attenuation;
    specularLight += light[i].specularIntensity * powFactor *
    attenuation;
    }
```

This will calculate the lights ambient, diffuse and specular contributions and accumulate them. We will now go through the instruction sequences three types of lights: directional, local lights, and the full lighting model with everything turned on.

These two functions crop enough so we don't want to keep duplicating the same code so we will define them here as 'macros'.

```
V = Norm (V)
reg[magVSquared] = Dot3 (reg[V+], reg[V+])
reg[r] = InvSqrt (reg[magVPSquared]) // seed
reg[rr] = Mul (reg[r], reg[r]) // Newton Raphson approx
reg[r5] = Mul (reg[r], coeff[ConstantHalf])
reg[rrx] = Mul (reg[rr], reg[magSquared])
reg[rrx] = Add (coeff[ConstantThree], -reg[rrx])
reg[invMagV] = Mul (reg[rrx], reg[r5])
reg[V+] = Mul3 (reg[VP+], reg[invMagV])

p = Power (a, b)
reg[p] = Log (^reg[a]) // log 0 returns 0
reg[p] = Mul (reg[p], reg[b])
reg[p] = ALog (reg[p])
```

8.3.4.2 Multiple Directional Lights

The minimum program to evaluate the Phong lighting model is given below. The ambient contribution of the scene and the light together with the emissive contribution of the material are lumped together as described in the Material section later on.

```
reg[normalDotVP] = Dot3 (reg[normal+], coeff[lightPosition+])
reg[normalDotHalfVector] = Dot3 (reg[normal+], coeff[lightHalfVector+])
powFactor = Power (reg[normalDotHalfVector], coeff[specularExponent])
reg[t1] = SGE (reg[normalDotVP], coeff[Zero])
reg[specFactor] = Mul (reg[powFactor], reg[t1])
// For the first light the diffuseLight and specularLight // would be written to i.e. the MAdd3
        instructions would // be Mul3.
reg[diffuseLight+] = MAdd3(coeff[lightDiffuseIntensity+], ^reg[normalDotVP],
        reg[diffuseLight+])
reg[specularLight+] =
MAdd3(coeff[lightSpecularIntensity+],

reg[specFactor], reg[specularLight+])
```

8.3.4.3 Multiple Local Lights

This type of lights are local lights with no attenuation and an infinite viewer. The eyeVector has been calculated and converted from its homogeneous representation into 3-space. The ambient contribution of the scene and the light together with the emissive contribution of the material are lumped together as described in the Material section later on.

```
// Normalised vector from the light to the eye
reg[VP+] = Add3 (reg[eyeVertex3+], -coeff[lightPosition+])
reg[VP+] = Norm (VP+)
// short hand, also leaves invMagVP available
// Half vector calculation
```

```
reg[halfVector+] = Add3 (reg[VP+], coeff[constVec0_0_1+])
// We maybe able to avoid the iterations to refine the
// result as the maximum magnitude can only be 2.
// This has not been assumed.
reg[halfVector] = Norm (halfVector)
// reg[normalDotVP] = Dot3 (reg[normal+], reg[VP+])
// This dot product cannot be done directly as it
// requires 3 reads from the scratch registers so this is // broken into several steps.
reg[temp+] = Mul3 (reg[normal+], reg[VP+])
reg[normalDotVP] = Add (reg[temp_x], reg[temp_y])
reg[normalDotVP] = Add (reg[normalDotVP], reg[temp_z])
// reg[normalDotHalfVector] = Dot3 (reg[normal+], reg[halfVector+])
// This dot product cannot be done direclty as it
// requires 3 reads from the scratch registers so this is // broken into several steps.
reg[temp+] = Mul3 (reg[normal+], reg[halfVector+])
reg[normalDotHalfVector] = Add (reg[temp_x], reg[temp_y])
reg[normalDotHalfVector] = Add (reg[normalDotHalfVector], reg[temp_z])
powFactor = Power (reg[normalDotHalfVector], coeff[specularExponent])
reg[t1] = SGE (reg[normalDotVP], coeff[Zero])
reg[specFactor] = Mul (reg[powFactor], reg[t1])
// For the first light the diffuseLight and specularLight // would be written to i.e. the MAdd3
        instructions would // be Mul3.
reg[diffuseLight+] = MAdd3
(coeff[lightDiffuseIntensity+],^reg[normalDotVP], reg[diffuseLight+])
reg[specularLight+] = MAdd3
(coeff[lightSpecularIntensity+], reg[specFactor], reg[specularLight+])
```

8.3.4.4 Full Function Lights

These lights have local viewer, local light, spot lights and attenuation. The eyeVector has been calculated and converted from its homogeneous representation into 3-space. The ambient contribution of the scene and the emissive contribution of the material are lumped together as described in the Material section later on

```
// Normalised vector from the light to the eye vertex
reg[VP+] = Add3 (reg[eyeVertex3+], -coeff[lightPosition+])
reg[VP+] = Norm (VP+)
// short hand, also leaves invMagVP available
// Spotlight
reg[spotDot] = Dot3 (-reg[VP+], coeff[spotlightDirection+])
reg[spotAtten] = Power (spotDot, coeff[spotlightExponent])
reg[spotOn] = SLE (reg[spotDot],
coeff[cosSpotlightCutoffAngle])
reg[spotAtten] = Mul (reg[spotOn], reg[spotAtten])
// Attenuation
reg[invMagVP2] = Mul (reg[invMagVP], reg[invMagVP])
reg[distAtten] = Move (coeff[constantAttenuation])
reg[distAtten] = MAdd (coeff[linearAttenuation],
```

```
reg[invMagVP], reg[distAtten])
reg[distAtten] = MAdd (coeff[quadraticAttenuation],
reg[invMagVP2], reg[distAtten])
// Combine both attenuation factors.
reg[attenuation] = Mul (reg[distAtten], reg[spotAtten]
// Half vector calculation
reg[halfVector+] = Add3 (reg[VP+], -reg[normalisedEyeVector])
reg[halfVector] = Norm (halfVector)
// reg[normalDotVP] = Dot3 (reg[normal+], reg[VP+])
// This dot product cannot be done direclty as it requires 3 // reads from the scratch registers so
        this is broken into
// several steps.
reg[temp+] = Mul3 (reg[normal+], reg[VP+])
reg[normalDotVP] = Add (reg[temp_x], reg[temp_y])
reg[normalDotVP] = Add (reg[normalDotVP], reg[temp_z])
// reg[normalDotHalfVector] = Dot3 (reg[normal+], reg[halfVector+])
// This dot product cannot be done direclty as it requires 3 reads from
// the scratch registers so this is broken into several steps.
reg[temp+] = Mul3 (reg[normal+], reg[halfVector+])
reg[normalDotVP] = Add (reg[temp_x], reg[temp_y])
reg[normalDotHalfVector] = Add (reg[normalDotHalfVector],
reg[temp_z])
powFactor = Power (reg[normalDotHalfVector],
coeff[specularExponent])
reg[t1] = SGE (reg[normalDotVP], coeff[Zero])
reg[specFactor] = Mul (reg[powFactor], reg[t1])
reg[specFactor] = Mul (reg[specFactor], reg[attenuation])
reg[diffFactor] = Mul (^reg[normalDotVP], reg[attenuation])
// For the first light the ambientLight diffuseLight and
// specularLight would be written to i.e. the MAdd3
// instructions would be Mul3.
reg[ambientLight+] = MAdd3 (coeff[lightAmbientIntensity+],
reg[attenuation], reg[ambientLight+])
reg[diffuseLight+] = MAdd3 (coeff[lightDiffuseIntensity+],
reg[diffFactor], reg[diffuseLight+])
reg[specularLight+] = MAdd3 (coeff[lightSpecularIntensity+],
reg[specFactor], reg[specularLight+]).
```

Two sided lighting would evaluate the lighting for both sides of the vertex (making use of the common expressions) and write the front colour to an even colour output register and the back colour to an odd colour output register. The Parameter Setup Unit (when suitably enabled) will select one colour out of the pair depending on the surface orientation.

The lighting presented above is all done in eye space as this is guaranteed to work for all types of lights. Many of the common lighting operations such are directional lights can be done in object space which avoids the need to transform the normal, saving computation time.

8.3.4.5 Material

Once the light's ambient, specular and diffuse contributions have been found they are combined with the material's colour as follows. OpenGL 1.2 introduced the idea of a secondary colour to drive a specular interpolator. This has not been covered here as it is only a very minor change.

The light and material properties are combined as shown in the following vector equations:

$$c = e_{cm} + a_{cm} * a_{cs} + \text{ambientLight} * a_{cm} + \text{diffuseLight} * d_{cm} + \text{specularLight} * s_{cm}$$

where:

e_{cm} is the emissive material colour (front or back)
a_{cm} is the ambient material colour (front or back)
a_{cs} is the ambient scene colour
a_{cm} is the ambient material colour (front or back)
d_{cm} is the diffuse material colour (front or back)
s_{cm} is the specular material colour (front or back)

reg[colour+] = Add3 (coeff[SceneAmbient+],
reg[ambientLight+])
reg[colour+] = Mul3 (reg[colour+], reg[colour+],
coeff[AmbientColour+]
reg[colour+] = Add3 (reg[colour+], reg[c],
coeff[EmissiveColour+]
reg[colour+] = Madd3 (reg[colour+], reg[diffuseLight+],
c[DiffuseColour+], reg[colour+]
out[colour+] = Madd3 reg[colour+], reg[specularLight+],
coeff[SpecularColour+], reg[colour+]
out[alpha] = Move (coeff[MaterialAlpha])

When there is no attenuation on any light it is possible to precompute the ambient and emissive contribution and store this in the coefficient memory. This will reduce computation time (10 cycles versus 16) (and also save the ambientLight accumulation in the lighting calculations):

$$\text{ambient} = e_{cm} + a_{cm} * a_{cs} + \text{ambientLight} * a_{cm}$$

When any of the lights have attenuation the savings are reduced to 13.

$$\text{ambient} = e_{cm} + \text{acm} * a_{cs}$$

8.3.5 User Clip Planes

The actual clipping against any user clipping planes is done in the Geometry Unit. All that is needed here is to generate the outcode vector. This is done as follows:

```
reg[userOutcode] = Move (coeff[ConstantZero])
// Repeat for each user clipping plane
reg[clipDistance] = Dot4 (reg[eyeVertex],
coeff[userClipPlane0])
reg[userOutcode] = ShiftSign (reg[userOutcode],
reg[clipDistance])
```

The final result is written to the x component of the parameter identified in the **VertexShadingMode** register as holding the SpecialParameter.

Note: It is possible to do user clipping in object space and so avoid the need to generate eye coordinates. The user clip planes in the Vertex Shader and Geometry Units must be defined in object coordinates.

The ShiftSign instruction does a left shift so the first clip plane tested will end up in bit position n - 1 of the outcode word where n is the number of clip planes processed in the program. It is recommended that the first clip plane tested (notionally UserClip 0) always ends up in bit position 5. This then allows UserClipMask bit 0 and UserClipPlane[0] (in the Geometry Unit) to correspond to the first clip plane tested in the program.

8.3.6 Projection and Viewport Mapping

The Culling and Geometry Units expect the vertex position data to be in the window coordinate system all ready to be used for rendering. In the rare case we need to clip against the viewing frustum the Geometry Unit will spend extra cycles to recover the information it needs from the window coordinates rather than force the clip coordinates to be maintained and passed down as part of the processed vertex data. The input vertex has been transformed by the ModelViewProjection matrix and the result is in the **projVertex** registers. The final processing is:

```
reg[oneOverW] = HRecip (reg[projVertex_w])
reg[ndc+] = Mul3 (reg[projVertex+], reg[oneOverW])
reg[dc] = Mul3 (coeff[viewPortScale+], reg[ncd+])
out[windowCoord+] = Add3 (coeff[viewPortOffset+],
reg[dc+])
out[windowCoord_w] = Move (reg[projVertex_w])
```

The OneOverW value is also used when writing texture coordinates out as these are also expected to be divided through by w.

Projection is usually done after clipping so never has to face a problem of what to do when w is close to zero. A homogeneous vertex with a w of zero is perfectly legal on a primitive prior to clipping and clipping can yield a valid primitive to draw. It is not possible for the Geometry Unit to recover the original clip coordinates if the homogeneous w is zero, hence it can not clip such a primitive correctly. The solution is to avoid doing the

projection and to mark this vertex as non projected so it can be forced to go through clipping and avoid the reverse projection in the Geometry Unit.

This is done using the ***HRecip*** instruction which will return 1.0 when the denominator is zero. Multiplying the homogeneous vertex and texture coordinates by one will not change them. The flagging of the vertex as non projected is handled with no further work here because the homogeneous w value already accompanies the window coordinates and the Cull and Geometry Units can test this and take the appropriate action if zero.

8.4 Shading (Gouraud, flat, modulate etc.)

Described from the perspective of Direct3D renderstate control there are three main approaches to shading supported by P10. In increasing order of complexity and resulting image flexibility, these are:

- Flat shading
- Gouraud shading
- Texture based per pixel shading techniques

These shading operations are performed in the Shading Unit (SU) using a combination of the input data available as introduced in Section 4.2.2, "Shading". These include diffuse and specular iterated vertex colours available from the plane equation registers, the eight multi-stage texture registers and a set of global registers that may be externally loaded. The specific register setup for the texturing pipelines and programming of the texture coordinate unit are discussed in Section 7.2 on texturing. Here we assume the textures are available in the input texture registers and concentrate on the program setup and processing of the shading unit.

Before looking at the programs required for various shading techniques, we will look at the main mechanisms for the configuration of the unit and the loading of programs.

The **ShadeMode** register is responsible for controlling the unit's operation and has the following field structure:

Field Name	Description
TileEnable	When set this bit enables a **Tile** command to start a program running.
TileAddrDefault	Holds the 7bit address of the program to run when a subtile is received (assuming it is enabled with the above field) when the input command fifo prog field is 0. This field also holds the address of the program to run when the Texture Coordinate Unit is disabled.
TileAddrFirst	Holds the address of the program to run when a subtile is received (assuming it is enabled) when the prog field is 1.
TileAddrMiddle	Holds the address of the program to run when a subtile is received (assuming it is enabled) when the prog field is 2.
TileAddrLast	Holds the address of the program to run when a subtile is received (assuming it is enabled) when the prog field is 3.

PlaneOriginAtZero	When set, this forces the plane equation origin to be at zero otherwise the plane origin is the coordinate of the first tile seen by this unit. This bit would normally be set when the Parameter Set Up Unit is not being used to set up the plane equations, such as for 2D operations.

TileEnable effectively enables or disables the unit for tile rasterisation. Assuming this bit is set, the following four 7bit fields hold the addresses for the position in the shading units program memory for up to four subprogram components that allow the shading unit to perform multipass processing. The inputs to the shading unit are from both the texturing pipe, via a command fifo from the texture filter unit, and directly from the parameter setup unit. This latter unit is the only interface when the texture coordinate unit (TCU) is disabled (see **TextureCoordMode** below). When the texture coordinate unit is in use its programs work in conjunction with those of the shading unit to allow advanced shading effects. It is the Command fifo writes from the TCU that contain the *prog* field mentioned above to indicate which of the First, Middle and Last program addresses to use. The First may be used to intialise local working registers in the SU, the Middle for multiple intermediate passes (e.g. accumulation) and the Last to perform any final processing (e.g. scaling of accumulated value).

The structure of the **TextureCoordMode** register and hence the tag used to setup texturing are described in detail in the section on texture coordinate generation. For now it is important to know that this register also contains a *TileEnable* field for unit enablement/disablement.

Programs are loaded into the shading unit one instruction at a time using the **ShadeProgramData** and **ShadeProgramAddr** registers. **ShadeProgramAddr** indicates the position in the shading unit program store where the next instruction, loaded with the **ShadeProgramData** register, should be placed. **ShadeProgramAddr** need only be set once for each set of consecutively stored instructions, with the address pointer incremented automatically on receipt of each instruction.

Data can be loaded for global use by the SIMD processors into 32 global byte registers. These are actually loaded in sets of four registers at a time using the **ShadeGlobal0..7** registers, where **ShadeGlobal0** will load global registers 0 to 3. These are packed into the accompanying data word with register 0 at the least significant byte and so on.

Program runs are initiated by either **Tile** commands from the rasteriser or a **RunShadingProg** command which can be used to kick start a program for program initiation, local register setup etc.

8.4.1 Flat Shading

Flat shading is the simplest form of shading leading to a facetted appearance, whereby each rendering primitive (e.g. triangle) is assigned a single colour. This colour is determined as a vertex operation in the transform and lighting subsection of the core with the flat shading constant colour taken from a particular assigned provoking vertex. The selection of flat shading or Gouraud shading is determined from the Direct3D API through the D3DRENDERSTATE_SHADEMODE render state, which may be D3DSHADE_FLAT or D3DSHADE_GOURAUD. To setup the P10 core for this state requires the setting of the *FlatShading* field of the **GeometryMode** register as below:

```
// Shading.
switch (pContext->RenderStates[D3DRENDERSTATE_SHADEMODE])
{
      case D3DSHADE_FLAT:
        pSoftP10->P10GeometryMode.bits.FlatShading = 1;
        break;
      default:
        pSoftP10->P10GeometryMode.bits.FlatShading = 0;
        break;
}
```

With this software copy later loaded to the core with:

```
SEND_P10_DATA(GeometryMode, pSoftP10->P10GeometryMode.word);
```

The *UseProvoking* mask field of the **ParameterSetupMode** register must also be set. This is effectively a mask that indicates which of the possible eight different colour parameters provided with a vertex should actually be treated as flat shaded when it is indicated that the primitive should be treated as such. For example, the simplest setting here would be 0xff, causing all colour parameters to be flat shaded. However, in reality, some colours like specular colour and fog etc. should not be.

Setting the *UseProvoking* mask will cause the core to internally set the provoking vertex. However, the choice of which vertex to use will differ depending on the API in use. OpenGL and D3D use different rules for the selection of the provoking vertex used to provide the single flat shading colour. For example for a triangle strip OpenGL will use the last vertex to arrive e.g. (v0,v1,**v2**), (v1,v2,**v0**), (v2,v0,**v1**), while D3D will use the first (**v0**,v1,v2), (**v1**,v2,v0), (**v2**,v0,v1). P10 is told which rule to use via the **Begin** command tag, used to indicate the start of primitive rendering. This has the format:

Bit-fields	Description	
0..3	The lower 4 bits sets up the primitive type to process on receiving each new vertex. It has the following values:	0 Null 1 Points 2 Lines 3 LineLoop 4 LineStrip 5 Triangles 6 TriangleStrip 7 TriangleFan 8 Quads 9 QuadStrip 10 Polygon 11 Grid
4..7	Bits 4…7 holds the grid width in vertices.	
8	ProvokingVertexRule, when set causes the D3D provoking vertex rules to be used.	

For example to render a triangle strip:

SEND_P10_DATA(Begin, P10_TRIANGLESTRIP | P10_PROVOKING_RULE_D3D);

Where P10_TriangleStrip = 6 and P10_PROVOKING_RULE_D3D = (1 << 8).

When setup correctly for flat shading the Plane equation registers will be automatically loaded with the constant flat shade colour and an example of a simple shading unit program would then be as follows:

```
Define(KdBlue, 0)
Define(KdGreen, 1)
Define(KdRed, 2)

C[0]= PassA(Plane[KdBlue]);
C[1]= PassA(Plane[KdGreen]);
C[2]= PassA(Plane[KdRed]) Done;
```

As shown here, the register addresses can be indicated by defined constants for readability

8.4.2 Gouraud Shading (Diffuse and Specular)

With Gouraud shading, the colours provided at each vertex (provided either by the application or calculated by the hardware lighting calculations) are linearly interpolated across the primitive to provide the representation of a smooth surface. This is internally configured as the default alternative to flat shading so that if the **GeometryMode** register is not setup for flat shading then the plane equation evaluators will perform the iteration for each SIMD element. A simple shading unit program that combines both diffuse and specular components is then as follows:

```
...
Define(KsBlue, 0)
Define(KsGreen, 1)
Define(KsRed, 2)
W[0]= PassA(Plane[KdBlue]);                    // Load up the diffuse
W[1]= PassA(Plane[KdGreen]);
W[2]= PassA(Plane[KdRed]);
W[0] = AddS (A[0], Plane[KsBlue]);             // Add the specular
W[1] = AddS (A[1], Plane[KsGreen]);
W[2] = AddS (A[2], Plane[KsRed]);
C[0] = Saturate (A[0]);
C[1] = Saturate (A[1]);
C[2] = Saturate (A[2]);
```

As explained in Section 4.4.4.7, there is a need to saturate the output values to the range 0 to 1 from the internal –8 to +8 format of the unit.

8.4.3 Texture Based Shading

Access to the filtered texture samples generated by the texturing pipe is performed simply by use of the texture registers accessible to each SIMD element. Some example shading unit programs using this information are as follows:

```
// (1)Modulate the texture with the diffuse colour
C[0] = MultS (T[0], Plane[0]);
C[1] = MultS (T[1], Plane[1]);
C[2] = MultS (T[2], Plane[2]);
C[3] = MultS (T[3], Plane[3]);
```

An example of the code currently generated by the driver is as follows.

```
PassA(A[0]) Call(FlagMode,0x00);
W[27]=  MultS(T[0.C],Plane[0.C]) ;
W[0.C]=       PassA(A[27]) Return;
Flag=         PassA(A[0]) Call(FlagMode,0x00);
W[27]=  MultS(T[0.C],Plane[0.C]) ;
W[0.C]=       PassA(A[27]) Return;
Flag&=  PassA(A[0]) Call(FlagMode,0x00);
W[27]=  MultS(T[0.C],Plane[0.C]) ;
W[0.C]=       PassA(A[27]) Return;
W[3]=          PassA(Plane[3]) ;
C[0]=         PassA(A[0]) ;
C[1]=         PassA(A[1]) ;
C[2]=         PassA(A[2]) ;
C[3]=         PassA(A[3]) Done;
```

This version shows the use of subroutining with component relative addressing. Here the .C extension ensures that the register address will be updated to the required component (0,1,2 or 3) based on the status of the flag mode in the instruction calling the subroutine. e.g. Flag&= offsets the register address by 2. The use of such subroutines has the advantage of reducing the overall program size that must be stored..

```
// (2) Blend the current stage texture with the vertex interpolated diffuse alpha value.
PassA(A[0]) Call(FlagMode,0x02);
W[24]   =   PassA(Plane[3]) ;
W[26]   =   PassA(Plane[0.C]) ;
W[27]   =   Sub(T[0.C],Plane[0.C],0) Call(0x00);
W[27]   =   MultS(A[27],B[24]) ;
W[27]   =   Add(A[27],B[26]) Return;
W[0.C]  =   PassA(A[27]) Return;
Flag    =   PassA(A[0]) Call(FlagMode,0x02);
W[24]   =   PassA(Plane[3]) ;
W[26]   =   PassA(Plane[0.C]) ;
W[27]   =   Sub(T[0.C],Plane[0.C],0) Call(0x00);
W[27]   =   MultS(A[27],B[24]) ;
W[27]   =   Add(A[27],B[26]) Return;
W[0.C]  =   PassA(A[27]) Return;
Flag&   =   PassA(A[0]) Call(FlagMode,0x02);
```

```
W[24]   =   PassA(Plane[3]) ;
W[26]   =   PassA(Plane[0.C]) ;
W[27]   =   Sub(T[0.C],Plane[0.C],0) Call(0x00);
W[27]   =   MultS(A[27],B[24]) ;
W[27]   =   Add(A[27],B[26]) Return;
W[0.C]  =   PassA(A[27]) Return;
W[3]    =   PassA(Plane[3]) ;
C[0]    =   PassA(A[0]) ;
C[1]    =   PassA(A[1]) ;
C[2]    =   PassA(A[2]) ;
C[3]    =   PassA(A[3]) Done;
```

8.5 Texturing

The download and setup of textures for use in rendering is described in Section 7.2, Texture Maps. This section considers texturing with regard to multi-texture rendering and provides a discussion of the methods by which P10 is programmed to access its texturing functionality.

8.5.1 Texture co-ordinate generation (1D, 2D, 3D; sharing the work on multiple co-ordinate sets)

The generation of the coordinates used to sample the textures in each of P10's texturing pipes is the responsibility of the programmable Texture Coordinate Unit (TCU) as introduced in Texture Coordinate Unit (s.4.3.3). Up to eight textures can be processed per fragment in a single pass.

The operation of the unit is controlled by the **TextureCoordMode** register, loaded using the tag of the same name. This has the format:

Field Name	Description	
FeedbackSource	This bit determines where the feedback data will come from.	0 = Download Image data 1 = Texture pipe
TileEnable	When set this bit enables a Tile comand to start a program running.	
TileAddr	Holds the 7bit address of the program to run when a Tile command is received (assuming it is enabled).	
PlaneOriginAtZero	When set, this bit forces the plane equation origin to be at zero otherwise the plane origin is the coordinate of the first tile seen by this unit.	

The *TileEnable* field effectively enables/disables the use of the texturing pipes in the P10 core. If not set, the **tile** commands received from the rasteriser will not enable texture coordinate generation so no texturing takes place. The **tile** commands instead bypass the associated texturing units and pass directly to the shading unit for possible flat or Gouraud shading. The **TextureCoordMode** command is therefore also passed onto the shading unit, which inspects it to see if any texturing results will be available.

Up to 128 instructions can be stored in the texture coordinate unit program store. Several programs can be stored to minimise reloading. The *TileAddr* field indicates the start address of the program to be run on receipt of a **tile** command.

The *FeedbackSource* field controls the operation of the feedback register port. For rendering this is generally set to 1 to allow feedback from the texture pipe for operations such as bump mapping as described in a later subsection. However, it can alternatively be used to download data directly via the **UserFragData0..63** registers. These allow data to be loaded to the SIMD processors responsible for specific tile elements (pixels).

Programs are loaded into the shading unit one instruction at a time using the **TextureProgramData** and **TextureProgramAddr** registers. **TextureProgramAddr** indicates the position within the shading unit program store where the next instruction, loaded with the **TextureProgramData** register, should be placed. **TextureProgramAddr** need only be set once for each set of consecutively stored instructions, with the address pointer incremented automatically on receipt of each instruction.

The **TexturePlaneScale0..7** register tags are used to load the corresponding registers that hold the dimensions, as a power of two, of each of the eight possible textures. These are used in the level of detail (LOD) calculations where it is necessary to know the scale of each coordinate axis (up to 4D) to estimate the texture minification ratio. These 16bit values are of the format:

- Bits 0…3 = S scale
- Bits 4…7 = T scale
- Bits 8…11 = R scale
- Bits 12…15 = Q scale

For example, for a 3D cubmap texture:

```
// Scale third texture coordinate by height ( as width == height because
// cubemaps must be square ( as well as pow2 ))
SEND_P10_DATA ( TexturePlaneScale0, (pTexture->logHeight << 8) |
    (pTexture->logHeight << 4) |
     pTexture->logWidth );
```

The texture coordinate unit possesses 32 floating point global registers that can be seen by all of the SIMD processing elements. The global registers are loaded individually, i.e. 32bits at a time using the **TextureGlobal0..32** tags, but internally they are read 64bits at a time to allow two global register values to be presented to the processor ALUs simultaneously. Which one of a pair of registers that is actually used is determined by the assembler syntax *Global0* or *Global1*. For example, the matrix elements for a bump mapping matrix can be loaded as follows…

```
SEND_P10_DATA( TextureGlobal0, AS_ULONG(fBumpMapMatrix[0]) );
SEND_P10_DATA( TextureGlobal1, AS_ULONG(fBumpMapMatrix[1]) )
SEND_P10_DATA( TextureGlobal2, AS_ULONG(fBumpMapMatrix[2]) );
SEND_P10_DATA( TextureGlobal3, AS_ULONG(fBumpMapMatrix[3]) );
```

(where AS_ULONG places the raw 32bit value of the floating point word in the data field) and then accessed in a program with:

```
Define(mat00, 0)
Define(mat10, 0)
```

```
Define(mat01, 1)
Define(mat11, 1)
// Transform the perturbation values using the bump transform matrix.
W[dxt] = MAdd(A[dx], Global0[@mat00], B[dy], Global1[@mat10]);
W[dyt] = MAdd(A[dx], Global0[@mat01], B[dy], Global1[@mat11]);
```

TCU programs are initiated in response to **tile** commands from the rasteriser. However, it is also possible to kick start a program run manually using the **RunTextureProg** command tag. The least significant 7 bits of the data word accompanying this tag contains the starting address of the program to be run. Note that this will only run if bit 31 of the data is set.

Two important registers that must be set to configure texturing are the **TextureIndexMode0..7** and **TextureIndexMipControl0..7** registers. These control how the coordinates generated in the TCU should be interpreted and the manner of the filtering to be performed, for each of the eight possible active texture stages. The structures of these registers are given below with the meaning of each field self-explanatory..

The **TextureIndexMode0..7** register:

Field Name	Description	
Width Height	Holds the width/height of the texture map as a power of two. The legal range of values for this field is 0 (= 1) to 13 (= 8192). If a border is present then the maximum value is 12, and for a 3D map it is 10 without a border, or 9 with a border.	
Depth	Holds the depth (i.e. number of slices) of the texture map as a power of two. The legal range of values for this field is 0 (= 1) to 10 (= 1024). *This field should be set to 0 for a 1D or 2D map.*	
Border	When set this indicates there is a one texel border surrounding the texture map.	
MapType	Selects the type of map and how many axis it has.	0 = 1D texture map 1 = 2D texture map 2 = 3D texture map 3 = Cube map
WrapU[3] WrapV[3] WrapW[3]	These fields selects how the u, v and w coordinate are to be wrapped to fit on the texture map.	0 = Clamp 1 = Repeat 2 = Mirror 3 = ClampEdge 4 = ClampBorder
MagnificationFilter	Selects the minification filter to use.	0 = Nearest 1 = Linear
MinificationFilter	Selects the minification filter to use	0 = Nearest 1 = Linear 2 = NearestMipNearest 3 = NearestMipLinear 4 = LinearMipNearest 5 = LinearMipLinear
FilterBank	When the filter mode is Nearest or Linear then this field will specify which filter bank to use in the Primary Texture Cache. Use of alternating banks will reduce thrashing between the texture maps in the cache.	

The **TextureIndexMipControl**[0..7] register:

Field Name	Description
MinLod	This field holds the minimum level of detail (lod) value. Any input lod less than this will be clamped to this value. Its format is 4.8 unsigned fixed point.
MaxLod	This field holds the maximum level of detail (lod) value. Any input lod greater than this will be clamped to this value. Its format is 4.8 unsigned fixed point.
BaseLevel	This 4bit field holds the map level which should be treated as level 0. Set to 0.
MaxLevel	This 4bit field holds the map level which should be treated as the last level in the mip-map chain. Set to 14.

8.5.2 Colour Lookup

This and the following sections consider the programs required for texture coordinate generation for actual texturing tasks. Perhaps the most common operation for texture coordinate generation is in the sampling of a 2D image texture map. The task of the texture coordinate unit program is to generate for each tile fragment, the interpolated perspectively correct texture coordinates and, depending on the type of filtering required, an estimate of the texture minification ratio for level of detail calculation. Unlike the vertex colours in the shading unit, the coordinates for each fragment are not generated directly by fixed function hardware, but by a user supplied program.

For 2D texture mapping the perspectively correct coordinates *s* and *t* for each pixel fragment are calculated by linearly interpolating the vertex supplied linear coordinates *S*, *T* and *Q* and applying:

$$s = S/Q \text{ and } t = T/Q$$

Where the *S*,*T* and *Q* values are initially determined for each vertex as a function of the parametrically supplied texture coordinates, multiplied by *Q*, a function of depth.

The texture minification (or compression) indicates how compressed or stretched a texture is when mapped to the screen pixels and so influences the filtering method used. This minification can be estimated from the change in *s* and *t* with respect to a change in the screen coordinate *x* and *y*. These partial derivatives can be calculated analytically as below:

$$ds/dx = ((dS/dx)Q - (dQ/dx)S) / Q^2$$

ds/dy, *dt/dx* and *dt/dy* are calculated similarly.

The minification may then be estimated from these values, for example by simple magnitude inspection to find the greatest axis of compression.

The plane equation partial derivatives and start values for the linearised coordinates *S*, *T* (plus possibly *R*) and *Q* are presented to the TCU by the parameter setup unit. An example program for MIPmap based texture lookup is then as follows:

```
// Evaluate the texture coordinate plane equations.
W[Q] =          MAdd(dPdx[@Q], X, dPdy[@Q], Y) SavedP;
W[S] =          MAdd(dPdx[@S], X, dPdy[@S], Y);
```

```
W[T] =          MAdd(dPdx[@T], X, dPdy[@T], Y);

// Add in the origin values.
Q2 = W[Q] =     MAdd(PStart[@Q], One, A[Q], One);
W[S] =          MAdd(PStart[@S], One, A[S], One);
W[T] =          MAdd(PStart[@T], One, A[T], One);

// Calculate the reciprocal of Q.
Div(One, B[Q]);

// Calculate the partial derivatives using the more accurate analytical method.
LoadMax  =      MSub(dPdx[@S], A[Q], dPdxSaved, B[S], PlaneScale[@S]);
MergeMax =      MSub(dPdx[@T], A[Q], dPdxSaved, B[T], PlaneScale[@T]);
MergeMax =      MSub(dPdy[@S], A[Q], dPdySaved, B[S], PlaneScale[@S]);
MergeMax =      MSub(dPdy[@T], A[Q], dPdySaved, B[T], PlaneScale[@T]);

// Wrap the S and T coordinates, performing the perspective correction by
// multiplying with the reciprocal of Q. Output the wrapped coords with the
// calculated mipmap lod value.
C[0] =    Wrap(DivResult, B[S]);
C[1] =    Wrap(DivResult, B[T]);
C[3] =    LOD FloatToInt(One) Return;
```

Where **dPdx, dPdy** and ***PStart*** are the plane equation registers holding the interpolation information for the vertex coordinates *S*,*T* and *Q* supplied by the parameter setup unit. The @ used in the plane equation register reads above causes the use of relative addressing. This is useful allowing the same code to be used for multiple plane equation information for the different texture stages; the *Return* at the end denotes this as a subroutine, which may be called multiple times. Such relative addressing can be similarly used for global registers. All that is required is that the plane equation and global register base address is setup in an instruction prior to use of the relative addressing. This can be done with the ***PlaneBase(pbase), GlobalBase(gbase)*** or combined ***PlaneGlobalBase(pbase,gbase)*** instruction options. The use of such relative addressing is demonstrated in the bump mapping program presented in the next section. Other features include the indirect addressing operator % demonstrated in the cube mapping subsection.

8.5.3 Bump Environment Mapping

This section describes the programming requirement for the powerful technique of bump environment mapping. This technique utilises the feedback path of the P10 texture pipe to allow the sampled result of one texture to be used to modify the texture coordinates used to sample another. That subsequent texture could be a simple 2D image, resulting in a distortion or patterning of its appearance (for example to simulate an underwater effect), or an environment or cube map in which the perturbed coordinates are themselves directional vectors such as surface normals or reflection vectors.

Let us consider the DirectX7 texturing operation D3DTOP_BUMPENVMAP. A bump map is sampled to produce (in the case of a 2D bump map) the signed perturbation values *dU* and *dV* to be fed back. These are then transformed by a 2x2 matrix such that:

dU' = dU.mat00 + dV.mat10
dV' = dU.mat01 + dV.mat11

These bump map matrix element values are supplied by the application and loaded into TCU global registers. An example of a multi-texture TCU program to perform this environment bump mapping is as follows:

```
//Stage 0 - Base Texture
PlaneGlobalBase(0,0) MAdd(A[0],A[0],A[0],B[0]) ;
W[9]= MAdd(dPdx[@3],X,dPdy[@3],Y) SavedP ;
W[0]= MAdd(dPdx[@0],X,dPdy[@0],Y) ;
W[3]= MAdd(dPdx[@1],X,dPdy[@1],Y) ;
Q2= W[9]= MAdd(PStart[@3],One,A[9],One) ;
W[0]= MAdd(PStart[@0],One,A[0],One) ;
W[3]= MAdd(PStart[@1],One,A[3],One) ;
Div(One,B[9]) ;
LoadMax= MSub(dPdx[@0],A[9],dPdxSaved,B[0],PlaneScale[@0]) ;
MergeMax= MSub(dPdx[@1],A[9],dPdxSaved,B[3],PlaneScale[@1]) ;
MergeMax= MSub(dPdy[@0],A[9],dPdySaved,B[0],PlaneScale[@0]) ;
MergeMax= MSub(dPdy[@1],A[9],dPdySaved,B[3],PlaneScale[@1]) ;
C[0]= Wrap(DivResult,A[0]) ;
C[1]= Wrap(DivResult,A[3]) ;
C[3]=LOD FloatToInt(One) ;
Command(FilterTexture,0,0,1,0,Default) MAdd(A[0],A[0],A[0],B[8]) ;

//Stage 1 - Bump Map Texture
PlaneGlobalBase(4,2) MAdd(A[0],A[0],A[0],B[0]) ;
W[9]= MAdd(dPdx[@3],X,dPdy[@3],Y) SavedP ;
W[0]= MAdd(dPdx[@0],X,dPdy[@0],Y) ;
W[3]= MAdd(dPdx[@1],X,dPdy[@1],Y) ;
Q2= W[9]= MAdd(PStart[@3],One,A[9],One) ;
W[0]= MAdd(PStart[@0],One,A[0],One) ;
W[3]= MAdd(PStart[@1],One,A[3],One) ;
Div(One,B[9]) ;
LoadMax= MSub(dPdx[@0],A[9],dPdxSaved,B[0],PlaneScale[@0]) ;
MergeMax= MSub(dPdx[@1],A[9],dPdxSaved,B[3],PlaneScale[@1]) ;
MergeMax= MSub(dPdy[@0],A[9],dPdySaved,B[0],PlaneScale[@0]) ;
MergeMax= MSub(dPdy[@1],A[9],dPdySaved,B[3],PlaneScale[@1]) ;
C[0]= Wrap(DivResult,A[0]) ;
C[1]= Wrap(DivResult,A[3]) ;
C[3]=LOD FloatToInt(One) ;
Command(FilterTexture,1,1,0,1,Default) MAdd(A[1],A[1],A[1],B[9]) ;

//Stage 2 - Environment Map Texture

PlaneGlobalBase(8,4) MAdd(A[0],A[0],A[0],B[0]) ;
W[9]= MAdd(dPdx[@3],X,dPdy[@3],Y) SavedP ;
W[0]= MAdd(dPdx[@0],X,dPdy[@0],Y) ;
W[3]= MAdd(dPdx[@1],X,dPdy[@1],Y) ;
Q2= W[9]= MAdd(PStart[@3],One,A[9],One) ;
```

```
W[0] = MAdd(PStart[@0],One,A[0],One) ;
W[3] = MAdd(PStart[@1],One,A[3],One) ;
Div(One,B[9]) ;
LoadMax= MSub(dPdx[@0],A[9],dPdxSaved,B[0],PlaneScale[@0]) ;
MergeMax= MSub(dPdx[@1],A[9],dPdxSaved,B[3],PlaneScale[@1]) ;
MergeMax= MSub(dPdy[@0],A[9],dPdySaved,B[0],PlaneScale[@0]) ;
MergeMax= MSub(dPdy[@1],A[9],dPdySaved,B[3],PlaneScale[@1]) ;
W[0] = MAdd(DivResult,B[0],Zero,Zero) ;
W[3] = MAdd(DivResult,B[3],Zero,Zero) ;
MAdd(A[0],A[0],A[0],B[0]) WaitForFeedbackData;
W[10] = MAdd(One,One,Zero,Zero,7) ;
W[14] = IntToFloat(Feedback(8,0,Zero)) ;
W[13] = IntToFloat(Feedback(8,8,Zero)) ;
W[14] = MSub(A[14],One,B[10],One,-7) ;
W[13] = MSub(A[13],One,B[10],One,-7) ;
W[10] = MAdd(A[14],Global0[@1],B[13],Global1[@1]) ;
W[0] = MAdd(A[0],One,B[10],One) ;
W[10] = MAdd(A[14],Global0[@2],B[13],Global1[@2]) ;
W[3] = MAdd(A[3],One,B[10],One) FinishedWithFeedbackData;
C[0] = Wrap(A[0],One) ;
C[1] = Wrap(A[3],One) ;
C[3]=LOD FloatToInt(One) ;
Command(FilterTexture,2,2,1,0,Default) MAdd(A[2],A[2],A[2],B[10]) Done;
```

The Command fifo write commands are broken down as follows:

Command(command, texID, destReg, loadshading, enablefeedback, prog)

The code shows the TCU program for a three-texture application, each stage partitioned by the appropriate command fifo writes indicating when feedback is initiated (***enablefeedback)*** and the target texture registers in the shading unit (***destReg***).

Although the bump map texture stage does not contribute any texel data itself to the texture registers in the shading unit, it is necessary to pass on the colours of the proceeding stage (if any, otherwise the diffuse colour) if the environment stage texture operation requires it. For example consider the DX7SDK sample BumpEarth; in that sample a patterned base texture is applied to stage N (e.g. image of an Earth map), a bump map to stage N+1 (e.g. based on relief map of the Earth), and an environment map to stage N+2 (e.g. simple highlights or a reflection map). Stage N+2 might then have a simple ADD texture operation to blend the results of the Earth image with the relief perturbed environment map. Apart from ensuring this colour is passed through no additional shading unit code is required for a bump environment map texture operation..

An extension to bump mapping that does require additional shading unit programming occurs with the introduction of luminance. Consider the texturing operation D3DTOP_BUMPENVMAPLUMINANCE. This is essentially the same as that given above with the important difference that a third component, luminance, is now sampled from the bump map and used to modulate the intensity of the resulting sampled texels of the environment map. The formula for this luminance part is:

$$L' = L.Scale+0ffset$$

Where L' is then used to modulate the environment map stage's filtered texels before their colour is used to blend with that of the base texture. The significance here is that while the luminance data is taken from the bump map in the same way as the coordinate perturbation values dU and dV, it is applied to the results of the environment map and not the coordinates used to sample it. Therefore the luminance component is not sent via the feedback fifo to the TCU, but passed to the shading unit (SU) requiring the additional SU functionality. The luminance value scale and offset are applied in the shading unit with the values having been loaded into global registers in the shading unit. As well as calculating L', this value must then be used to modulate the RGB channels of the environment map. This has the added effect that the results of the environment map stage (e.g. Stage N+2) are now no longer stored in shading unit texture registers, but in local registers and that stages texture operation (e.g ADD in the given example) must adjust appropriately.

Unbounded Luminance Scale and Offset: While L' and L are expected to be limited to the range 0-1, the scale and offset values are not. To allow values greater than 1 to be loaded for scale and offset via the 8bit shading unit global registers, an arbitrary unsigned fixed point format of 2.6 was selected. The driver converts the value to this range and the shading unit expands from this to its internal s.3.8 format in a load from the global register to an internal working register.

Signed Textures and their download: A feature of bump normal perturbation maps is that they contain signed data. The dU and dV parts of the bump map are generally created and stored by applications as two's complement signed integers. Although the TCU could correctly handle these by reading from the feedback fifo with sign extension and then performing a signed IntToFloat conversion, it is the texture filter unit that performs the filtering of the bump map and this uses interpolators that do not consider the signed case. To allow for the correct filtering of these values, the driver intercepts the download of these textures and introduces a +128 bias to the 2's complement 8bit signed fields. For simplicity lower resolution formats such as 6L5V5U bump maps are converted to 8L8V8U at this time. The texture is then filtered as a 0-255 intensity image and the bias removed, as shown in the program above, as the feedback dU and dV data is converted from integer to float, unbiased with a subtraction of 128 and finally scaled to the required +-1 range.

8.5.4 Cube Mapping

Cubemapping is a relatively simple extension to 2D texture mapping from the point of view of the chip. The essence of the technique is that an app provides a texture with six faces (optionally mipmapped) which correspond to the positive and negative directions of each of the three axes (X, Y and Z) and texture coordinates that act as reflection vectors. These reflection vectors are then used to choose a particular face from the cubemap and within that face to generate a set of 2D texture coordinates.

The face is chosen by finding the major (largest) component of the reflection vector (encoded as a three component texture coordinate). The other two components are then divided by this major component which effectively projects those two components on to the chosen face. Because the faces are oriented looking into the centre of the cube which they form, the signs of the generated 2D texture coordinates must be adjusted according to which face is chosen. Additionally the projected coordinates are in the range [-1,1] and so they must be translated and scaled into the range [0,1] for normal texture mapping.

The division that performs the projection is also in exactly the right place to perform the perspective divide - the 3D texture coordinates (i.e. the reflection vector) would have been scaled by the reciprocal of homogeneous w (RHW) in the Vertex Shader unit and then RHW gets divided out again during the face projection.

It is possible to compute the level of detail (LOD) for mipmapping analytically by computing the full derivatives from the partial derivatives defining the plane equations and the formula for the derivative of a quotient (due to the divide by the major component). The partial derivatives must be scaled to match the transformation of the 2D coords from [-1,1] to [0,1]. Various approximations to the LOD can be made by combining the full derivatives in different ways e.g. taking the maximum of the absolute values and/or using an octagonal approximation to the square root of the sum of the squares of the derivatives. Just as for 2D mipmapping the miplevels of a cubemap must be laid out in a fixed pattern - all of the miplevels for all of the faces are contiguous in graphics memory. A cubemap must have square, power-of-two faces and so therefore must any mipmap levels. The chip computes the total size of one face's mipmap chain (including any rounding up to tile units for the small miplevels) and uses that as a stride to access the miplevels of the other faces. This means that the driver must always allocate space for all the miplevels down to 1x1 even if the app does not provide them.

The plane equation evaluation requires a planeScale value for each dimension of the texture map to convert from texture coordinates into texels. As a cubemap must be square all three dimensions should be set to the same planeScale value e.g. 8 for a 256x256 cubemap.

The chip does not provide direct support for filtering across the faces but this can be implemented by using texture borders to replicate the edge texels of the adjacent faces on to a given face. However DXT compressed textures cannot have borders and so this techniques will not work for DXT mipmapped cubemaps.

The chip provides a TextureCoord Unit instruction called CubeSort which determines the major component and sets up the data to support an indirect addressing mode (indicated with the % symbol) so that the major component is always in a known place from the point of view of the TCU program. This indirect addressing mode, which is solely for the use of cube-mapping programs, also automatically accounts for the sign adjustments mentioned above. To support this indirect addressing mode the texture coordinates and their two derivatives must be laid out in a fixed manner i.e..

```
    Define(reg_S,     0)
Define(reg_dSdx,      1)
Define(reg_dSdy,      2)
Define(reg_T,         3)
Define(reg_dTdx,      4)
Define(reg_dTdy,      5)
Define(reg_R,         6)
Define(reg_dRdx,      7)
Define(reg_dRdy,      8)
```

The other registers can be allocated in any manner that doesn't conflict with above mapping e.g.

```
Define(reg_deriv0,    10)
Define(reg_deriv1,    11)
```

```
Define(reg_dsdx_,       12)
Define(reg_dsdy_,       13)
Define(reg_dtdx_,       14)
Define(reg_dtdy_,       15)

Define(reg_dx,          7)
Define(reg_dy,          8)

Define(reg_S_,          7)
```

The following TCU cubemapping program using the octagonal approximation to calculate the LOD. The global registers contain the magic values (11/32) and (21/32) and CubeScale is -1.

```
// Extract the current stage's texture coordinates from the plane equations.
W[reg_S] =MAdd(dPdx[@plane_S], X, dPdy[@plane_S], Y);
W[reg_T] =              MAdd(dPdx[@plane_T], X, dPdy[@plane_T], Y);
W[reg_R] =MAdd(dPdx[@plane_R], X, dPdy[@plane_R], Y);

// Add in the origin values.
W[reg_S] =MAdd(PStart[@plane_S], One, A[reg_S], One);
W[reg_T] =              MAdd(PStart[@plane_T], One, A[reg_T], One);
W[reg_R] =MAdd(PStart[@plane_R], One, A[reg_R], One);

LoadDivResult(A[reg_S]);
CubeSort(DivResult, A[reg_T], B[reg_R]);

// Start the reciprocal of R calculation and load R into the lod logic.
Div(One, B[%reg_R]);
Q2 =    Select(A[%reg_R], A[%reg_R]);

// Move the pPdx and dPdy values to the local registers so indirect
// addressing can be used. Scale by the planescale values.

W[reg_dSdx] =MAdd(dPdx[@plane_S], One, Zero, Zero, PlaneScale[@plane_S]);
W[reg_dTdx] =MAdd(dPdx[@plane_T], One, Zero, Zero, PlaneScale[@plane_T]);
W[reg_dRdx] =MAdd(dPdx[@plane_R], One, Zero, Zero, PlaneScale[@plane_R]);

W[reg_dSdy] =MAdd(dPdy[@plane_S], One, Zero, Zero, PlaneScale[@plane_S]);
W[reg_dTdy] =MAdd(dPdy[@plane_T], One, Zero, Zero, PlaneScale[@plane_T]);
W[reg_dRdy] =MAdd(dPdy[@plane_R], One, Zero, Zero, PlaneScale[@plane_R]);

// Calculate the partial derivatives analytically. Scale by 1/R^2 will be done in the lod unit.
// NOTE: This suffers from stalls, some of which could be avoided.

W[reg_deriv0] =   MAdd(A[%reg_R], B[%reg_dSdx], Zero, Zero);
W[reg_deriv1] =   MAdd(A[%reg_S], B[%reg_dRdx], Zero, Zero);
W[reg_dsdx_] =    MSub(A[reg_deriv0], One, B[reg_deriv1], One, CubeScale);
```

```
W[reg_deriv0] =     MAdd(A[%reg_dTdx], B[%reg_R], Zero, Zero);
W[reg_deriv1] =     MAdd(A[%reg_T], B[%reg_dRdx], Zero, Zero);
W[reg_dtdx_] =      MSub(A[reg_deriv0], One, B[reg_deriv1], One, CubeScale);

W[reg_deriv0] =     MAdd(A[%reg_dSdy], B[%reg_R], Zero, Zero);
W[reg_deriv1] =     MAdd(A[%reg_S], B[%reg_dRdy], Zero, Zero);
W[reg_dsdy_] =      MSub(A[reg_deriv0], One, B[reg_deriv1], One, CubeScale);

W[reg_deriv0] =     MAdd(A[%reg_dTdy], B[%reg_R], Zero, Zero);
W[reg_deriv1] =     MAdd(A[%reg_T], B[%reg_dRdy], Zero, Zero);
W[reg_dtdy_] =      MSub(A[reg_deriv0], One, B[reg_deriv1], One, CubeScale);

// Take abs values

W[reg_dsdx_] =      AMax(A[reg_dsdx_], Zero);
W[reg_dtdx_] =      AMax(A[reg_dtdx_], Zero);
W[reg_dsdy_] =      AMax(A[reg_dsdy_], Zero);
W[reg_dtdy_] =      AMax(A[reg_dtdy_], Zero);

// Compute octagonal approx. to Euclidean distance

W[reg_deriv0] = MAdd(A[reg_dsdx_], Global0[global_fact0], B[reg_dtdx_],
    global0[global_fact0]);
W[reg_deriv1] = AMax(A[reg_dsdx_], B[reg_dtdx_]);
W[reg_dx] = MAdd( A[reg_deriv0], One, B[reg_deriv1], Global1[global_fact0]);

W[reg_deriv0] = MAdd(A[reg_dsdy_], Global0[global_fact0], B[reg_dtdy_],
    Global0[global_fact0]);
W[reg_deriv1] = AMax(A[reg_dsdy_], B[reg_dtdy_]);
W[reg_dy] = MAdd( A[reg_deriv0], One, B[reg_deriv1], Global1[global_fact0]);

// Take the max of the lengths in x and y and take log2 ( implicit in LoadMax special function
    )

LoadMax = AMax(A[reg_dx], B[reg_dy]);

// Calculate (%S/%R + 1.0) * 0.5 (and similarly for %T).
// This is the division by the major coordinate and rescaling to [0,1].

W[reg_S_] =         MAdd(DivResult, A[%reg_S], One, One, CubeScale);
W[reg_T] =              MAdd(DivResult, A[%reg_T], One, One, CubeScale);

// Convert to the correct output format.

C[0] =    Wrap(A[reg_S_], One);
C[1] =    Wrap(A[reg_T], One);
C[3] =    LOD FloatToInt(One);          // Op=nop
```

8.6 Localbuffer processing (setting up the mode registers)

This section describes the programming required to setup the local buffer processing on the P10 core, largely from the perspective of the Direct3D driver. More specific application details, such as for multi-sample antialiasing, are included in the relevant sections of the document.

Local buffer memory management and base address control are setup with the **LBMode** tag. This has the following fields.

Field Name	Description
SameTileEnable	When set this enables 'same tile' caching. If this tile has the same coordinates as the previous tile then this provides a performance optimization where successive primitives are likely to be in the same tile.
Width	This field holds the width in tiles of the buffer. The range is 0…2047.
PixelBytePitch	This field defines the offset between tiles in memory, normally equal to the depth of a tile in bytes. The range is 1…8.
OffsetBetween Buffers	This field holds the offset between successive multi-sample buffers. It is measured in multiples of 1024 byte tiles.

The *Width* and *PixelBytePitch* fields are set based on the format of the local buffer surface target in use. Note that this refers to the whole local buffer including both depth and stencil (and GID for OpenGL).

For rendering *SameTileEnable* is usually set to 1 to take advantage of any optimization that may result.

Individual control of the Depth, Stencil and GID buffers is performed with the **DepthMode**, **StencilMode** and **GIDMode** registers.

The structure of the **GIDMode** is as follows:

Field Name	Description
Enable	When set this enables GID testing to be done.
Reference	Used when Enable set. Holds the GID value to test each pixel against.
Present	When set this indicates the local buffer pixel format includes the GID field. GID field is always the least significant byte if present.
EarlyExitProcessing	When set this enables early exit processing for the GID, stencil and depth tests.

Of particular significance here is the *EarlyExitProcessing* which provides optimisation through enabling early termination of tests. As said, this affects stencil and depth tests and so is relevant even when there is no GID present.

The structure of the **StencilMode** register is:

Field Name	Description

Enable	When set this enables the stencil test and the replacement of the stencil value dependent on the outcome of the stencil and depth test. Else always pass and stencil data not updated.	
DPpass	How stencil buffer updated when both stencil and depth test pass.	0 = Keep (i.e. local buffer value not changed) 1 = Zero 2 = Replace with StencilData.Reference 3 = Increment (with saturation) 4 = Decrement (with saturation) 5 = Invert 6 = Increment (with wrapping) 7 = Decrement (with wrapping)
DPfail	How stencil buffer updated when stencil test passes and depth test fails.	
Sfail	How stencil buffer updated when stencil test fails.	
CompareFunction	Selects the test used to compare the buffer stencil value with the reference supplied in StencilData.	0 = Never 1 = Less 2 = Equals 3 = Less Equals 4 = Greater 5 = Not Equal 6 = Greater Equal 7 = Always
Present	When set this indicates the local buffer pixel format includes a stencil field. The stencil field is always the byte following the GID field (if present) or byte 0 if there is no GID field.	

As an example of setting up this register consider the Direct3D case. For D3DRENDERSTATE_STENCILENABLE false all fields are zero. When true Present and Enable are set and the test fields assigned the values indicated by D3DRENDERSTATE_STENCILFAIL (Sfail), D3DRENDERSTATE_STENCILZFAIL (DPfail), and D3DRENDERSTATE_STENCILPASS (DPpass) and D3DRENDERSTATE_STENCILFUNC (comparefunction).

Closely related to this register and assigned at the same time in the driver is the **StencilData** messsage of the format shown below:

Field Name	Description
Reference	Holds the value the stencil data from the local buffer is compared with.
CompareMask	Holds the mask which is ANDed with both the stencil data read from the local buffer and the reference stencil value prior to their comparison is done.
WriteMask	Holds the mask used to only allow certain bits in the local buffer stencil field to be updated.

Again these correspond directly to D3D renderstate: D3DRENDERSTATE_STENCILREF, D3DRENDERSTATE_STENCILMASK and D3DRENDERSTATE_STENCILWRITEMASK.

Depth buffering is controlled by the **DepthMode** register, of the format:

Field Name	Description	
Enable	When set enables the depth test and the replacement of depth value dependent on the outcome of the test.	
WriteMask	When set this allows the depth value in the buffer to be updated.	
Compare Function	Selects the test used to compare the buffered depth value with the new iterated source value.	0 = Never 1 = Less 2 = Equals 3 = Less Equals 4 = Greater 5 = Not Equal 6 = Greater Equal 7 = Always
Width	This field holds the width in bits of the depth field in local buffer.	0 = 16 bits wide 1 = 24 bits wide 2 = 32 bits wide 3 = 16 bits wide
Format	Format of the Z value in the local buffer.	0 = Integer 1 = Floating Point
Complement	When set this causes the set up calculations to be done with 1.0 - Z value at each vertex rather than Z. This, in conjunction with a floating point Z format allows a non linear Z buffer to be used.	
SamplePoint	Determines where the sample point in a pixel is considered to be.	0 = (at 0.5, 0.5) OpenGL 1 = (at 0.0, 0.0) D3D
MultiSample Enable	Enables multi-sample processing of the local buffer.	
MultiSampleMask	Normally has the same number of bits set (starting from bit 0) as there are sub pixels samples set up in the rasteriser, but setting fewer bits allows use of effects such as motion blur and depth of field.	
UseAllSub Samples	When set, this indicates that all the sub pixel buffers are to be written, i.e. one or more buffers are not going to be skipped to implement motion blur, etc.	

Again the fields are set based on the API render state and the required buffer dimensions. For Direct3D the *Enable* bit is set if D3DRENDERSTATE_ZENABLE is equal to either D3DZB_TRUE or D3DZB_USEW. The latter case refers to use of non-linear W-buffer and if set by the application causes both the *Format* and *Complement* fields to be set to 1, which would otherwise be 0. *WriteMask* is set based on the status of D3DRENDERSTATE_ZWRITEENABLE. The last three fields are described in more detail in the latter section on full scene antialiasing (FSAA).

Finally, it is also necessary to set up the memory base address for the local buffer. This is performed using the **LBBaseAddr** register, which simply expects a DWORD (the address is actually a 28bit value) containing the start base address of the memory allocated for the local buffer in byte tiles.

For example in the DX driver:

```
LBBaseAddress = pContext->ZPixelOffset + dwLBByteOffset;
    SEND_P10_DATA(LBBaseAddr, LBBaseAddress );
```

Where pContext->ZPixelOffset is calculated from the byte position of the allocated local buffer surface, divided by 64 to give the address in byte tiles. Due to the fact that the stencil is always the lower significant byte in the local buffer memory words, dwLBByteOffset is used to offset the address by the byte depth of the stencil buffer (0 or 1 in P10) when the stencil buffer exists but is not enabled.

*NOTE: It is necessary to resend a **RouterMode** command following DepthMode, StencilMode or StencilData update to avoid a race condition in the local buffer cache access.*

8.7 Framebuffer processing (Dithering, Logical Ops, Blending, Accumulation buffers/deep buffers)

Frame buffer processing is the domain of the programmable Pixel Unit (PU) and Pixel Address Unit (PAU). Basic mode configuration for these units and frame buffer processing are as follows.

8.7.1 Configuring the Frame Buffers

Configuration of the pixel address unit for access to the assigned frame buffer memory is controlled by the **FBMode**, **FBBuffer** and **FBBaseAddr** registers.

The **FBMode** register has the following structure:

Field Name	Description
SameTileEnable	Intended to provide rendering optimization for small primitives.
EntryPoint	Holds the 5bit start address in the pixel address unit's program memory to control the program to run when a tile command is received.

Of importance here is the *EntryPoint* field which controls the starting point for the programmable pixel address units instruction sequencer. Usually, only a single PAU program is stored and this is therefore normally 0.

Up to five separate frame buffers may be configured using the register **FBBuffer0**..**FBBuffer3** and **FBBufferGlobal**, where the latter is a special case that provides a common global memory buffer. Usually just **FBBuffer0** and possibly **FBBufferGlobal** are used, with **FBBuffer1** to FBBuffer3 used for features such as front and back buffers and left and right stereo. The **FBBuffer** registers are of the form:

Field Name	Description
ReadEnable	Enables reads to this frame buffer. Only used for destination addresses, this bit is ORed with the corresponding bit in the FBBufferReadEnables tag and so can be alternatively controlled from there.

Field Name	Description
AAReadOnly	Used together with ReadEnable, if set then frame buffer reads only done when the tile has aaEnable set. When aaEnable not set then this means that all fragments have 100% coverage and no destination blending will be done. Note that this is only useful for depth dependent antialiasing with no alpha transparency.
Width	Width in tiles of the buffer. Range 0..2047.
PixelBytePitch	The offset between tiles in memory in bytes (normally equal to the tile colour depth). Range 1..16.
PixelSize	Defines the number of bytes read from memory (+1, so 0 equals 1byte) and transferred to the cache. Normally this is the depth of a tile in bytes, but can be less if a subset of the field needs to be read and/or written. Range 0..3.
SubFieldStartByte	Defines the first byte to transfer from the cache to the Pixel Unit. Normally this is set to zero, but can be different if a subset of the field needs to be read and/or written.
SubFieldByte Count	Defines the number of bytes to transfer from the cache (+1) to the Pixel Unit. Normally this is the same as the PixelSize, but can be less if a subset of the field needs to be read and/or written.
XMask	When set this will cause the x coordinate to be ANDed with the xMask register before the address is calculated. This can be used for pattern replication. These masks are loaded as part of the PAU instruction set.
YMask	When set this will cause the x coordinate to be ANDed with the xMask register before the address is calculated. This can be used for pattern replication.
Height	Holds the height of the buffer region in tiles. Used only for clipping tile reads outside the region as can occur when aligning tiles in a source read.

The **FBBufferEnables** tag is used to define which of the four possible buffers the program will use. If no buffers are enabled then the frame buffer subsystem is disabled and no programs in the Pixel Address Unit or in the Pixel Unit will be run.

The **FBBufferReadEnables** tag defines the read enable status of the possible buffers. Bit 0 corresponds to buffer 0, bit 1 to buffer1, and so on and bit 4 to the global buffer. The status is ORed with the ReadEnable of the **FBBuffer** register so either can be used.

The base address for each existing frame buffer is setup with the **FBBaseAddr0..3** and **FBBaseAddrGlobal** tags. Addresses are given in byte tiles.

An example of frame buffer configuration in the DX driver follows:

```
SEND_P10_DATA( FBBufferEnables, 1 );
SEND_P10_DATA( FBMode,                  FBMode(SameTileEnable, FALSE ) |
        FBMode(EntryPoint, 0 ) );
// Configure destination buffer
SEND_P10_DATA( FBBuffer0,               FBBuffer(ReadEnable, TRUE ) |
    FBBuffer(AAReadOnly, FALSE ) |
    FBBuffer(Width, DDSurf_Pitch(lpBlt->lpDDDestSurface)/(DestPixelSize<<3) ) |
    FBBuffer(PixelBytePitch, DestPixelSize  ) |
    FBBuffer(PixelSize, DestPixelSize-1 ) |
    FBBuffer(SubFieldStartByte, 0 ) |
```

```
        FBBuffer(SubFieldByteCount, DestPixelSize-1 ) |
        FBBuffer(MaskX, 0 ) |
        FBBuffer(MaskY, 0 ) |
        FBBuffer(Height, 0 ) );

    FBBaseAddress = ((lpBlt->lpDDDestSurface->lpGbl->fpVidMem
            - pThisDisplay->dwScreenFlatAddr)/64);
    SEND_P10_DATA( FBBaseAddr1,FBBaseAddress );
```

Further examples of using these registers are included in the section on full scene antialiasing (FSAA).

8.7.2 Loading the Pixel Unit and Pixel Address Unit Programs

The Pixel Address Unit (PAU) controls the memory access operations during frame buffer processing. PAU programs are loaded two instructions at a time using the **FBProg** register. The first 15bit PAU instruction is held in the lower half of the passed DWORD and the second in the upper half.

A simple example of a PAU program used for alpha blending in the DX driver is:

```
Program(PAUDestReadWrite, 0x00)
SendDestAddrAndTile(buf0, puReg0, Only);
```

This causes the destination buffer to be set to the address of FBBuffer0 and spawns a tile command that is sent to the Pixel Unit indicating that the subprogram indicated by the *Only* address (see **PixelMode** below) should be run.

Constant data can be loaded for use in these programs via the **FBAddrInfo** registers. Examples of the use of such constants and more advanced PAU programs are provided in the later section on full scene antialiasing (FSAA). A description of the full instruction set is provided in the PAU assembler documentation referenced in P10 Reference Guide volume 3, section 1.2.4.3, Pixel Address Unit Instruction Set.

The Pixel Unit (PU) programs perform the processing of the colour channel data read from the frame buffer and received from the shading pipes and the following sections on blending and dithering provide examples of the practical use of such programs. As for the PAU, a guide to the full instruction set for the pixel unit is provided in the PU assembler documentation. Here we introduce the registers required to load these programs and setup for their correct operation.

The **PixelMode** register is used to configure the program start addresses and is of the form:

Field Name	Description
TileAddrOnly	Holds the 7bit address (128 instruction program space) of the program to run when a Tile command is received with a progID of 0 and the sameTile bit in the Tile command is 0.

TileAddrFirst	Holds the address of the program to run when a Tile command is received when the progID is 1.
TileAddrMiddle	This field holds the address of the program to run when a Tile command is received when the progID is 2. It also holds the address of the program to run when the progID is 0, but the sameTile bit in the Tile command is set.
TileAddrLast	TileAddrLast This field holds the address of the program to run when a Tile command is received when the progID is 3.

These fields allow up to four subprograms to be loaded into the pixel unit instruction space. The program entry point to be run is determined by the PAU unit program which controls the operation of this unit. For example the PAU will receive a **tile** command from the rasterisation process, which initiates the running of the PAU code. The instructions *SendTile(progID)*, *SendSourceAddrAndTile(buffer, puReg, progID)* or *SendDestAddrAndTile(buffer, puReg, progID)*, in the PAU program then spawn a new **tile** comand that is passed to the pixel unit and runs a program there starting from the appropriate address as indicated by the *progID* and the **PixelMode** register.

In the example above the progID was set to Only (interpreted as =0 by the assembler). For more complicated multi-pass frame buffer processing a First program can be used for register initialization in the pixel unit, a Middle program used in a loop for intermediate processing (e.g. accumulation) and a final Last program for result output. Examples of such programs are included in the following subsections and in the section on full scene antialiasing.

The sameTile bit is included in the **tile** command received from the pixel address unit when sameTileEnable is set in the **FBMode** register and the same tile is processed consecutively. Essentially this allows alternative programs to be stored, one that reads the pixel cache and another, pointed to by the TileAddrMiddle address when sameTile is set, that uses local working registers assumed to hold the previous processed contents of the pixel cache. For example:

```
W[0] = C[0] = Lerp (P[0], F[0], B[15]) Done;
sameTile:
W[0] = C[0] = Lerp (A[0], F[0], B[15]) Done;
```

To load pixel unit programs to the core the **PixelProgramAddr** and **PixelProgramData** commands are used. The **PixelProgramAddr** command loads the corresponding register with the address in the PU instruction space where the subsequent instruction, loaded with a **PixelProgramData** register, will be loaded. Instructions are loaded one at a time. The **PixelProgramData** register causes the program address to be incremented so only a single **PixelProgramAddr** tag need be sent for each set of consecutve instructions loaded. When loading multiple subprograms such as for the First, Middle and Last multi-pass approach described previously, a record of the program address should be maintained and incremented in the driver to allow the correct setting of the **PixelMode** register.

Pixel Unit programs can also make use of registers included in the unit for storage of constants. The unit possesses 32 global byte registers, which can be accessed by any of the tile SIMD processing elements. For example:

```
E = Global[0]    W[8] = PassB( E );
```

These are loaded in groups of four using the PixelGlobal register tag. For example …

SEND_P10_DATA(PixelGlobal0, (Val1 << 8 | Val0));

…will load byte values Val0 and Val1 to global registers 0 and 1 respectively.

As well as such global access registers, each SIMD processing element has a 32bit fragment data register available to it, accessible in the microcode as four byte registers using the syntax::

E = Fragment[1] W[0] = AddS(E, F[0]);

These are loaded 32bits at a time with the **UserFragData** tag. For example

SEND_P10_DATA(UserFragData8, (Val3 << 24 | Val2 << 16 | Val1 << 8 | Val0));

The following section on dithering provides an example of the use of these registers.

Although programs are usually triggered by the **Tile** command generated by the rasteriser, it is possible to initiate program runs manually using the **RunPixelProg** command. One application is in the running of a 'start of day' programs that reset or load local registers for later use. The format of the data sent with this register is as follows:

bits 0…6	Run address
bits 7…14	Run data
bits 15…19	Pass number
bit 30	enableData
bit 31	enableRun

8.7.3 Blending

Perhaps the most important rendering task performed in frame buffer processing is that of blending the newly rendered pixel fragments with those already in the frame buffer. This section describes this process from the perspective of the Direct3D API.

The general formula for frame buffer blending is as follows:

*DestNew(RGBA) = (DestOld(RGBA) * DestFactor) OP (Source(RGBA) * SourceFactor)*

The formula is split into two product terms and a combining operation, usually an addition. Here *Source* is the newly rendered pixel fragment and the *DestNew* and *DestOld* values refer respectively to the value to be written to the frame buffer and the value currently there. All refer to each of the RGBA colour channels. *DestFactor* and *SourceFactor* are factors used to modulate the source and destination colour channels before they are combined. A common example used for transparency blending is:

*DestNew(RGB) = (DestOld(RGB) * (1-Source(A))) + (Source(RGB) * Source(A))*

See the Direct3D SDK or OpenGL Red Book for other variations.

To support the numerous possible combinations of this formula, the DX driver splits the program generation into the concatenation of pre-assembled subprograms referring to:

- Destination formatting when required, e.g. 1555 must be converted to the internal 8888 format used in calculations.
- Destination blend factor loading - first colour (RGB) and then alpha (A)
- Destination colour component and blend factor modulation (the left-hand product term)
- Source blend factor load - first colour (RGB) and then alpha (A)

- Source colour component and blend factor modulation (the righthand product term)
- Combine source and destination product terms with blend op (e.g. Add)
- Perform dithering if required (see next section)
- Construct output format and write to cache (e.g. internal 8888 to 1555)

These concatenate into a single program and so **PixelMode** is simply loaded with a zero and the simple pixel address unit program example, *PAUDestReadWrite*, introduced previously is used.

Below is an example of 1555 format (without dithering) applying the transparency blending formula given above. Each sub-program is indicated.

```
Destination Formatting
//First the blue component (can use the intrinsic 565 mechanism)
        W[0] = PassA( P[0.L] );
//Then green, the most complicated
E = 64  W[1] = MultL( P[1], E );                    //Get upper 2 bits of G
E = 64  W[2] = MultU( P[0], E );                    //Get lower 3 bits of G
        W[2] = Or( A[1], B[2] );s                   //Combine the two parts
E = 0xF8 W[2] = And( A[2], E );                     //Lower 3 bits need clearing
E = 8   W[1] = MultU( A[2], E );                    //Now get upper 3 bits for replication
        W[2] = Or( A[2], B[1] );                    //and apply
//Now get red
E = 2   W[1] = MultL( P[1], E );                    //Shift over alpha bit
E = 0xF8 W[1] = And( A[1], E );                     //Mask out shifted green bits
E = 8   W[4] = MultU( A[1], E );                    //Now get upper 3 bits for replication
        W[4] = Or( A[1], B[4] );                    //and apply
//Now alpha
E = 2   W[6] = MultU( P[1], E );
E = 1   W[6] = ~Sub( A[6], E );
//Load the destination factor: 1 - Source Alpha
        W[8] = ~PassB( F[3] );
        W[10] = PassB( B[8] );
        W[12] = PassB( B[8] );
        W[14] = ~PassB( F[3] );         // Actually constructed from separate alpha path
//Perform the destination product (upper byte only approximation)
        W[9]  = Modulate( A[0], B[8] );             // blue
        W[11] = Modulate( A[2], B[10] );            // green
        W[13] = Modulate( A[4], B[12] );            // red
        W[15] = Modulate( A[6], B[14] );            // alpha
Load the destination factor: Source Alpha
        W[0] = PassB( F[3] );
        W[2] = PassB( B[0] );
        W[4] = PassB( B[0] );
        W[6] = PassB( F[3] );
Perform the Source product (upper byte only approximation)
        W[1] = Modulate( A[0] , F[0] );
        W[3] = Modulate( A[2] , F[1] );
```

```
            W[5] = Modulate( A[4] , F[2] );
            W[7] = Modulate( A[6] , F[3] );
Perform addition of the products ( upper byte only approximation)
            W[0] = AddS( A[1], B[9] );
            W[1] = AddS( A[3], B[11] );
            W[2] = AddS( A[5], B[13] );
            W[3] = AddS( A[7], B[15] );
Destination formatting for output
    E = 0x80 W[3] = And( E, B[3] );
    E = 32   W[0] = MultU( A[0], E );          //Blue component
    E = 4    W[4] = MultL( A[1], E );          //lower 3 bits of green
    E = 0xE0 W[4] = And( A[4], E );
             C[0] = Or( A[0], B[4] );
    E = 4          W[1] = MultU( A[1], E );    //Upper 2 bits of green
    E = 128  W[4] = MultU( A[2], E );
    E = 0x7C W[4] = And( A[4], E );            //Separate red component
             W[2] = Or( A[1], B[4] );
             //Combine green and blue
             C[1] = Or( A[2], B[3] ) Done;
             //and alpha
```

Note that this lengthy 1555 example has been included to indicate the overall process. While the program has used the fact that the final required format is a reduced precision 5bit to simplify the processing to the upper byte only in the product and addition operations (otherwise double byte arithmetic is needed), it is still fairly large. The more common 565 and 8888 formats do not require the manual format conversion and are therefore smaller. Also it is possible to treat the transparency blend above as a special case and utilize the Lerp instruction available in the pixel unit for a more efficient implementation. For example the 565 special case version of the same operation is:

```
W[4] = PassB( F[3] );
C[0.L] = Lerp( P[0.L], F[0], A[4] );
C[0.M] = Lerp( P[0.M], F[1], A[4] );
C[1.H] = Lerp( P[1.H], F[2], A[4] ) Done;
```

Other commonly used blend modes can be similarly optimised.

8.7.4 Dithering

Colour formats less than 32bit (e.g. 16 bit 565) can show obvious colour quantisation. One way to improve this is to perform colour dithering where essentially reduced colour resolution is replaced by spatial averaging. The colours/intensity of groups of neighbouring pixels are jittered about the required colour to provide the illusion of a greater colour depth. This is achieved by the addition of the contents of a dither matrix to the image pixels with the elements of the matrix corresponding to a particular pixel grid element. Correct screen/window alignment of this matrix is simplified by the SIMD tile operation of frame buffer processing in P10, where the 4x4 dither matrix can simply be aligned to the elements of the tile array.

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

The matrix shown in the above diagram is suitable for a 4bit channel and this is the initial matrix used in the DirectX driver. For 5 and 6 bits the matrix value must be right shifted by 1 or 2 bits respectively.

As introduced earlier each pixel unit fragment processor possesses four User Fragment Data registers that may be loaded for their personal use. These are used here to hold the dither value of the appropriate aligned matrix element using the **UserFragData** tag as follows:

```
// Supports 6 bit (UserFragData[0]) and 5 bit (UserFragData[1]) channels.
for( y=0; y<TILEHEIGHT; y++ )
{
    for( x=0; x<TILEWIDTH; x++ )
    {
        dwPackedMatrix = (DWORD)( DitherMatrix[ (x % 4) + ((y % 4) * 4)] >> 1 );
        dwPackedMatrix = (dwPackedMatrix<<8) + (dwPackedMatrix >>1 );
        SEND_P10_DATA_OFFSET(UserFragData0, (x+(y*TILEWIDTH)),
    dwPackedMatrix);
    }
}
```

Note that the supported colour formats require up to two values to be loaded, e.g. 5 and 6 bits for 565, just 5 bits for 555 or 4 bits for 4444. The matrix values are shifted appropriately during this one off loading process. The values are then in place for use in the pixel unit code, only requiring reloading if the fragment registers are required for another purpose during the rendering process.

These registers can be accessed in pixel unit programs by use of the Fragment[] argument. For example the application of dithering to the 565 alpha transparency blend program included in the last section would be:

```
        W[4] = ~PassB( F[3] );
        W[0] = Lerp( P[0.L], F[0], A[4] );
        W[1] = Lerp( P[0.M], F[1], A[4] );
        W[2] = Lerp( P[1.H], F[2], A[4] );
E = Fragment[1]     C[0.L] = AddS( E, B[0] );          // 5bit
E = Fragment[0]     C[0.M] = AddS( E, B[1] );          // 6bit
E = Fragment[1]     C[1.H] = AddS( E, B[2] ) Done;     // 5bit
```

8.7.5 Accumulation Buffers

The accumulation buffer is an OpenGL defined buffer. This buffer consists of four channels (R, G, B, and A) and cannot be rendered to directly. The accumulation buffer is controlled exclusively through the following routine (except to clear it):

```
void glAccum( operation, float value );
```

The possible operations are ACCUM, LOAD, RETURN, MULT, and ADD. The accumulation buffer operations apply identically to every affected pixel. Accumulation buffer values are taken to be signed values in the range [-1; 1].

Operations:

ACCUM - obtains R, G, B, and A components from the buffer currently selected for reading as a value in [0; 1]. Each result is then multiplied by *value*. The results of this multiplication are then added to the corresponding color component currently in the accumulation buffer, and the resulting color value replaces the current accumulation buffer color value.

LOAD - has the same effect as ACCUM, but the computed values replace the corresponding accumulation buffer components rather than being added to them.

RETURN - takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by value, and clamps the results to the range [0; 1] The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization.

MULT - multiplies each R, G, B, and A in the accumulation buffer by value and then returns the scaled color components to their corresponding accumulation buffer.

ADD - same as MULT except that value is added to each of the color components.

To implement the accumulation buffer it was decided to store the results as a signed 2.14 fixed-point number. Therefore, to store and process the four channels the accumulation needed to take place over two operations (2 channels per accumulation buffer component, 4 channels per buffer).

Tag	Requirements
FBBuffer*0*	Set to the read buffers parameters
FBBuffer*2*	Set to the accumulation buffer 0 parameters
FBBuffer*3*	Set to the accumulation buffer 1 parameters
FBBaseAddr*0*	Set to the read buffer base address.
FBBaseAddr*2*	Set to accumulation buffer 0 base address.
FBBaseAddr3	Set to accumulation buffer 1 base address.
FBBufferEnables	Enable buffer *0* only.
DrawRectangle2D	Set up to rasterise the entire window region

Additionally, the pixel address unit was loaded with the following program:

```
SendSourceAddr (buf0, puReg0);              // colour buffer read into reg 0
SendDestAddrAndTile(buf2, puReg1, First);   // first two channels from buffer into reg 1
SendDestAddrAndTile(buf3, puReg1, Last);    // last two channels from buffer into reg 1
```

The pixel unit was loaded with the following program. This program has two entry points as the pixel address program invokes this twice. On the invocation, the “first” program entry point is called. This entry point, call the accumulation function to process the first two color channels, then the next two channels saved away in local registers for the next invocation. On the second invocation, the “last” entry point is called this entry point processes the final two channels that have previously been stored away.

On each invocation of the program, 2 color channels are processed. Additionally, multi-byte multiplications and additions are used so as to keep the accuracy of the final calculations. Here is an example of the program to perform the accumulation operation.

```
first:
        W[0] = PassA(P[0]);
        W[1] = PassA(P[1]) Call(@Acc); // process the first 2 channels

        W[0] = PassA(P[2]);
        W[1] = PassA(P[3]) Done;                // move the data for the next 2 channels

last:
        PassA(A[8]) Call(@Acc);                 // process the saved data
        PassA(A[8]) Done;

Acc:
    // scale and accumulate first colour component
    // calculate the first two bytes
    E = Global[ACCUM_SCALE_HIGH]   W[4] = MultU(A[0], E);

    E = Global[ACCUM_SCALE_HIGH]   W[5] = MultL(A[0], E);  // upper byte
    E = Global[ACCUM_SCALE_LOW]    W[6] = MultU(A[0], E);  // upper byte

    // propagate the carry information
        W[5] = Add(A[5],B[6]);
```

```
E = 0     W[4] = AddC(A[4], E);

// scale to normalise fixed point format
E = 2     W[4] = MultL(A[4], E);          // lower byte
E = 2     W[6] = MultU(A[5], E);          // upper byte
E = 2     W[5] = MultL(A[5], E);          // upper byte

      W[4] = Add(A[4], B[6]);

// calc next byte
E = Global[ACCUM_SCALE_LOW]
      W[7] = MultL(A[0], E);              // upper byte


// strip off top bits
E = 2     W[7] = MultU(A[7], E);


// factor in carry information
      W[5] = Add(A[5],B[7]);
E = 0     W[4] = AddC(A[4], E);


// Add into component in Accumulation buffer and write out
      C[0] = Add(P[4], B[5]);
      C[1] = AddC(P[5], B[4]);

// scale and accumulate second colour component
E = Global[ACCUM_SCALE_HIGH]
      W[4] = MultU(A[1], E);

E = Global[ACCUM_SCALE_HIGH]
      W[5] = MultL(A[1], E);              // upper byte

E = Global[ACCUM_SCALE_LOW]
      W[6] = MultU(A[1], E);              // upper byte


// propagate the carry information
      W[5] = Add(A[5],B[6]);
E = 0     W[4] = AddC(A[4], E);

// scale to normalise fixed point format
E = 2     W[4] = MultL(A[4], E);          // lower byte
E = 2     W[6] = MultU(A[5], E);          // upper byte
E = 2     W[5] = MultL(A[5], E);          // upper byte

      W[4] = Add(A[4], B[6]);
```

```
// calc next byte
E = Global[ACCUM_SCALE_LOW]
      W[7] = MultL(A[1], E);                    // upper byte


// strip off top bits
E = 2      W[7] = MultU(A[7], E);

// factor in carry information
      W[5] = Add(A[5],B[7]);
E = 0      W[4] = AddC(A[4], E);

// Add into component in Accumulation buffer and write out
      C[2] = Add(P[6], B[5]);
      C[3] = AddC(P[7], B[4]) Return;
```

The other accumulation buffer operations rely on similar principles. The multi-byte mathematics is common for all the operations. For example, the LOAD operation is an ACCUM operation without the addition to the buffer.

8.8 2D Operations (blits, pattern fills, fonts, pixel depth conversions, 2D logic ops.)

This section covers in more detail the most common operations required for 2D rendering; for example, rectangular color fills, rectangular pattern fills, text font rendering, screen to screen copies, bitmap depth conversion operations, and the various logical operations expected by GUIs such as X11 and Microsoft Windows.

8.8.1 Simple Solid Color Operations

The simplest 2D rendering operation is a solid color fill. This uses the most basic Pixel Unit and Pixel Address Unit programs to fill destination tiles with a color specified in the **PixelGlobal0** register. Here is a standard Pixel Address Unit program which will load destination tiles and send them to the Pixel Unit for processing.

```
Program(SimpleDestinationReadProgram, 0 )
      SendDestAddrAndTile( buf0, puReg0, Only);
```

This program takes the destination addresses from the Rasteriser, loads the tiles from Buffer0 memory if required, and passes them to the Pixel Unit for processing. This program can be used for almost all 2D rendering operations which use only the screen and a solid color or color pattern; you simply configure Buffer0, load the appropriate Pixel Unit program to perform the processing and, if destination tiles must be loaded for a logical operation, enable destination reads in the **FBBufferReadEnables** register.

The byte plane tiles will arrive at the Pixel Unit in registers P[0] to P[3], depending on the pixel depth settings in **FBBuffer0**; for example, at sixteen bits per pixel P[0] will hold the low byte of each pixel and P[1] the high byte. If you are rendering 8:8:8 RGB in a thirty-two bit framebuffer you can configure **FBBuffer0** to either load all four tiles (and then ignore the fourth tile in the Pixel Unit program) or only load the three tiles which store color data.

Loading three tiles requires less processing, but loading four uses the framebuffer memory interface more efficiently; you may need to benchmark both alternatives to determine the best choice for your application.

Here's a typical solid fill program at eight bits per pixel:

```
Program(SolidFillP8, 0x00)
    E = Global[0]       C[0]= PassA(E) Done;
```

This program loads the low byte of the color from the **PixelGlobal0** register and writes it to every pixel of the destination tile. It can readily be expanded to greater pixel depths by writing to more cache tiles. For example, a thirty-two bit four-tile version would be:

```
Program(SolidFillP32, 0x00)
    E = Global[0]       C[0] = PassA(E);
    E = Global[1]       C[1] = PassA(E);
    E = Global[2]       C[2] = PassA(E);
    E = Global[3]       C[3]= PassA(E) Done;
```

This program reads each byte from the **PixelGlobal0** register and writes them sequentially to the byte planes of the destination.

Pseudo-code to render a solid filled rectangle of size (w, h) pixels at screen coordinates (x, y):

```
LoadPixelAddressProgram (SimpleDestinationReadProgram);
LoadPixelUnitProgram (SolidFillProgram);
PixelGlobal0 = ForegroundColor;
FBBaseAddr0 = TileAddressOfDestination; // Set up Buffer0 in the Pixel Address Unit
FBBuffer0 = FBBufferForDestination;
RectanglePosition = (y << 16 | x);
DrawRectangle2D = (0xE000 | (h << 16 | w));
```

The simple solid fill program can easily be extended to support logical operations with a solid color by performing the operation while writing to the destination tile. An example (dest XOR color) thirty-two bit three-tile program would be:

```
Program(SolidFillDPx8, 0x0)
    E = Global[0]       C[0] = Xor(P[0], E);
    E = Global[1]       C[1] = Xor(P[1], E);
    E = Global[2]       C[1] = Xor(P[2], E) Done;
```

More complicated operations between the destination and a solid color may require temporary registers to store intermediate results when the operation cannot be accomplished with one instruction. For example, NOT (dest OR ((NOT color) AND dest) at sixteen bits per pixel:

```
Program (SolidROPBizarre, 0x0)
    E = Global[0]       W[0] = And(P[0], ~E);
    E = Global[1]       W[1] = And(P[1], ~E);
        C[0] = ~Or(P[0], B[0]);
```

```
C[1] = ~Or(P[1], B[1]) Done;
```

This program stores the intermediate results in the Pixel Unit's W registers, which are then accessed in the later instructions through their B register alias to complete the operation and write to the destination. As you can see, relatively complex logical operations can be implemented in this way by a small program.

Pseudo-code for a logical operation on a rectangle of size (w, h) pixels at screen coordinates (x, y):

```
LoadPixelAddressProgram (SimpleDestinationReadProgram);
LoadPixelUnitProgram (SolidLogicalOperationProgram);
PixelGlobal0 = ForegroundColor;
FBBaseAddr0 = TileAddressOfDestination; // Set up Buffer0 in the Pixel Address Unit
FBBuffer0 = FBBufferForDestination;
FBBufferReadEnables |= 1;                    // Enable reads for Buffer0
RectanglePosition = (y << 16 | x);
DrawRectangle2D = (0xE000 | (h << 16 | w));
```

These pseudo-code examples assume that you reload the programs for every rendering operation. Since the solid fill routines are so small and very common in many applications, you will probably find that you get the best performance by keeping them loaded into the Pixel Address Unit and Pixel Unit at fixed addresses so that you only need to set the program address in the **PixelMode** and **FBMode** registers to enable them rather than download the program for every solid fill.

8.8.2 Color Pattern Operations

We can render an 8x8 tile-aligned color pattern using the **UserFragData** registers; the Pixel Address Unit will continue to use the SimpleDestinationRead program, with different Pixel Unit programs to perform the pattern fill.

Each of the sixty-four **UserFragData** registers holds the color value (eight to thirty-two bits) for one pixel in a tile, and these can be read by the Pixel Unit. **UserFragData0** to 7 specify the colors for the first row of pixels in the 8x8 pattern, **UserFragData8** to 15 the second row, and so on.

The UserFragData register for each pixel is accessed through the Pixel Unit Fragment registers; programs are identical to solid fills, except that we read the color values from the Fragment registers rather than the Global registers. A Pixel Unit program to perform a solid fill with a color pattern at 8bpp:

```
Program(PatternFillP8, 0x00)
    E = Fragment[0]    C[0]= PassA(E) Done;
```

This program reads the low byte of the Fragment register for each pixel, and writes it to the tile. As usual, it can easily be expanded to other color depths:

```
Program(PatternFillP16, 0x00)
    E = Fragment[0]    C[0]= PassA(E);
    E = Fragment[1]    C[1]= PassA(E) Done;
```

The pseudo-code for performing a color pattern fill is identical to solid fills, other than loading the pattern fill program into the Pixel Unit and writing the color pattern to the **UserFragData** registers. One slight complication is that the pattern must be written to the registers as a thirty-two bit dword per pixel regardless of the destination color depth, whereas on the host it may well be stored as a packed eight or sixteen bit bitmap; in that case you will need to extract the color of each individual pixel from the pattern bitmap.

With a sixteen bit brush on IA32 processors you can use MMX **punpck** instructions to efficiently load four pixels from the brush as one qword and unpack them into four dwords ready to write to a DMA buffer. At eight bits, or on processors which lack such instructions, unpacking will be more complicated and implementation is left to the reader; note, however, that you do not need to zero the unused bits of the color values, as the Pixel Unit program will never access them.

Logical operations can be implemented as for solid fills. For example, a program to AND the thirty-two bit destination with the color pattern:

```
Program(PatternDPa32, 0x0)
    E = Fragment[0]    C[0]= And(P[0], E);
    E = Fragment[1]    C[1]= And(P[1], E);
    E = Fragment[2]    C[2]= And(P[2], E) Done;
```

If a 2D logical operation needs to mix a pattern with a transfer from host memory to the framebuffer (i.e. a download) then the above method won't work, because the download needs to use the **UserFragData** data registers, a way around this is to download the pattern into the framebuffer and then to perform the download and combine the pattern with a blt.

To do this you will need a more sophisticated pixel address program:

```
Program( pixelAddressFBBrushProgram, 0 )
    SendDestAddr( buf0, puReg0 );
    Add( r0, A0, tileX );
    Add( r1, A1, tileY );
    LoadXY( r0, r1 );
    Copy( r3, A3 );
    LoadXYMask( A2, r3 );
    SendSourceAddrAndTile( buf4, puReg1, Only );
```

The A0 and A1 registers specify the (x,y) offset into the pattern to begin rendering from and A2 and A3 contain the XYmask which specifies that the pixel address unit should wrap back to the start of the pattern when it reaches the end. These parameters can be passed to the pixel unit program via the FBAddrInfo0 and FBAddrInfo1 registers.

Here is an 8-bit pixel unit program that takes a pattern in the framebuffer and XOR's it with the destination in the framebuffer, then OR's it with the source download data before writing it back to the destination in framebuffer.

```
Program(SDPxo8, 0x0)
        W[12]= PassA(P[4]) ;
        W[4]= Xor(P[0],B[12]) ;
  E=Fragment[0]    C[0]= Or(E,B[4]) Done;
```

To set-up the pattern we do the following (over and above the normal download code):

```
LoadPixelAddressProgram (pixelAddressFBBrushProgram);
LoadPixelUnitProgram (SDPxo8);

FBAddrInfo0= (xPatOff) | (yPatOff<<16);      // Parameter A0 and A1 – (x,y) offset
FBAddrInfo1= (1) | (1<<16);                  // Parameters A2 and A3 - xyMask
FBBaseAddrGlobal= TileAddressOfPattern;      // Set up pattern address in Pix Address Unit
FBBufferGlobal= FBBufferForPattern;          // Set up Buffer4 in the Pix Address Unit
```

The download is then rendered in the normal fashion and the download data is written to the **PixelData** register.

8.8.3 Monochrome Pattern Fills

You can fill areas with a monochrome pattern using the **AreaStipple** registers for transparent patterns (fill foreground pixels, leaving background pixels untouched) or the **PixelMask** register for opaque patterns.

Again, the Pixel Address Unit will run the SimpleDestinationRead program. For transparent patterns we can use the solid fill Pixel Unit programs, as the **AreaStipple** tests will produce a Tile Mask which the Pixel Unit will use to determine which pixels to update; the solid fill program will attempt to replace all pixels in the tile with the color specified in **PixelGlobal0**, but writes to memory are gated by the Tile Mask so that only the foreground pixels will actually be updated.

For opaque patterns you must create new Pixel Unit programs which render the foreground or background color based on the Pixel Mask. These programs are very similar to those used for monochrome bitmap downloads: a suitable program to perform a solid opaque pattern fill at sixteen bits:

```
    Program(MonoOpaqueFillP16, 0x0)
> Program(MonoOpaqueFillP16, 0x0)
>       E = Global[0]     C[0] = PassA(E) Flag=PM;
>       E = Global[4]     C[0]&= PassA(E);
>       E = Global[1]     C[1] = PassA(E);
>       E = Global[5]     C[1]& = PassA(E) Done;
```

This program takes the foreground color in the **PixelGlobal0** register and the background color in **PixelGlobal1**. The first instruction fills the first byte plane of the destination tile with the low byte of the color in **PixelGlobal0**, and copies the Pixel Mask to the Pixel Unit Flags register. The second instruction replaces the pixels specified by the Pixel Mask with the low byte of the color specified in **PixelGlobal1**. The remaining two instructions perform the same operation with the second byte of the color and the second byte plane of the tile, giving a sixteen bit result.

Whichever method you use, you must load a tile-aligned sixty-four bit 8x8 monochrome pattern into the appropriate registers. If your pattern is stored on the host in tile-aligned form, then this is simply a matter of reading the sixty-four bit pattern from memory and writing it to the registers. Otherwise you must rotate the pattern appropriately so that the top left pixel of the pattern is aligned with the top left pixel of the tiles; implementation is left to the reader, but if you must rotate often you may wish to cache a copy of the pattern

for each of the possible x rotations and only perform rotation in y at runtime. This is a good compromise between performance and storage as y rotation can be performed easily by any CPU with a sixty-four bit rotate instruction.

Assuming you have already set up Buffer0 to point to your destination, here is the pseudo-code for a monochrome pattern fill of size (w, h) pixels at screen coordinates (x, y)

```
LoadPixelAddressProgram (SimpleDestinationRead);
RotatePattern (Pattern64, x, y);
If (TransparentPattern) {
    PixelGlobal0 = ForegroundColor;
    LoadPixelUnitProgram (SolidFillProgram);
    FBBufferReadEnables |= 1; // Must always enable destination reads here.
    AreaStipple0 =  low dword of Pattern64;
    AreaStipple1 = high dword of Pattern64;
    RasterMode |= (AreaStippleEnable | AreaStipple8x8);
}
else {
    PixelGlobal0 = ForegroundColor;
    PixelGlobal1 = BackgroundColor;
    LoadPixelUnitProgram (MonoOpaqueFillProgram);
    PixelMask64 = Pattern64;
    // No destination reads required here for solid fill, only for a logical operation.
}

RectanglePosition = (y << 16 | x);
DrawRectangle2D = (0xE000 | (h << 16 | w));
```

These programs can readily be extended to support logical operations. For transparent brushes you can use the solid brush logical operation programs, but for opaque brushes you must expand the monochrome brush into a temporary register and then perform the logical operation between that register and the destination. An example ((dest XOR NOT pattern) OR pattern) program at 8bpp:

```
Program(MonoOpaqueROPPDPnxo8, 0)
                    E = Global[0]                W[0] = PassA(E) Flag=PM;
                    E = Global[4]                W[0.1] = PassA(E);
                                W[4] = Xor(P[0], ~B[0]);
                                C[0] = Or(A[4], B[0]) Done;
```

This program first expands the monochrome pattern in the pixel mask to an eight bit color pattern in the W[0] register. It then performs the logical operation between the destination and the expanded pattern, using the W[4] register to store intermediate results.

8.8.4 Screen To Screen Copies (BitBlt)

Screen to screen (or, more precisely, video memory to video memory) copies require a new Pixel Address Unit program which can read tiles from two buffers: Buffer0 for the destination, and BufferGlobal for the source. Here is a typical copy program:

```
Program( SimpleCopyProgram, 0x0 )
```

```
Add( r0, A0, tileX );
Add( r1, A1, tileY );
LoadXY( r0, r1 );
SendSourceAddr( buf4, puReg1 );
LoadXYFromTile();
SendDestAddrAndTile( buf0, puReg0, Only );
```

This program requires that you configure the source and destination buffers using the **FBBaseAddr0**, **FBBaseAddrGlobal**, **FBBuffer0** and **FBBufferGlobal** registers, and pass the relative x and y offset between the two buffers in the **FBAddrInfo0** register; using BufferGlobal for the source allows the same program to support writes to up to four buffers if required for 3D page-flipping. Note that the destination tile must be sent *after* the source tile when the source and destination may overlap.

The first two instructions in the program use the x and y offset values from FBAddrInfo0 to calculate the x and y coordinates of the source tile from the x and y coordinates of the destination tile. The next two instructions load those coordinates and then send the source tile to the Pixel Unit. The final two instructions load the destination tile coordinates and send that tile to the Pixel Unit.

As with fill programs, the destination tile byte planes will arrive at the Pixel Unit in registers P[0] to P[3]. The source tile byte planes will arrive in registers P[4] to P[7]. The Pixel Unit programs must combine the source and destination with any required logical operation and then write the updated destination out to memory. A program to perform a straightforward copy at sixteen bits:

```
Program(Blt16, 0x0)
        C[0] = PassA(P[4]);
        C[1] = PassA(P[5]) Done;
```

This program simply copies the source tiles to the destination.

Pseudo-code to perform a screen to screen copy of rectangle size (w, h) pixels from coordinates (xs, ys) to (xd, yd):

```
LoadPixelAddressProgram(SimpleCopyProgram);
LoadPixelProgram(BltProgram);
FBBaseAddr0 = TileAddressOfDestination;
        // Set up Buffer0 in the Pixel Address Unit
FBBuffer0 = FBBufferForDestination;
FBBaseAddrGlobal = TileAddressOfSource;
FBBufferGlobal = FBBufferForSource;
        // No need to set FBBufferReadEnables for straightforward copy
FBAddrInfo0 = (ys – yd) << 16 | (xs – xd);
        // Pass coordinate deltas to Pixel Address Unit
        // May need to mask out sign bits above if x and y coords are 32-bit.
RectanglePosition = (yd << 16 | xd);
DrawRectangle2D = (0xE000 | (h << 16 | w));
```

This code first sets up the Pixel Unit and Pixel Address Unit programs for the copy. Then it configures the source and destination buffers, and passes the offset between the source and destination coordinates to the Pixel Address Unit so that the program can calculate source tiles from destination tiles. Finally it sets the destination position and performs the copy.

As usual, the programs can easily be extended to support logical operations. A program to perform a (src AND dest) operation at sixteen bits:

```
Program(BltSDa16, 0x0)
        C[0] = And(P[0], P[4]);
        C[1] = And(P[1], P[5]) Done;
```

To perform such an operation you would load this program into the Pixel Unit and enable destination reads in the **FBBufferReadEnables** register.

In some cases you may need to perform logical operations between the source, destination and a pattern (solid, color or monochrome). Programs to perform such operations can be written using similar techniques to the pattern fill programs. For example, a program to perform (src OR (NOT dest AND pattern)) with an 8x8 opaque monochrome pattern at eight bits:

```
Program(OpaqueMonoBltSDPano8, 0)
        E = Global[0]           W[0] = PassA(E) Flag=PM;
        E = Global[1] W[0.1]= PassA(E);
                      W[0] = ~And(P[0], B[0]);
                      C[0] = Or(A[0], P[4]) Done;
```

This program first expands the monochrome pattern specified by the Pixel Mask into the W[0] register, then performs the logical operation with the source, destination and the expanded pattern. A program to perform (dest XOR (src OR pattern)) with an 8x8 color pattern at eight bits:

```
Program(ColorBltDSPox8, 0)
        E = Fragment[0]         W[0] = Or(P[4], E);
                      C[0] = Xor(A[0], P[0]) Done;
```

This program performs the logical operation between the source, destination and the color pattern specified in the **UserFragData** registers.

The main exceptions are operations requiring a transparent monochrome pattern. As the generated Tile Mask gates all writes to the destination you cannot use the **AreaStipple** registers for operations which also require a source. Instead you must load the pattern into the Pixel Mask as for opaque patterns, and then explicitly process the pattern yourself using the Pixel Mask. A program to perform (pattern AND (src AND dest)) with an 8x8 transparent monochrome pattern at eight bits:

```
Program(TransparentMonoBltPSDaa8,  0)
        W[0] = And(P[4], P[0]) Flag=PM;
        C[0] = PassA(A[0]);
        E = Global[0] C[0] &= And(A[0], E) Done;
```

This program first ANDs the source and destination byte tiles, and copies the pixel mask to the flags. It stores the result in the W[0] register and copies that to the destination. It then ANDs the W[0] register with the foreground color specified in the **PixelGlobal0** register, using the Pixel Mask to perform the transparent update of the destination.

8.8.5 Text Font Rendering

You can render fonts directly from host memory using the **Bitmask** register; such operations are covered in the Bitmask Operations section. For glyphs whose width or height is greater than 127 pixels you must render from host memory, however, for the best performance for smaller glyphs you should minimize the amount of data you download by caching font glyphs in offscreen memory and rendering using the **GlyphAddr** and **RenderGlyph** registers.

Cached font glyphs are stored as monochrome bitmaps in individual bit-planes (zero to thirty-one) of offscreen memory tiles, and the font cache must always be thirty two bit planes; the font glyph to render is specified in the **GlyphAddr** register by the tile address of the first tile of the glyph and the bitplane number. The initial glyph position is set using the **GlyphPosition** register, then the glyph is rendered using the **RenderGlyph** register, which specifies the size of the glyph and the offset to the glyph; this allows you to render multiple glyphs without resetting the **GlyphPosition** register for each one.

Note that the glyph rendering operations internally utilise the **FBBaseAddrGlobal**, **FBBufferGlobal**, **PixelGlobal7** and **FBAddrInfo3** registers; if you have cached values in these registers then you must reload them after the rendering operation.

However, before you can render a glyph from offscreen memory you must first download it. For best performance you should download successive glyphs from the same font to each bit plane in turn rather than completely fill one bit plane of your offscreen cache area before downloading to the next; in most cases this will significantly reduce memory bandwidth requirements for rendering operations as thirty-two glyphs can be read from memory simultaneously.

To download the glyph you can use the simple Pixel Adress Unit program, but need a new Pixel Unit program:

```
Program(FontDownloadProgram, 0x0)
        E = Global[28] W[0] = And( P[0], ~E ) Flag = PM;
        E = Global[28] W[1] =  Or( A[0],  E );
        C[0] = SelectB( A[0], B[1] ) Done;
```

This program uses the low byte of the **PixelGlobal7** register as a mask to determine which plane to write the font glyph into. The glyph itself is downloaded by writing to the **Bitmask** register, and setting the **RasterMode** to generate a Pixel Mask from the bitmask data.

Pseudo-code to download a glyph of size (w, h) pixels to bit-plane p at tile address a:

```
LoadPixelUnitProgram(FontDownloadProgram);
tw = (w + 7) >> 3;                      // Calculate tile width
FBBuffer0 = (tw << 2) | (1 | (4 << 13) | (1 << 17) | (1 << 21));
FBBaseAddr0 = a + (p >> 3);             // Offset start tile to correct plane
```

```
PixelGlobal7 = (1 << (p & 7));          // And set plane mask for that tile
RasterMode |= GeneratePixelMask;        // Generate pixel mask from bitmask data
RectanglePosition = (0 << 16 | 0);
DrawRectangle2D = (0xE000 | SyncOnBitmask | PackedBitmask);
Now write the glyph data to the Bitmask register.
CacheControl = (InvalidatePixel | FlushPixel);
```

This code sets the PackedBitmask bit in DrawRectangle2D. Unlike other bitmask operations (e.g. downloading a monochrome bitmap) where any excess data at the end of a scanline is thrown away, this retains the data for the next scanline, allowing you to download glyph data which is packed to a byte boundary rather than a dword boundary and avoiding unpacking the data from bytes into dwords. If your data is already packed to dword boundaries then you will not need to set the *PackedBitmask* bit.

Now that the glyph is downloaded, you can render it. This requires another Pixel Address Unit program:

```
Program(TextRenderAddressProg, 0x0 )
    SendDestAddr( buf0, puReg0 );
    Add( r0, A6, tileX );
    Add( r1, A7, tileY );
    LoadXY( r0, r1 );
    SendSourceAddrAndTile( buf4, puReg1, Only );
```

Here's an example Pixel Unit program to draw transparent text on a sixteen-bit destination:

```
Program(DrawTransparentTextProgram16, 0x0)
    E = Global[31]          Flag = Bit(P[4], E);
    E = Global[0]           C[0]&= PassA(E);
    E = Global[1] C[1]&= PassA(E) Done;
```

This program can readily be extended to support other color depths. Pseudo-code to render transparent Text starting at location (x, y) in Buffer0:

```
LoadPixelAddressProgram(TextRenderAddressProgram);
LoadPixelUnitProgram(DrawTransparentTextProgram);
FBBufferReadEnables |= 1;       // Set destination reads.
PixelGlobal0 = TextColor;
GlyphPosition = (y << 16) | x;
advanceX = advanceY = 0;
For each glyph:
        GlyphAddr = (glyphTileAddr << 5) | glyphBitPlane;
        RenderGlyph = glyphWidth | (glyphHeight << 6) | (advanceX << 14) |
                (advanceY << 23);
        Update advanceX and advanceY appropriately;
        if (advanceX > 255 || advanceX < -255 || advanceY > 255 || advanceY < -255)
                GlyphPosition = (y + advanceY) << 16 | (x + advanceX);
                advanceX = advanceY = 0;
```

The Pixel Unit program can also be extended to support logical operations by performing the operation between the text color and destination in a temporary register, and then merging that into the destination using the pixel mask. For example, to render XOR text at eight bits:

```
Program(DrawXORTransparentTextProgram8, 0x0)
    E = Global[31]          Flag = Bit(P[4], E);
    E = Global[0]           W[0] = Xor(P[0], E);
        C[0]&= PassA(A[0]) Done;
```

You can also render opaque text by first filling the destination with the background color and then filling it again with the text color, using the Pixel Mask to ensure that only the text foreground pixels are updated:

```
Program(DrawOpaqueTextProgram8, 0x0)
    E = Global[31]          Flag = Bit(P[4], E)
    E = Global[4]           C[0] = PassA(E)
    E = Global[0] C[0]&= PassA(E) Done;
```

For logical operations with opaque text you must first render the text to a temporary register, and then perform the operation between that temporary register and the destination:

```
Program(DrawXOROpaqueTextProgram8, 0x0)
    E = Global[31]          Flag = Bit(P[4], E);
    E = Global[4]           W[0] = PassA(E);
    E = Global[0] W[0.1]= PassA(E);
        C[0] = Xor(P[0], B[0]) Done;
```

Both of these programs assume the text color is loaded in the **PixelGlobal0** register and the background color in **PixelGlobal1**.

8.8.6 Bitmap Depth Conversion

You can perform bitmap depth conversion operations by configuring the chip for a copy operation between two buffers of different depths or color formats, and running an appropriate Pixel Unit program to perform the color conversion. Here is an example program for 16->32 bit color conversion, which uses multiply instructions to simulate bitwise shift operations:

```
Program(ColorConvert16To32, 0x0)
    E = 8       W[0] = MultL(P[4], E); // W[0] = blue
    E = 1       W[1] = And(P[4], E); // W[1] = bottom bit of blue
    E = 7       W[1] = MultL(A[1], E); // replicate bottom bit into bottom three
                C[0] = Or(A[0], B[1]); // And merge them together
    E = 32      W[0] = MultU(P[4], E); // W[0] = middle three green bits.
    E = 32      W[1] = MultL(P[5], E); // W[1] = top three green bits.
                W[0] = Or(A[0], B[1]); // W[0] = top six bits of green
    E = 32      W[1] = And(P[4], E); // W[1] = bottom bit of green
```

```
E = 24          W[1] = MultU(A[1], E); // W[1] = replicate into bottom two bits of
green
                C[1] = Or(A[0], B[1]); // Merge green together
E = 0xF8        W[0] = And (P[5], E); // W[0] = top five bits of red
E = 8           W[1] = And(P[5], E); // W[1] = bottom bit of red
E = 224         W[1] = MultU(A[1], E); // W[1] = replicate into bottom three red bits
                C[2] = Or(A[0], B[1]) Done; // And merge red bits together
```

Although the program requires many instructions, because each instruction operates on an entire tile this conversion will still be much faster than performing the conversion using the host CPU.

8.9 Video Operations and the DXVA Driver

DXVA is Microsoft's API to enable access to hardware accelerated MPEG decoding. There are several levels of DXVA acceleration ranging from raw bit-stream decompression to simple motion compensation. It is recommended that the reader has some knowledge of the MPEG-2 standard, especially section 7.6 which deals with motion compensation on which most of this section is based.

This section explains how a DXVA driver can be structured to implement motion compensation (mocomp) on P10. To a first approximation mocomp on P10 is glorified multi-texturing and since the details of programming P10 to perform multi-texturing are described elsewhere in this reference, this section will not give exact details of how the chip will be programmed. Rather this discussion will be at an intermediate level of decomposing the commands sent through the DDI (Device Driver Interface) into multi-texturing operations.

8.9.1 Video scaling (replication and pixel dropping)

Video scaling (either replication or pixel dropping) is implemented as a texturing operation in P10. However the obvious technique of drawing two textured triangles with the position and texture coordinates set appropriately will not work when the video operation is performed via the isochronous channel. There is only a rectangle rasteriser in the isochronous channel and none of the Vertex Shader or Setup units are available either. This means that the driver must set up the plane equations for the texturing operations manually. This manual setup technique is discussed in "Sub-Pixel Sampling & Interlacing" to which you are referred.

8.9.2 Using the Isochronous channel for video overlays

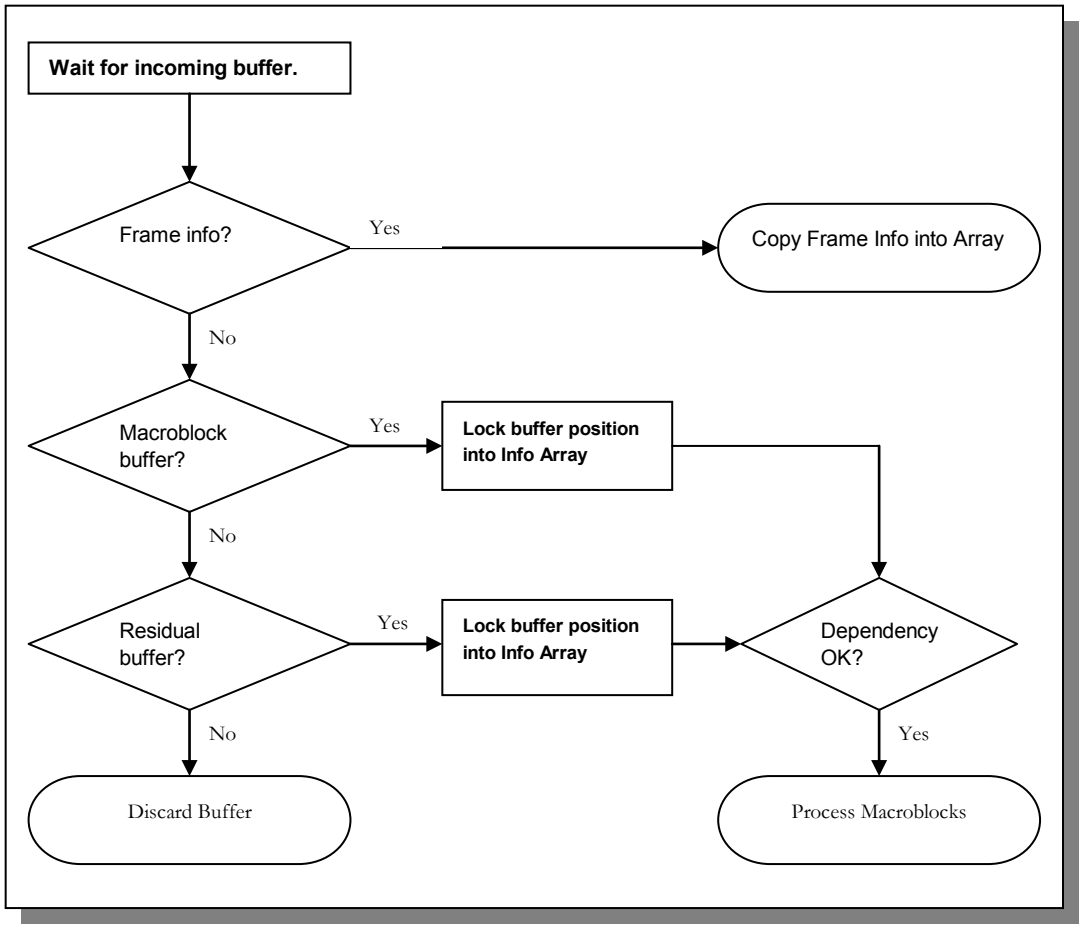
8.9.3 Probe & Locking

DXVA driver functionality is split into 2 main parts:

- Probe & Locking of the DXVA configuration, and
- The Main Function Loop.

This part of the driver handles configuration queries. It determines if the current probing or locking configuration is acceptable or not. If it isn't the function should reject the configuration and give back the configuration details that it didn't accept. Futher detail can be obtained from the Microsoft sample DXVA source.

8.9.4 Main Function Loop



The data in DXVA is passed around in buffers, which are contained in DirectDraw Surfaces. Once the probing and locking has been completed DXVA sends this data with

Fig 3.9-1 Driver Algorithm for Buffer Transfer

Begin & End Frames. The DDMOCOMPCALLBACKS documentation states that the Begin & End functions do not need to be called in pairs, but DXVA overrides that and

explicitly specifies that Begins & Ends will be paired. As a result, when decompressing Frame Structured pictures only one Begin & End frame pair is passed, but in Field Structured pictures there will be 2 pairs, one for the Top and one for the Bottom Field. Once a Begin Frame has been called the driver follows the functionality shown above.

The flow chart above shows the functionality of the driver when handling DXVA function 1. This works as follows:

In the Begin Frame call we get a pointer to a surface and an index to go with it. The index and surface pointer go into an array where we keep the current information about all the surfaces we have handled so we can form predictions from them. The index is then also stored to indicate the current frame we are writing to. Then the driver waits for the buffers to be passed into it via the **RenderMoCompFrame** callback. This acts as a command parser which copies the buffer (or its pointer) into the array so we can decode the frame later. Once all the buffer information required is received the driver automatically begin to procress the macroblocks and residual data.

Before each frame is processed the dependencies of the frame (reference frames, buffer information & output location) are checked to make sure that they are valid. Once this done the **ProcessMacroBlock** function is called which goes to process all the data. The flow diagram for the function is shown below:

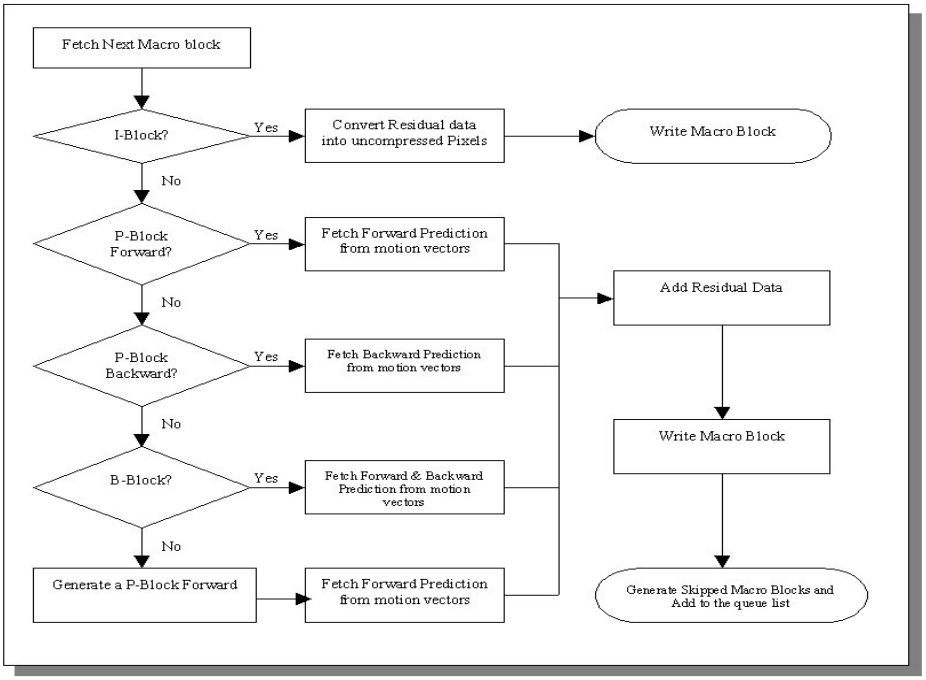


Fig 3.9-2: Macro Block Processing Algorithm

The main issue to notice here is that this algorithm does not care about which kind of frame type we are handling, All the dependencies must be true for us to reach this point so we only care about each individual macro block. Each block is matched to what kind it is, the information required to process it is fetched and then the macro block is executed.

If the macro block has no known type MPEG-2 dictates we generate a P-Block from it. It should be noted that the left hand side of the flow chart is done in the driver while the right hand side of the flow chart is done by the hardware once it is programmed properly.

8.9.5 Implementation

8.9.5.1 Data formats, Data flow & Programs: Surface formats & General Data flow

Some assumptions have been made to simplify the design and implementation. First of all, all buffers will be held in system memory apart from the overlay surfaces, these in turn will be held in YV12 planar format. This format simplifies the motion compensation processes and reduces the number of errors introduced since the compression side is done on a planar basis as well. This format also provides the opportunity to implement the NV12 format, which Microsoft hints to be the preferred format in future specifications.

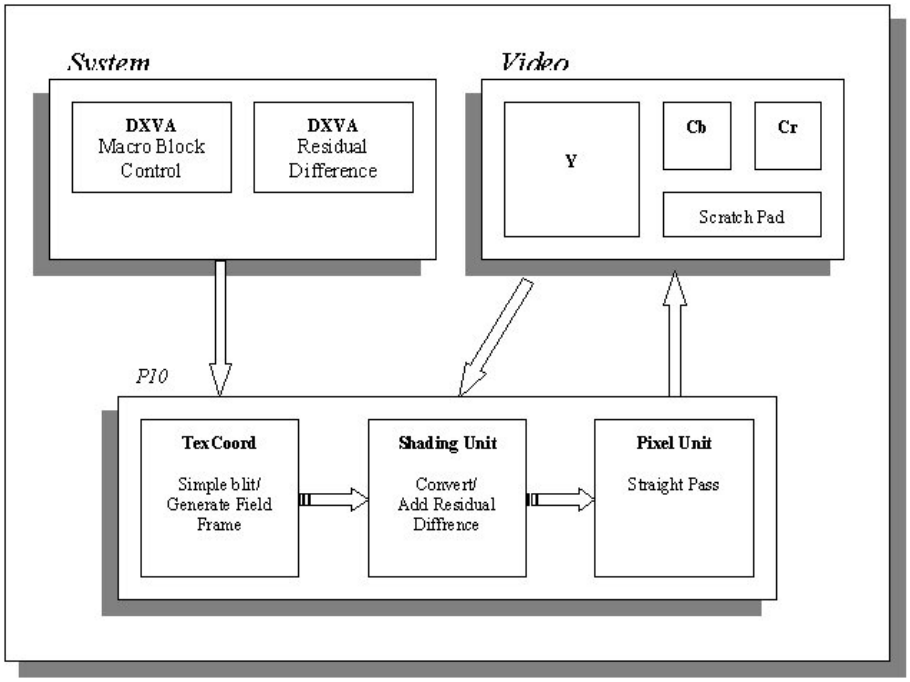


Fig 8.9-3 Simplified data flow through P10

From the simplified diagram above it can be seen how all the DXVA buffers are kept in system memory in order to save time in the processing of the macroblock control commands. When needed, the residual data is DMAed into a “scratch pad” area of video memory which has previously been reserved by the driver. When the residual data is read into the shader (as part of a multi-texture operation) it is converted from signed to unsigned for the I type blocks. When dealing with reference frames all the data is converted in the shader from unsigned to signed, processed and then transformed back

into unsigned on the output. This is because all the residual data is held in signed format. Ideally the residual data would all be 16-bit signed integers but since P10 can't handle such a data type variable 8-bit residual differences with overflow blocks will be used instead. It can also be seen from the diagram that the TexCoord unit and the Pixel Unit are both left doing simple operations.

Because of the planar surface formats and the macro-block structure of the data it is possible to process the mocomp requests 32-bits at a time instead of at the more obvious 8-bit granularity implied by the surface formats. The net result is that can handle 4 residual differences at a time instead of only one giving a substantial increase in mocomp "fill-rate".

8.9.5.2 Alpha Blending

The DXVA MPEG2_B profile requires that we support alpha blending. The format of the surface that is used to blend can either be a DPXD surface with Highlight and DCCMD command blocks, and Index Alpha & Colour surface or an AYUV surface. For simplicity of the implementation we have decided to use AYUV surface format as this will require less processing to be done by the chip and the driver. The formula used to alpha blend determined by Microsoft is as follows:

```
if( Alpha != 0)
{
Final Channel = (( Alpha + 1) * Source_Channel + (255 – Alpha) * Picture_Channel + 128) >>
            8;
}
else
{
Final Channel = Picture_Channel;
}
```

…where *Picture_Channel* is the decode image and *Source_Channel* is the alpha blending surface.

But also according to Microsoft this is only true for YUV 4:4:4 based formats. If a surface has some of its channels sub-sampled, as in the 4:2:0 profile used by the MPEG2 profile, only the second sample shall be used in the sub-sampled channel. This coupled with the YV12 surface format used in the final overlay surface poses several problems. However using a multi-pass, multi-sampling approach to alpha blending should solve these.

Using the multi-pass would mean that in the first pass (for Y) the equation would be unchanged. However when the second and third pass came for U & V P10 would have to be programmed to pick different pixels, in the same way that multi-sampled AA rendering does but instead we would drop 3 out of the 4 samples and only alpha blend with the last one.

8.9.5.3 Sub-Pixel Sampling & Interlacing

Sub-pixel sampling plays the major role in motion compensation in MPEG-2, simply because all motion vectors are in half-pixel units. This is accomplished in P10 is by using the texture filter units to interpolate between 2 pixel values.

The process of interlacing is also done in the Shading unit. This is accomplished by the use of plane equations which are set up to wrap around on a per-pixel basis. Normally the plane-equations are set up by upstream units in the chip to ramp up through a range of values (possibly clamping at the extreme ends of the range). However it is possible to set up the plane equations directly using the ColorPlaneStart/DXN/DXY tags in such a way as to cause the output of the plane equation evaluators to oscillate between 0 and 1on a per scanline basis. The oscillating value is then used to select one of two inputs giving rise to field interleaving. The globals in the TexCoord program allow the field select (top or bottom) to be changed cheaply i.e. avoiding a whole program reload when the fields change.

The following pseudo-code shows the TextureCoord and Shader unit programs that implement this scheme:

```
SetTexCoordGlobal(3, 1.0f/float(1 << 3));

SetTexCoordMode(eTexCoordModeEnable);
SetProgramMpeg(eTextureMap0);
      // This will multi texture anyway
SetShaderProgramMPEGInterleave(0, 1);

unsigned32 st = 0;
unsigned32 dx = 0;
unsigned32 dy = 0x200000;
      // Causes plane equations to oscillate in y

SendMessage(ColourPlaneStart0, st);
SendMessage(ColourPlaneDX0, dx);
SendMessage(ColourPlaneDY0, dy);
```

TexCoord Program

```
TexCoordInstr("PlaneGlobalBase(0 ,0 ) FloatToInt(DivResult)");

TexCoordInstr("W[0] = MAdd(One, X, One,Global0[0])");
TexCoordInstr("W[1] = MAdd(One, Global1[0], One,Y)");
TexCoordInstr("W[2] = MAdd(One, Global0[1], One,Y)");

TexCoordInstr("C[0] = Wrap(A[0], Global1[1])");
TexCoordInstr("C[1] = Wrap(A[1], Global1[1])");

TexCoordInstr("Command(FilterTexture, 0,0 , LoadShade,
            NoFB, Default) FloatToInt(DivResult)");
```

```
TexCoordInstr("C[0] = Wrap(A[0], Global1[1])");
TexCoordInstr("C[1] = Wrap(A[2], Global1[1])");

TexCoordInstr("Command(FilterTexture, 1,1 , LoadShade,
            NoFB, Default) FloatToInt(DivResult)",eTCDone);
```

Shader program

```
ShaderInstr("W[Red] = PassA(Plane[0])");
ShaderInstr("Flag = Sub(A[Red], Const[0], ==)");

ShaderInstr("W[Red  ] = PassA(T[" + STR(tRedB  ) + "])");
ShaderInstr("W[Green] = PassA(T[" + STR(tGreenB) + "])");
ShaderInstr("W[Blue ] = PassA(T[" + STR(tBlueB ) + "])");
ShaderInstr("W[Alpha] = PassA(T[" + STR(tAlphaB) + "])");

ShaderInstr("C[Red  ] = SelectA(T[" + STR(tRedA  ) + "], B[Red  ])");
ShaderInstr("C[Green] = SelectA(T[" + STR(tGreenA) + "], B[Green])");
ShaderInstr("C[Blue ] = SelectA(T[" + STR(tBlueA ) + "], B[Blue ])");
ShaderInstr("C[Alpha] = SelectA(T[" + STR(tAlphaA) + "], B[Alpha])",
```

This will greatly speed up processing since it only takes 2 instructions in the Shader instead of having to render different lines for different fields in completely separate render operations.

8.9.5.4 Clipping Macroblocks

Clipped macroblocks occur when a motion vector causes the reference macroblock to start outside the picture boundary. Detection of this will be done in the driver while generating the S &T coordinates for the textures. If it is found that the texture coordinates lie outside the reference picture the macro blocks that need processing will be copied to the scratch pad and the clipped area filled in with black. This will make sure that there are no unknown pixel values being generated by the motion compensation.

8.9.5.5 Frame structured Pictures

“I” Frames

This is the most straightforward operation in the whole DXVA driver. Each macro block will be split up into four 8x8 blocks. Then every block is rendered with 3 texture coordinates: one Y, one Cb & one Cr. The shader will only pass the pixel values to the pixel unit, where these will be converted from signed to unsigned. There are no motion vectors and the driver will ignore concealment vectors.

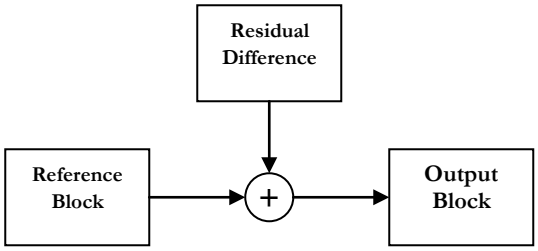
```
// Sample DRIVER Pseudo code for I frames

for(every macroblock in the picture)
{
    for(every block inside the macroblock)
    {
```

```
            DMA to scratch pad.
            Program Shader to convert from 8-bit signed
            to 8-bit unsigned.
            Calculate the blocks position in the final YV12
            Surface.
            Blit 8x8 8-bit block from scratch pad to Overlay
            Surface
        }
}
```

"P" Frame structured blocks

P-Frame blocks are the simplest form of motion compensation. A reference block offset by a motion vector is fetched, the residual differences are added and then the block is written to the output picture. The basic processing flow is as follows:



The reference block is generated from the motion vectors. These are used to calculate texture coordinates in the reference frame. These are then used to do a multi texture blit operation in which the shader will be used to add or subtract the residual difference. This will be done for every block that has a residual difference present in the macro block control command. If a block has no residual difference a simple blit operation will be used to copy the block from the reference picture to the output picture. The proposed driver pseudo code is as follows:

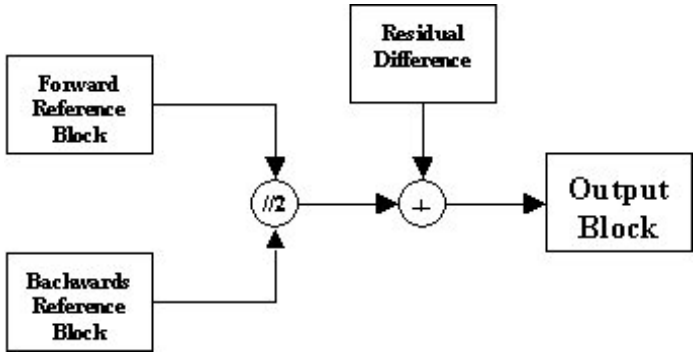
```
// Sample DRIVER Pseudo code for P frames
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 6; i++)
    {
```

```
            Calculate Reference block Tex Coords.
            if(HaveResdiffBlocks[i])
            {
            Change shader to add resdiff program
            Change TCU to multi textured blit program
            Calculate Residual Difference TexCoord location from offset_into_resdiff
            Blit 8x8 8-bit block
            offset_into_resdiff++
            }
            else
            {
                Change shader to blit program
                Change TCU to single textured blit program
                Blit 8x8 8-bit block
            }
        }
    }
```

“B” Frame structured blocks

B frame blocks are a step forward from P frame blocks. The processing is virtually the same apart from the fact that the reference block is composed by averaging 2 reference blocks, one in the forward reference frame and one in the backwards frame. The processing flow is as follows:



The reference blocks are generated from the motion vectors. These are used to calculate 2 sets of texture coordinates in the reference frames. These are then used to do a multi texture blit operation in which the shader will be used to average the incoming pixels and then add or subtract the residual difference. This will be done for every block that has a residual difference present in the macro block control command. If a block has no residual difference a simple blit operation will be used to copy the block from the reference picture to the output picture, but the averaging will still be done. The proposed driver pseudo code is as follows:

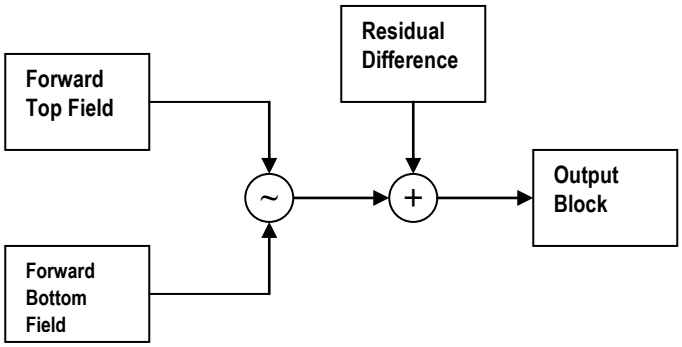
```
// Sample DRIVER Pseudo code for B frames
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 6; i++)
    {
        Calculate Forward Reference block TexCoord location
        Calculate Backward Reference block TexCoord location
        if(HaveResdiffBlocks[i])
```

```
            {
                Change shader to average and add resdiff program
                Change TCU to multi textured blit program
                Calculate Residual Difference TexCoord location
                                from offset_into_resdiff
                Blit 8x8 8-bit block
                offset_into_resdiff++
            }
            else
            {
                Change shader to average blit program
                Change TCU to multi textured blit program
                Blit 8x8 8-bit block
            }
        }
}
```

“P” Field structured blocks

This is a more complicated form of motion compensation but still relatively straight forward. As with P frame block the residual difference is added to reference block. But in this case the reference block is formed by interlacing 2 fields. The fields change on a line by line basis and can be formed from any field in the macro block pointed to by the motion vector, i.e. the top field of the reference block can be formed from the top field pointed by motion vector 1 while the bottom field can also be generated by the top field pointed to by motion vector 2. The process flow of the block is as follows:



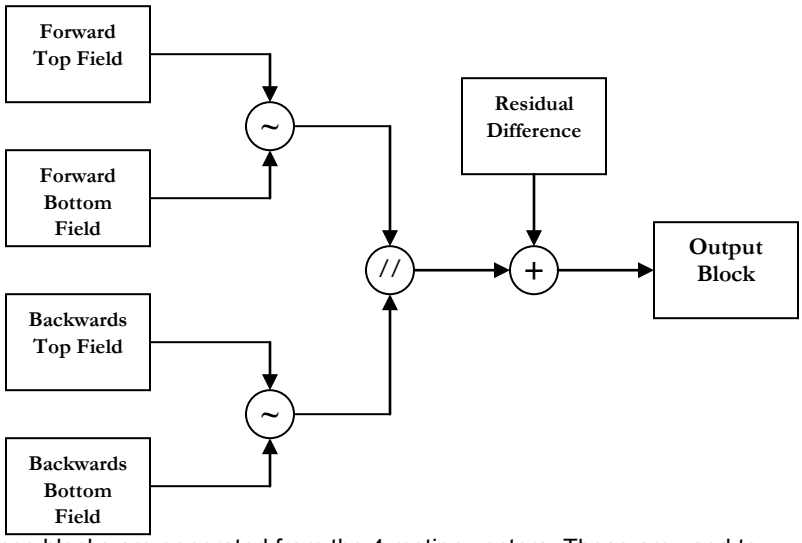
The reference blocks are generated from the motion vectors. These are used to calculate 2 sets of texture coordinates in the reference frames. These are then used to do a multi texture blit operation in which the shader will be used to interlace the incoming pixels and then add or subtract the residual difference. This will be done for every block that has a residual difference present in the macro block control command. If a block has no residual difference a simple blit operation will be used to copy the block from the reference picture to the output picture, but the interlacing will still be done. The proposed driver pseudo code is as follows:

```
// Sample DRIVER Pseudo code for P field blocks
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 6; i++)
    {
        Calculate Top Field Reference block TexCoord location
        Calculate Bottom Field Reference block TexCoord location
        if(HaveResdiffBlocks[i])
        {
            Change shader to interlace and add resdiff program
            Change TCU to multi textured blit program
            Calculate Residual Difference TexCoord location
                        from offset_into_resdiff
            Blit 8x8 8-bit block
            offset_into_resdiff++
        }
        else
        {
            Change shader to interlace blit program
            Change TCU to multi textured blit program
            Blit 8x8 8-bit block
        }
    }
}
```

"B" Field structured blocks

This is the equivalent of the B frame blocks in field format. Instead of having 2 motion vectors this type has 4 motion vectors and 4 field selection bits. Apart from the interlacing the processing for the type of block is the same as the B frame block. The flow for this block is as follows:



The reference blocks are generated from the 4 motion vectors. These are used to generate 4 texture coordinates that will be used to generate the 2 reference frames. These are then used to do a multi texture blit operation in which the shader will be used to interlace & average the incoming pixels, then add or subtract the residual difference. This will be done for every block that has a residual difference present in the macro block control command. If a block has no residual difference a simple blit operation will be used to copy the block from the reference picture to the output picture, but the interlacing & averaging will still be done. The proposed driver pseudo code is as follows:

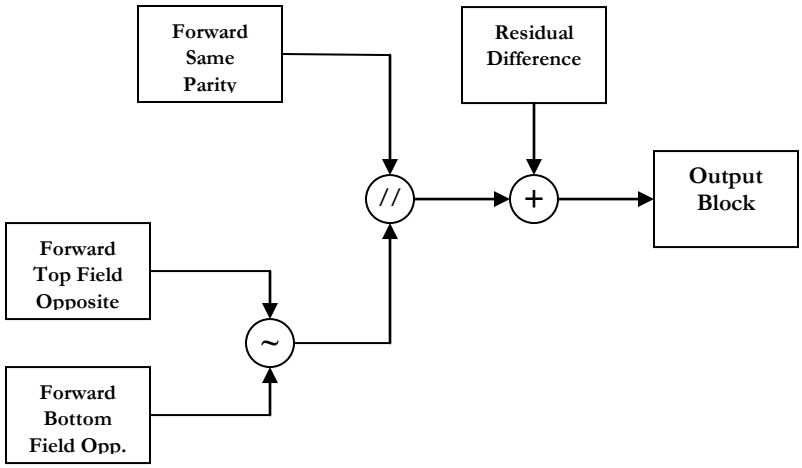
```
// Sample DRIVER Pseudo code for P field blocks
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
```

```
    for(i = 0; i < 6; i++)
    {
        Calculate Forward Top Field Reference block
        TexCoord location
        Calculate Forward Bottom Field Reference block
        TexCoord location
        Calculate Backwards Top Field Reference block
        TexCoord location
        Calculate Backwards Bottom Field Reference block
        TexCoord location
        if(HaveResdiffBlocks[i])
        {
            Change shader to interlace, average and add
            resdiff program
            Change TCU to multi textured blit program
            Calculate Residual Difference TexCoord location
                        from offset_into_resdiff
            Blit 8x8 8-bit block
            offset_into_resdiff++
        }
        else
        {
            Change shader to interlace, average blit program
            Change TCU to multi textured blit program
            Blit 8x8 8-bit block
        }
    }
}
```

Dual Prime blocks

Dual-prime blocks are relatively simple. The MPEG-2 spec goes into great detail about what dual-prime blocks are, but it over complicates the matter. Dual primed blocks are composed of a normal Forward predict block which is averaged with another reference block. This other reference block is generated as follows: the TOP field of the block is generated from the BOTTOM field of block while the BOTTOM field is generated from the TOP field from another block. This field selection order never changes, the processing flow for this block is as follows:



DXVA passes 4 motion vectors into the macro block control command, only 3 are used because one of them is a duplicate. These are used to generate 3 texture coordinates that will be used to generate the 1 normal reference block and 1 interlaced reference block. These are then used to do a multi texture blit operation in which the shader will be used to interlace & average the incoming pixels, then add or subtract the residual difference. This will be done for every block that has a residual difference present in the macro block control command. If a block has no residual difference a simple blit operation will be used to copy the block from the reference picture to the output picture, but the interlacing & averaging will still be done. The proposed driver pseudo code is as follows:

```
// Sample DRIVER Pseudo code for P field blocks
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
```

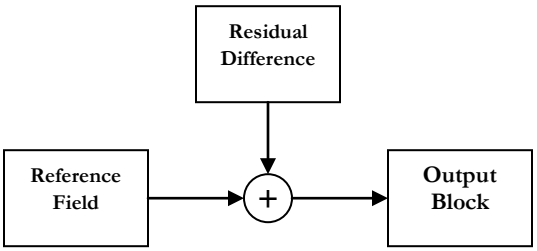
```
            Program Texture B to point at resdiff blocks
        }
        for(i = 0; i < 6; i++)
        {
            Calculate Forward Same parity Reference block
            TexCoord location
            Calculate Opposite Top Field Reference block
            TexCoord location
            Calculate Opposite Bottom Field Reference block
            TexCoord location
            if(HaveResdiffBlocks[i])
            {
                Change shader to dual primed program.
                Change TCU to multi textured blit program
                Calculate Residual Difference TexCoord location
                                from offset_into_resdiff
                Blit 8x8 8-bit block
                offset_into_resdiff++
            }
            else
            {
                Change shader to no res-diff dual primed program
                Change TCU to multi textured blit program
                Blit 8x8 8-bit block
            }
        }
    }
```

"I" Frames

Same as Frame-structured pictures above.

"P" Field structured blocks

P Field blocks are done in the same way as the P-Frame blocks in the Frame structured pictures. But instead of doing the whole 16 lines only 8 interlace lines are done according to which field we are processing. The flow for a block is as follows:



The reference field is read as though a P Frame Field mocomp is being done. Then the shader plane equations are set to select which field we are going to motion compensate. The pixel unit is also set up so that only the lines for the field where the result is going are written to the frame buffer. All the generation of the texture coordinates and addition of the residual difference is done as if a P-Frame block is being processed. The proposed driver pseudo code is as follows:

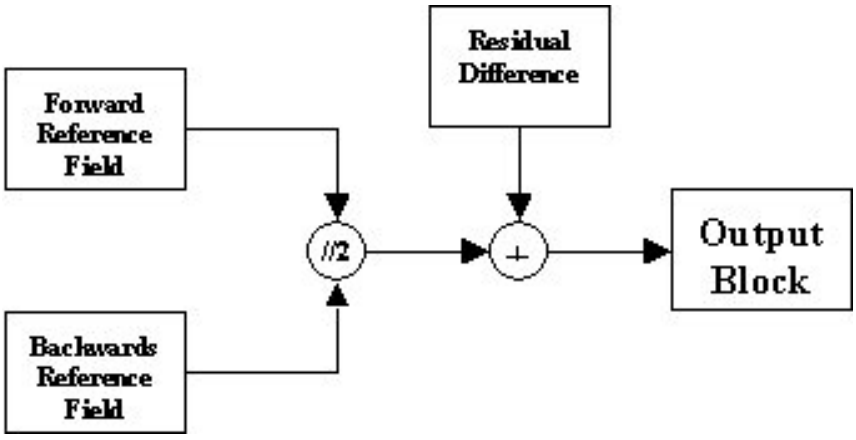
```
// Sample DRIVER Pseudo code for P frames
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 4; i++)
    {
        Calculate Reference block Tex Coords.
        Load field selection
        Load Destination Field
```

```
                if(HaveResdiffBlocks[i])
                {
                    Change shader to add field-resdiff program
                    Change TCU to multi textured blit program
                    Calculate Residual Difference TexCoord location
                                from offset_into_resdiff
                    Blit 8x8 8-bit block
                    offset_into_resdiff++
                }
                else
                {
                    Change shader to blit program
                    Change TCU to single textured blit program
                    Blit 8x8 8-bit block
                }
            }
        }
```

“B” Field structured blocks

Again the processing of this type of block is similar to its Frame Structure counterpart. Both the forward and backwards reference fields have their own field selection bit. Again only 8 lines are being processed instead of 16. The processing flow is as follows:



The processing for this type of block is done basically in the same way as its Frame structure counterpart. The Shading unit plane equations are being used to do all the processing or interleaving the incoming blocks. The proposed driver pseudo code is as follows:

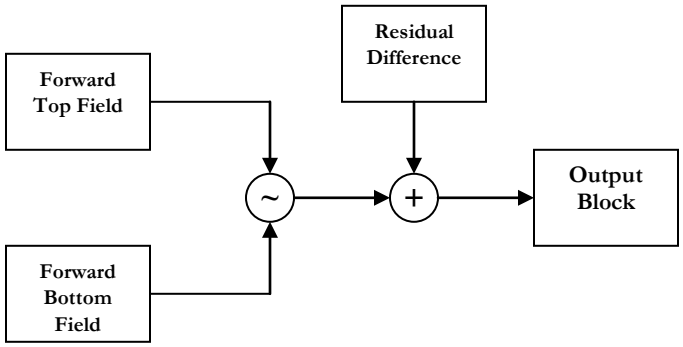
```
// Sample DRIVER Pseudo code for B frames
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 4; i++)
    {
        Calculate Forward Reference block TexCoord location
        Calculate Backward Reference block TexCoord location
        Load Forward field selection
        Load Backwards field selection
        Load Destination Field
```

```
            if(HaveResdiffBlocks[i])
            {
                Change shader to field average and add resdiff
                        program
                Change TCU to multi textured blit program
                Calculate Residual Difference TexCoord location
                        from offset_into_resdiff
                Blit 8x8 8-bit block
                offset_into_resdiff++
            }
            else
            {
                Change shader to field average blit program
                Change TCU to multi textured blit program
                Blit 8x8 8-bit block
            }
        }
    }
```

"P" 16x8 MC structured blocks

This kind of motion compensation is unique to field structure pictures. In this type the macroblock is split up into 2 16x8 sections. However the processing of those sections are the same as the P Field block. The flow for this macro block is as follows:



Following the P field block processing, the macro block's texture coordinates are generated from the incoming motion vectors. The shader equations are then set up to select which field the lines are coming for both incoming fields. The plane equations are also used to switch between the top 16x8 section and the lower 16x8 sections. The proposed driver pseudo code is as follows:

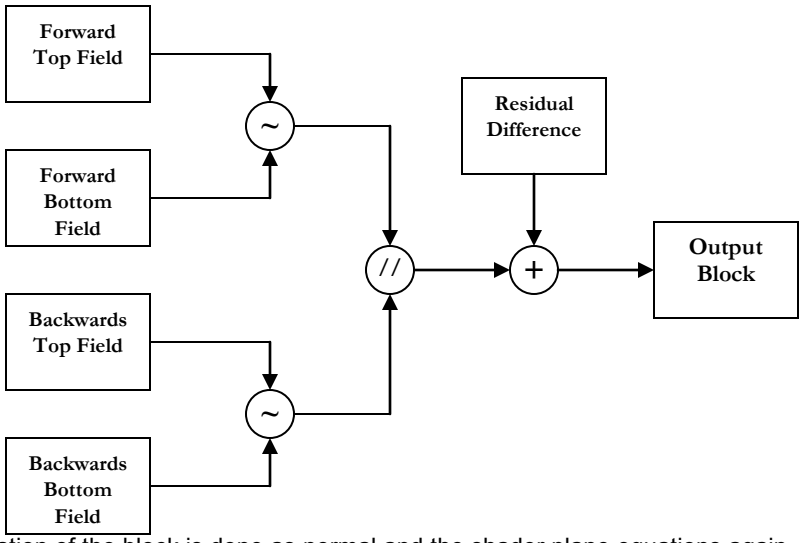
```
// Sample DRIVER Pseudo code for P field blocks
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 4; i++)
    {
        Calculate Top Field Reference block TexCoord location
        Calculate Bottom Field Reference block TexCoord location
        Load Top field selection
        Load Bottom field selection
        Load Destination Field
        if(HaveResdiffBlocks[i])
        {
            Change shader to 16x8 and add resdiff program
```

```
                Change TCU to multi textured blit program
                Calculate Residual Difference TexCoord location
                        from offset_into_resdiff
                Blit 8x8 8-bit block
                offset_into_resdiff++
            }
            else
            {
                Change shader to 16x8 blit program
                Change TCU to multi textured blit program
                Blit 8x8 8-bit block
            }
        }
    }
```

“B” 16x8 MC structured blocks

Again the flow of this type of macro block follows the same pattern as its B-Field counter part. But instead of only having 2 fields to process we have 4 because of the 16x8 formatting. The flow for this block is as follows:



The generation of the block is done as normal and the shader plane equations again relied upon to select the fields from the incoming blocks. The proposed driver pseudo code is as follows:

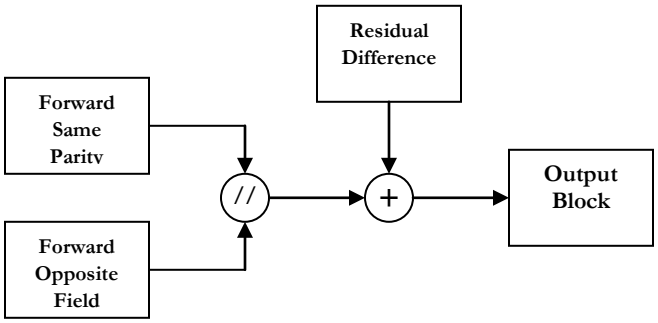
```
// Sample DRIVER Pseudo code for P field blocks
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 4; i++)
    {
        Calculate Forward Top Field Reference block
        TexCoord location
        Calculate Forward Bottom Field Reference block
        TexCoord location
```

```
                Calculate Backwards Top Field Reference block
                TexCoord location
                Calculate Backwards Bottom Field Reference block
                TexCoord location
                Load Forwards Top field selection
                Load Forwards Bottom field selection
                Load Backwards Top field selection
                Load Backwards Bottom field selection
                Load Destination Field
                if(HaveResdiffBlocks[i])
                {
                    Change shader to interlace, average and add
                    resdiff program
                    Change TCU to multi textured blit program
                    Calculate Residual Difference TexCoord location
                                from offset_into_resdiff
                    Blit 8x8 8-bit block
                    offset_into_resdiff++
                }
                else
                {
                    Change shader to interlace, average blit program
                    Change TCU to multi textured blit program
                    Blit 8x8 8-bit block
                }
            }
        }
```

Dual Prime blocks

This is a simplified case of the Frame Structure dual primed block. But instead of the 4 fields to interpolate we only have 2 because of the field nature of this picture. The flow of the block is as follows:



This is in fact a hard wired B-field block, with both forward and backwards textures pointing to the same picture and the field signs hard wired to point to opposite fields. Therefore the proposed driver pseudo code is as follows:

```
// Sample DRIVER Pseudo code for B frames
bool HaveResdiffBlocks[6]
dword offset_into_resdiff

for(every macroblock in the picture)
{
    offset_into_resdiff = 0
    Read pattern code and fill in HaveResdiffBlocks accordingly.
    if(have any resdiff blocks)
    {
        Copy ResDiff blocks to Scratch pad.
        Program Texture B to point at resdiff blocks
    }
    for(i = 0; i < 4; i++)
    {
        Calculate Forward Reference block TexCoord location
        Calculate Backward Reference block TexCoord location
        Load Forward field selection
        Load Backwards field selection
        Load Destination Field
        if(HaveResdiffBlocks[i])
        {
            Change shader to field average and add resdiff
                    program
            Change TCU to multi textured blit program
            Calculate Residual Difference TexCoord location
```

```
                                    from offset_into_resdiff
                    Blit 8x8 8-bit block
                    offset_into_resdiff++
                }
                else
                {
                    Change shader to field average blit program
                    Change TCU to multi textured blit program
                    Blit 8x8 8-bit block
                }
            }
        }
```

8.9.6 Summary

The main features of this implementation are:

- Up to 6 Textures will be used (if 8-bit overflow is done);
- Textures are reprogrammed in a block by block basis;
- Main processing is done by the Shading unit;
- All processing is done in signed arithmetic. The Shading unit is used to Convert from Unsigned to signed and the Pixel unit to do the reverse;
- Pixel sampling rules are changed to sample in the centre of the pixel;
- Residual difference blocks are programmed as 2D textures but the TexCoord is recalculated in the TexCoord Unit. This saves the time taken to re-format the residual difference;
- Macro Block control buffers should be stored in the system memory otherwise the command processing will be slowed down significantly.

9 Antialiasing

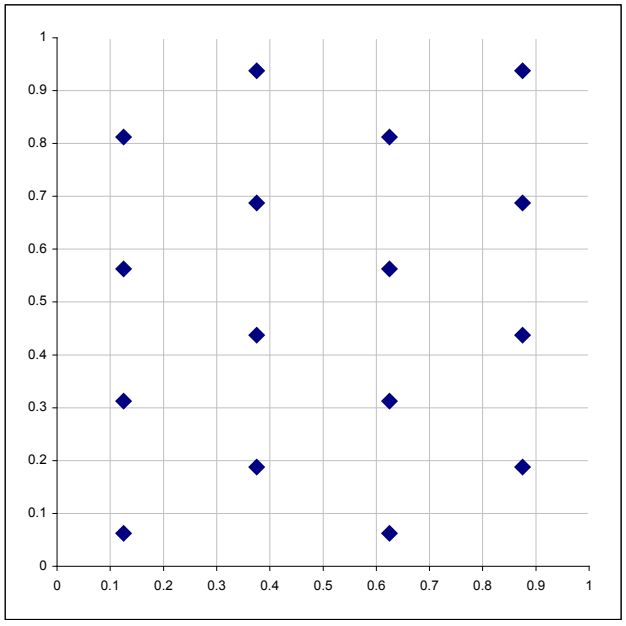
9.1 Sample point position (how many sample points)

P10 architecture supports edge anti-aliasing by performing rasterisation of the object several times and accumulating the results. Each time the object is rasterised the sampling point position is slightly offset.

P10 allows for a maximum of 16 sample point positions to be downloaded and used. Sample point positions on the P10 are pairs of 4 bit numbers and are downloaded using either the AALineSamples or the AATriangleSamples Tag. One sample position is contained per byte of tag data. These 4 bit numbers represent the sub-pixel position of the sampling point relative to the top-left corner of a given pixel.

The sample point positions are always downloaded as positive numbers so there must be an internal adjustment to correctly position the AA sample points around the conceptual center of the pixel. This correction is necessary to handle both the OpenGL and the DirectX pixel sampling rules.

The following is an example of a 16-element spatial arrangement (the center point of the pixel is assumed to be at (0.5, 0.5)).



These values are required to recreate the same sample pattern on the P10:

X	6	10	14	2	6	14	10	6	10	2	2	6	14	14	2	10
Y	7	1	3	1	11	7	9	15	5	9	13	3	15	11	5	13

Additionally to downloading the sample pattern, the Rasteriser Unit needs to be informed of the numbers of samples to perform per rendering operation. This is achieved by setting the AATriangleSamplePoints and the AALineSamplePoints fields in the RasterMode tag. The number placed in these fields should be (the number of samples – 1);

Tag	Requirements
RasterMode	Set up the correct values of the AA triangle sample points and the aa line sample points (should be set to number of samples – 1).
AATriangle Samples	Used to download up to 16 triangle sample positions.
AALineSamples	Used to download up to 16 line sample positions.

Once the rasteriser has been configured any objects rendered will not only generate a tile mask but also a coverage value will be generated. This coverage value represents the number of sampling iterations in the rasteriser unit considered the pixel to be associated the object. According to the OpenGL specifications this coverage value replaces the pixels associated alpha value and is used to perform subsequent blend operations.

This is an example of a 4-sample pixel unit program pre-amble to generate the OpenGL coverage values.

```
    PassA(A[0]) JumpFalse (AA,@finish);
    E = 4 W[13] = PassA(E);
    E = 64 W[14] = PassA(E);      // Load W[14] with scale factor

    // W[3] = scaled coverage - the scale factor has been loaded into W[14]
    E = Coverage W[3] = MultL(A[14], E);

    // Compare coverage value with num samples - this has been loaded into W[13]
    E = Coverage Flag = Sub(E, B[13], ==);

    // If coverage == num samples, W[3] = 255
    E = 255 W[3] = SelectA(E, B[3]);

    // W[3] = scaled coverage * alpha
    W[3] = Modulate(A[3], F[3]);

finish:
    // standard processing begins here W[3] holds the alpha/coverage value
```

The program shown makes use of a programming optimization to skip the alpha value calculations if the tile is totally within the object.

Also, the program has to perform a check to see if the coverage value received is equal to the current number of sample points. This check has to be performed to clamp the maximum coverage value within the 0-255 range of the pixel unit.

9.2 OpenGL Antialiasing (triangles, dual line patterns, points)

The OpenGL specifications allow for specific control over the performing anti-aliasing on each of the main render primitives. To enable anti-aliased rasterisation of a specific type of primitive the corresponding bit in the Primitive Setup Unit needs to be set. (*AAPointEnable, AALineEnable* and *AATriangleEnable* bits in **PrimSetUpMode** command). Enabling a bit field in this register does two things:

1. The geometry of the primitive is changed to the smooth version.
2. If the primitive type has anti-aliasing enabled then the rasteriser is forwarded a register to enable coverage value generation.

The primitive setup mode register has the following fields to control anti-aliased primitives:

Field	Requirements
AAPointEnable	Enable the generation of smooth points.
AALineEnable	Enable the generation of smooth lines.
AATriangleEnable	Enable the generation of smooth triangles.

This control is necessary as there are various OpenGL geometry rules for primitives that are dependent upon whether the primitives are being anti-aliased (smoothed) or not. Aliased points are rendered as squares where as smooth points are round. Aliased lines have ends which are perpendicular to the major axis of the line, anti-aliased lines have ends perpendicular to the direction of the line.

Additionally, the P10 architecture allows for a anti-aliased line optimization to be performed. The benefit of running the chip in this mode of operation is that smaller numbers of sample points can be used. Generally, when anti-aliasing the sample points are chosen so that no matter what orientation the object being drawn is to the sample pattern the coverage values will produce a smooth edge value.

However, by using the line orientation to select the sample pattern allows a less generalized, more specialized sample pattern to be downloaded. This dual sample method generates smooth lines using only four samples that are almost comparable to a 16-sample general sample pattern.

To enable this the DualAALineSamplePatterns bit in the **RasterMode** command needs to be enabled. When this bit is set it will cause the AA sample pattern for lines to depend upon the orientation of the line being draw (i.e. x major or y major). If the orientation is x major then the AA sample points will be taken from the lower half of the AA line sample point table, otherwise for y major lines they are taken from the upper half of the table.

Field	Requirements
DualAALineSample Patterns	Enable using dual sample patterns for lines.

Obviously, as the sample table is divided in two in this mode, the maximum number of samples in each set is 8 when in this mode.

9.3 Full Scene AA (FSAA, Multi sampling, Super sampling)

This section describes an example of programming P10 to perform multi-sample full scene antialiasing (FSAA) as required by the DX7 API in response to a request for *D3DANTIALIAS_SORTINDEPENDENT*, or for a multisample surface in DX8. Note that for DX7 the number of samples required *N* (the quality of the antialiasing) is set by the driver, usually from a selection made by the end user via a control panel interface, while it DX8 it is a attribute of the surface.

Multi-sample antialiased rendering can be broken down into the following process:

- Allocation of multiple frame buffers and, if required, local buffers, one for each sample position to be recorded.
- Rendering of the scene as normal. The difference however, is that although shading is performed just once for each pixel, visibility is determined in the rasteriser for multiple jittered sample positions within each pixel. Communicated via coverage masks broadcast to the other relevant units in the P10 core, the visibility status of each sample is used to update the relevant sub-frame and -local buffer. In the case of the local buffer, separate depth values are interpolated for each sample position to permit correct handling of intersecting edges.
- Multiple copies of the same scene now exist in each of the sub buffers, each sampled from a slightly different position. At the end of rendering the frame, prior to blitting or flipping to the front buffer for display, these images are blended together in a combining blit, to the back buffer surface. Any dithering, if required, is applied at this stage.

The individual steps are now described in more detail. When the requirement for FSAA is determined from the render state, video memory is allocated for *N* additional frame buffers. These frame buffers are created in addition to the back buffer render surface so that they are independent of any blit operation or flip chain.

The render target base address is then switched from the back buffer surface to the start of this newly allocated video memory. Similarly, additional local buffer memory is also allocated. The setup of the **FBBuffer**, **FBBaseAddr**, **LBBuffer** and **LBBaseAddr** registers to the core are then as described in the sections on Local- and Framebuffer Processing above.

Control of the addressing of the extended multi-buffer memory is handled differently in P10 for the frame buffers and local buffers. The addressing of the frame buffer memory is controlled by a pixel address unit (PAU) program downloaded from the driver. Because of this the multiple frame buffers are created as a single block, equal to *N* times the size of that of the back buffer after it has been resized for tile alignment.

```
// Calculate the tile aligned dimensions of the back buffer surface
dimX = DDSurf_Width(pSurfRender);
```

```
dimX = ((dimX + TILEWIDTH) &~ TILEWIDTH);
dimY = DDSurf_Height(pSurfRender);
dimY = ((dimY + TILEHEIGHT) &~ TILEHEIGHT);

// Determine the memory allocation requirement for the antialias buffer
mmrq.dwBytes = ( dimX * dimY * pixelSize * DXGlobals.dwAASampleCount );
```

The sub-buffers corresponding to each sample position are considered to be positioned consecutively in the memory block with the offset to each performed by the destination address setup in the PAU program. Reset the PAU program address:

```
SEND_P10_DATA( PixelProgramAddr,        0 );
```

The PAU for three samples or more is as follows:

```
Program(PAUaaBufferRenderGTE3, 0x00)
Add(r0, A0, tileX);
Add(r1, A0, tileY);

// Load the required number of samples - 2
Copy(r2, a2);

// First subpixel buffer
SetTileMaskFromCoverage();
SendDestAddrAndTile(buf0, puReg0, First);

loop2:

// Sucessive subpixel buffer
Add(r1, A1, r1);            // Increment tileY
LoadXY(r0, r1);
SetTileMaskFromCoverage();
SendDestAddrAndTile(buf0, puReg0, Middle);

Dec(r2, r2);
JumpNotZero(r2, loop2);

// Last subpixel buffer
Add(r1, A1, r1);            // Increment tileY
LoadXY(r0, r1);
SetTileMaskFromCoverage();
SendDestAddrAndTile(buf0, puReg0, Last);
```

The arguments A0 and A1 seen in the above program are loaded as constants with the register **FBAddrInfo** as shown below.

```
SEND_P10_DATA( FBAddrInfo0,        (dimY << 16) | 0);
```

In contrast, the addressing to each sub-region of the antialiasing local buffer is performed automatically in hardware in the local buffer address unit. This assumes that the multiple

buffers are consecutive in memory, with the added proviso that the buffers are of a size such that the provided offset is a multiple of 1024bytes and the memory allocated supports a stride of 24 byte tiles. The offset is setup as part of the **LBMode** register.

The programming required for this setup is provided below.

```
// Calculate the tile aligned dimensions of the render targets local buffer
dimX = DDSurf_Width(pZBuffer);
dimX = ((dimX + TILEWIDTH) & ~TILEWIDTH);
dimY = DDSurf_Height(pZBuffer);
dimY = ((dimY + TILEHEIGHT) & ~TILEHEIGHT);

// Determine the preliminary memory allocation requirement
mmrq.dwBytes = ( dimX * dimY * ZpixelSize );

// Setup the offset between buffers in units of 1024 byte tiles
mmrq.dwBytes /=TILEAREA;                    // Tile byte size of single buffer
mmrq.dwBytes += 1023;
mmrq.dwBytes &= ~1023;                      // To units of 1024

// Set up LBMode bits for Multisample Antialiasing
P10LBMode.bits.OffsetBetweenBuffers = mmrq.dwBytes >> 10;

// Local buffer has hardwired stride of 24byte tiles. Add 24 to ensure no data missed.
mmrq.dwBytes += 24;

// Total allocation memory requirement is then
mmrq.dwBytes *= (TILEAREA * dwMultisampleCount);
```

In addition to the **LBMode**, other render state that must be setup correctly in the P10 core include the **RasterMode**, **PrimSetupMode**, **DepthMode** and **AATriangleSample** registers.

The **RasterMode** must be setup with the desired number of samples and whether or not the coverage information shall be used as a mask or a count.

```
P10RasterMode.bits.AATriangleSamplePoints = DXGlobals.dwAASampleCount - 1;
P10RasterMode.bits.AAType = 0;                          // Mask form of coverage AA
```

The **PrimSetupMode** informs the primitive setup engine that antialiased triangles shall be rendered.

```
P10PrimSetupMode.bits.AATriangleEnable = 1;
```

The positions of the jittered sample points are loaded via the **AATriangleSamples** register, setup in the driver according to the required number of samples. This is further described in Section 9.1.

The **DepthMode** register also includes fields specific to multi-sample operation.

```
// Set up DepthMode bits for Multisample Antialiasing
P10DepthMode.bits.MultiSampleEnable = 1;

for (mask = 0; mask < DXGlobals.dwAASampleCount; mask++)
    P10DepthMode.bits.MultiSampleMask |= (1 << mask);
```

The above shows the settings for antialiasing, but for multi-sample rendering effects other than antialiasing, such as motion blur, the mask will not always include all the samples. For this reason the UseAllSubSamples field is included to inform the P10 core that all subsamples are used and optimisations can be used.

```
P10DepthMode.bits.UseAllSubSamples = 1;
```

The pixel unit programs for the rendering phase are unchanged from those described in Section 8.7, Framebuffer Processing with the exception that the issue of dithering must be postponed until after the sub-buffer image combine blend and not performed for each individual rendered sub-buffer image. Therefore, the alpha blending pixel unit program load is independent of whether or not antialiasing is used. The First, Middle and Last program calls in each of the *SendDestAddrAndTile* instructions in PAU program ***PAUaaBufferRenderGTE3*** above refer to the same program address, 0, and are there for the correct operation of the pixel address unit, not to influence the pixel unit operation.

```
SEND_P10_DATA( PixelMode, 0 );
```

Finally, a set of additional pixel unit and pixel address unit programs is required for the combining blit operation.

The source and destination base address setup for the blit from the enlarged antialiasing buffer to the actual back buffer are performed using two P10 frame buffers, setting up both FBBuffer0, FBBaseAddr0 and FBBuffer1, FBBaseAddr1 pairs as described in Section 8.7.

The PAU program used to process the blit is similar to that used during the rendering phase. The versions for greater than or equal to three samples is given below (for two samples no loop is needed):

```
Program(PAUaaBufferCombineGTE3, 0x00)
Add(r0, A3, tileX);
Add(r1, A3, tileY);

// Load the required number of samples - 1
Copy(r2, a2);

// First subpixel buffer
SendSourceAddrAndTile(buf1, puReg1, First);

loop2:

// Second subpixel buffer
Add(r1, A1, r1);                // Increment tileY
LoadXY(r0, r1);
```

```
SendSourceAddrAndTile(buf1, puReg1, Middle);

Dec(r2, r2);
JumpNotZero(r2, loop2);

// Set up the addresses for the display target write.
LoadXYFromTile();
SendDestAddrAndTile(buf0, puReg0, Last);
```

As before, the constants A0, A1,A2 and A3 used here are loaded as:

```
SEND_P10_DATA( FBAddrInfo0,        (dimY << 16) | 0);
SEND_P10_DATA( FBAddrInfo1,        0 | (DXGlobals.dwAASampleCount-1));
```

As shown in the PAU program's *SendDestAddrAndTile* calls above, the pixel unit programs are split into two or more passes depending on the number of samples required. The first pass program for the first sample simply loads the colour data to local accumulation registers, while the middle program is used for accumulation of the subsequent samples. Finally, the last program scales the result and extracts the required colour. The examples shown are for a 565 16-bit colour format, without dithering. Alternate programs for other supported pixel formats are also required.

```
Program(PUaaBufCombineFirst565, 0x00)
            W[0] = PassA( P[4.L] );
E = 0       W[1] = PassA( E );
            W[2] = PassA( P[4.M] );
E = 0       W[3] = PassA( E );
            W[4] = PassA( P[5.H] );
E = 0       W[5] = PassA( E ) Done;

Program(PUaaBufCombineMiddle565, 0x00)
            W[0] = Add( P[4.L], B[0] );
E = 0       W[1] = AddC( E, B[1] );
            W[2] = Add( P[4.M], B[2] );
E = 0       W[3] = AddC( E, B[3] );
            W[4] = Add( P[5.H], B[4] );
E = 0       W[5] = AddC( E, B[5] ) Done;

Program(PUaaBufCombineLast565, 0x00)
// Multiply by 64/N
E = Global[0]           W[8] = MultL(A[1], E);
E = Global[0]           W[1] = MultU(A[0], E);
                        W[1] = Add(A[1], B[8]);
E = Global[0]           W[0] = MultL(A[0], E);
// Divide by 64
E = 0x4   W[1] = MultL(A[1], E);
E = 0x4   W[0] = MultU(A[0], E);
          C[0.L] = Or(A[0], B[1]);
```

```
// Multiply by 64/N
E = Global[0]          W[8] = MultL(A[3], E);
E = Global[0]          W[3] = MultU(A[2], E);
                       W[3] = Add(A[3], B[8]);
E = Global[0]          W[2] = MultL(A[2], E);
// Divide by 64
E = 0x4   W[3] = MultL(A[3], E);
E = 0x4   W[2] = MultU(A[2], E);
          C[0.M] = Or(A[2], B[3]);

// Multiply by 64/N
E = Global[0]          W[8] = MultL(A[5], E);
E = Global[0]          W[5] = MultU(A[4], E);
                       W[5] = Add(A[5], B[8]);
E = Global[0]          W[4] = MultL(A[4], E);
// Divide by 64
E = 0x4                W[5] = MultL(A[5], E);
E = 0x4                W[4] = MultU(A[4], E);
                       C[1.H] = Or(A[4], B[5]) Done;
```

The program addresses are set up as follows, where DownloadPixUnitProgWithCount and ExtendPixUnitProgWithCount are macros that perform the required **PixelProgramAddr** and **PixelProgramData** dma and also update a program length count in pContext.

```
P10PixelMode.bits.TileAddrFirst = 0;
DownloadPixUnitProgWithCount( PUaaBufCombineFirst565 );

P10PixelMode.bits.TileAddrMiddle = pContext->dwPixelProgramLength;
ExtendPixUnitProgWithCount(PUaaBufCombineMiddle565);

P10PixelMode.bits.TileAddrLast = pContext->dwPixelProgramLength;
          ExtendPixUnitProgWithCount( PUaaBufCombineLast565 );
```

10
Exotica

10.1 Beyond ordinary graphics functions (imagination, examples)

10.2 Vertex Shader applications

10.2.1 Tessellation

Tessellation is the process where some high order surface is divided up in to triangles or quads for display. The finer the tessellation the higher the quality of the image will be. It only makes sense for the Vertex Shading Unit to get involved in the tessellation if the level of tessellation is high enough to amortise the additional driver work to present the individual patches for tessellation and to accommodate a two pass algorithm[3] where the first pass generates the tessellation points and the second pass transforms and lights them.

There are many ways to describe these surfaces - Bezier patches, Cubic splines, B-splines, NURBS, etc. and the thing they all have in common is a set of control points and a weighting function which relates the control points to each tessellation vertex.

The surfaces are frequently defined by a matrix equation as follows (assuming a bicubic patch, but similar formulations exist for surfaces with a different order):

$$Q(u,v) = \mathbf{UMPM}^T\mathbf{V}$$

$$\mathbf{U} = \begin{bmatrix} u^3 & u^2 & u^1 & u \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} v^3 & v^2 & v^1 & v \end{bmatrix}^T$$

where P is the 4x4 matrix of control points and M is the basis matrix which is a function of the surface type.

Some basis matrices are:

[3] In some cases a single pass algorithm can be used where the tessellation points are calculated, transformed and light in the same program with out intermediate results (i.e. the tessellated vertices) being written out to memory. If the tessellation level for a patch if fine enough or a number of patches can be tessellated before changing to the second pass then there is probably not much in the performance of either method. The two pass method is more robust and probably easier to manage (from a vertex shading program perspective).

$$\mathbf{M}_{Bezier} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{M}_{B-spline} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \quad \mathbf{M}_{Catmull-Rom} = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

These equations can also be cast into the form:

$$Q(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} \omega_{ijuv} P_{ij}$$

where ω_{ijuv} is a weight for a set of i, j, u and v values.

The tessellation not only generates positions but can also generate colours, textures and normals. Many types of surfaces can also provide the analytic normals at the tessellation vertices which will be more accurate and less hassle (for the user).

Each tessellation vertex is assigned a u, v position (normally in the range 0…1) and the uv tuple together with the order of the surface (i.e. linear, cubic, etc.) and the basis functions define the weights applied to each control point (the number of control points is strongly related to the order of the surface) to calculate the parameter at the uv position. Each uv position will have its own set of unique weights. If a triangle is being tessellated in the plane, maybe for displacement mapping (see later) or to get better lighting, then the uv weights are replaced by the barycentric coordinates for the required tessellation vertices.

The basic principal is that the control points are stored in the coefficient memory and the uv[4] position or set of weights[5] for that uv position are written to the Vertex Shader as if they were the parameters for a corresponding tessellation vertex. The last weight parameter is set up as the trigger parameter and when this is received, conceptually at least, the vertex shading program is run. It is little more than one or more weighted sums of the control points. The results of the weighted sums are written as parameters and the window coordinate is set to some value (it is later discarded) which will pass through the clip test process. The Vertex Machine Unit is set up to do Points and the Geometry Unit has the **UploadParameters** mode set to *VertexBufferData*. The Context Unit has the address where in memory the tessellation vertex parameters are to be written loaded into the VertexBufferAddr register. The **WaitForCompletion** command can be used between passes to ensure the results of the first pass are in memory before the second pass starts, however this is not really necessary except when the number of tessellation vertices is low. The second pass will read the results of the first pass as a vertex array.

For bicubic patches there are 16 control points so each component takes 16 multiply accumulated to calculate so for just a xyz position will take 48 cycles or an effective throughput of 3 cycles for 16 VPs. As more data types (colour, texture, etc.) are

[4] Just storing the uv position would result in many more calculations being needed. For example all the products of (u^3, u^2, u, 1) and (v^3, v^2, v, 1) would be needed. The cost of this can be reduced by using Homer's rule for polynomial evaluation.

[5] Using a set of weights for each uv position makes the assumption that it is more economical to calculate the weights once and store them rather than calculate them on the fly. This is a reasonable assumption as patches are rarely tessellated in isolation within a frame or between frames. The set of weights lend themselves to being cached in on-card memory.

evaluated then the through put will go down. When processing a mesh of patches the circular buffer addressing mode in the coefficient memory can be used so only the four new control points need to be written, replacing the oldest set of 4 control points.

OpenGL evaluators can be done using the scheme outlined so far. The evaluators allow for different orders for each parameter type being evaluated and this can be handled by different programs and sets of weights or by promoting all evaluators to the highest order (degree elevation). The vertex array allow the individual parameters to be read in consecutively or from their own arrays so there is plenty of flexibility here to allow the data to be read in whatever format is convenient for tessellation. The preservation of the current parameters and the material (if colour material is enabled) is something the ICD will need to worry about.

DX now includes some support for standard tessellation methods such as Bezier, but also has a unique triangle tessellation scheme which uses the normals at each vertex to predict how the surface is behaving. The purpose of this scheme is to allow existing artwork to be 'improved', however Microsoft only recommend one or two levels of tessellation so the driver overhead per triangle may be too much to make doing the tessellation in P10 viable.

The driver should test that the control mesh is in view before proceeding to tessellate it.

The position (regular or irregular) and sequence the tessellation vertices are generated in is defined by the order the weights (or barycentric coordinates) are presented in. On the second pass the preferred primitive type is Grid as this maximises the reuse of vertex data and for high order surfaces this is easy to arrange as the uv positions are basically organised on a rectilinear grid. For triangles this is not as easy so the best primitive type is a triangle strip with some duplicated tessellation vertices at the end of each 'row' to allow one continuous strip to weave its way in serpentine fashion over the area of the base triangle.

The type of tessellation can be fixed or adaptive. If adaptive tessellation is required then it is up to the driver to estimate the projected area on the screen of the control points (or the convex hull holding the control points) and select (or generate) an appropriate set of weights. It probably makes more sense to do this estimation on the control mesh rather than individual patches because the main danger to watch out for with adaptive tessellation is cracks between patches where different levels of tessellation have been selected.

This scheme can be adapted to subdivision surfaces, but with the driver probably handling the triangles or quads with irregular vertices (i.e. the valence or number of edges arriving at a vertex is not the usual amount for that type of subdivision surface).

10.2.2 Displacement Mapping

Displacement mapping is a technique where a surface is tessellated and the tessellation vertices are displaced along the normal by an amount looked up from a displacement map. The displacement map is really a height field stored in a texture map. The displaced surface will naturally also perturb the normal from the base surface so the surface lighting will match the new geometry. The advantage displacement mapping has

over bump mapping[6] is that the visibility along the silhouette edge follows the cues given by the lighting, but this comes at a very high cost as the tessellation triangles need to be very small - of the order of a few pixels in size.

The displacement mapping process can be broken down into the following stages:

- Tessellate the input surface. This has already been covered.
- For each tessellation vertex sample the displacement map and return the displacement D for each vertex. The sampled value of D cannot be negative otherwise the texture filtering will not work so the displacement map is biased to make all entries positive. The sampled results are stored in an array which can then be read in as an element of a vertex array operation.
- Perturb the vertex position along the direction of the normal using the equation:

$$\mathbf{P}' = \mathbf{P} + Ds\hat{\mathbf{N}}$$

  where s is an optional scale value. The original vertex position is calculated as per the tessellation methods already outlined.
- Calculate the new normal for the perturbed position.
- Render the new triangles.

We will now expand on two of these stages: how to sample the displacement map and how to calculate the new normal.

10.2.2.1 Sampling the Displacement Map

To sample the displacement map each tessellated vertex is past to the rasteriser subsystem as a single pixel point. The point parameters hold the u, v coordinates of the displacement map and the x, y coordinates of the point's location in the displacement buffer where the displacement value is to be written. Successive points will increment through the buffer.

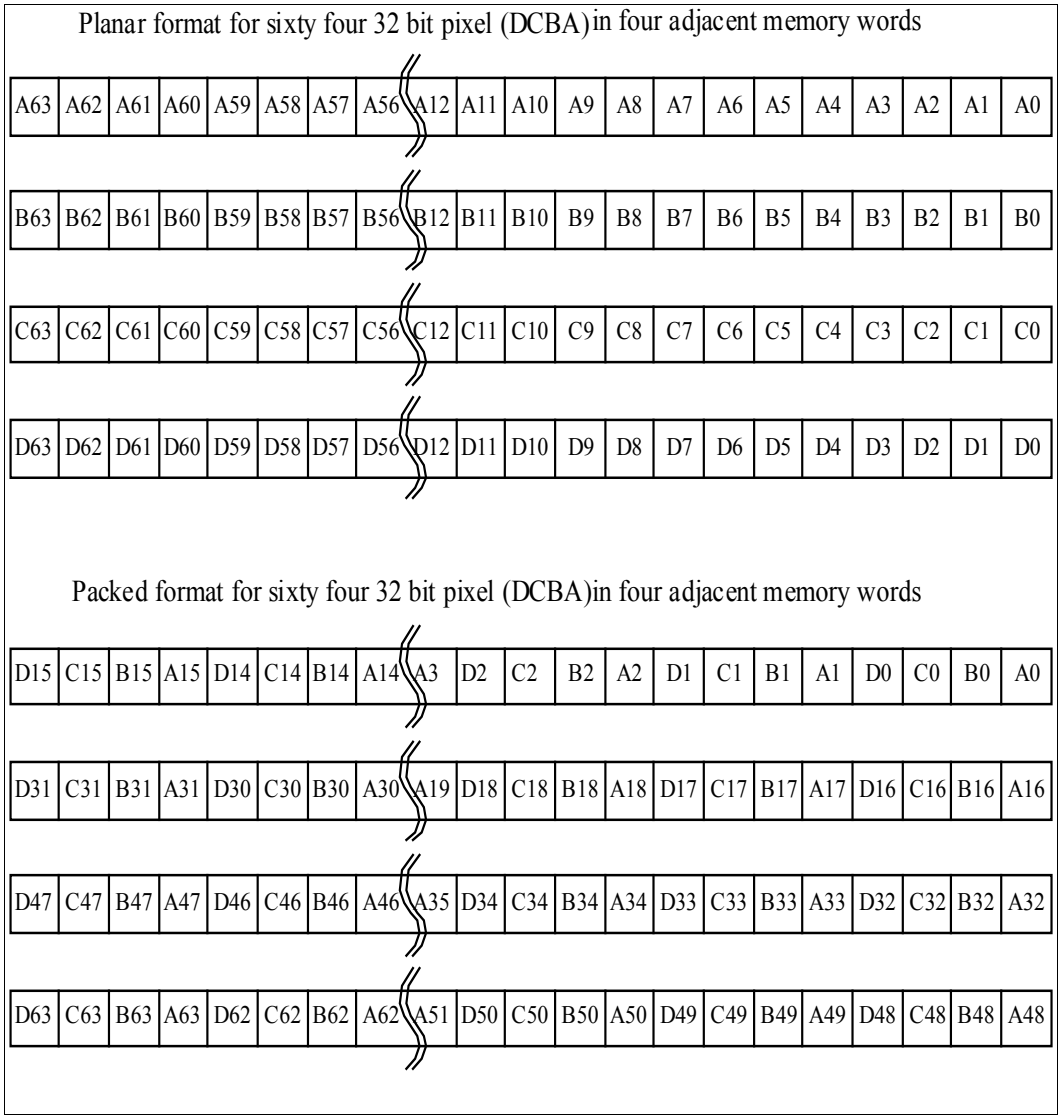
The Texture Coordinate program uses the u, v values directly and doesn't perspectively divide them. The lod value is taken from a global register (if needed) as a fixed level of detail is used across the whole primitive[7] The Shading Program just copies the texture sample to the fragment's colour and the Pixel Address and Pixel Program will append the colour to the displacement buffer.

The displacement buffer is written in byte planar format, however the vertex array support in the GPIO expects to read vertex data in packed byte format. The two formats are shown in the following diagram:

---

[6] Bump mapping simulates fine geometric detail by faking the lighting at each pixel by using texture operations with a height map (or some other encoding of the fine surface detail) to give the specular appearence of desired surface. As only the shading is modified the silhouette edge still looks smooth. Bump mapping does not introduce any more triangles in to the scene.

[7] A varying level of detail also implies the size of the tessellation triangles is changing over the area of the base surface and while this will concentrate the detail where it is most visible it has to be done very carefully not to introduce cracks. If triangles which share a common edge are tessellated and displaced by differing amounts along the common edge then the two triangles will no longer form a continuous surface. This is a very hard problem to solve so the simplification of a constant tesselation level over the base surface is assumed here.



This difference in formats will be solved in future chips by allowing the GPIO to read data in planar byte format, but for now two solutions present themselves.

- The displacement data can be written sparse. Successive points differ in x by 4 and each used byte is written to a separate buffer. The vertex array is set up to read a packed colour from each buffer and the least significant byte will appear as the lowest floating point component. The other three components are undefined. This method will use one input parameter per byte so is quite wasteful of memory and bandwidth, however if only a single displacement byte is being used then this is not too bad.

- The displacement buffer is reformatted using a separate pass after the sampling step but before it is read as a vertex array. This can be done by setting up four texture reads (assuming we are repacking 32 bits) which when combined and written in byte planar format will give the correct results when read in packed byte format. The above diagram shows the desired mapping (read A0, A16, A32, A48 and output as one pixel), (read B0, B16, B32, B48 and output as next pixel), etc. This can be done, albeit with a fairly complicated program.

The **WaitForCompletion** command can be used to ensure the vertex array processing doesn't start until the last of the displacement data has been written to memory.

10.2.2.2 Calculating the Normal

The simplest solution here is not to use vertex lighting but to use the same per pixel lighting techniques used in bump mapping to shade the displaced geometry. The displacement and what ever form the bump map information is encoded (for example as a normal perturbation map) in can share the same texture map, or be in separate maps if more convenient or the displacement and bump mapping are done to different resolutions.

If the displaced normal is required for vertex lighting then this needs to be calculated but the Vertex Shader doesn't have access to any of the other local displaced vertex information. We can borrow from the bump mapping algorithms and sample the normal perturbation map at the same time the displacement is sampled. The normal perturbation is passed back to the vertex shading program in the same word used to send the displacement. The normal perturbation tells how to rotate (or tip) the normal so it will point in the correct direction after the vertex displacement has been applied. The normal perturbation map usually encoded the rotation in tangent space so the original normal will need to be transformed into this space before it can be rotated. The subsequent lighting calculations can be also be done in tangent space. The bump mapping literature should be consulted for details on this.

Cearly from the above the simplest thing to do is to use bump map lighting per pixel and this also has the advantage of only requiring a single value (byte) to be passed back as the displacement value so the additional pass to convert the planar byte data to packed byte data can be avoided.

10.3 Texture Co-ordinate applications

10.3.1 Convolution

A convolution example has been given for the Pixel Unit and this is the fastest way to do a simple convolution, unfortunately things get more complicated when the convolution filter goes outside of the image array when sampling near the edge. Several schemes are used to fill in for the missing pixels, ranging from repeating the edge pixels, repeating a row of border pixels or substituting a constant border colour. These are very difficult or time consuming for the Pixel subsystem to handle, but are handled trivially by the texture wrap modes.

The Texture Coordinate Unit could generate the addresses in much the same way the Pixel Address Unit does and the Shading Unit do the weighted accumulation for the convolution operation. There is no efficient way for the Texture Coordinate Unit to broadcast the same value (current convolution weight) to each fragment processor in the Shading Unit or for the Shading Unit to pick up a pass dependent value from its global

registers. The solution here is for the pixel data to be returned to the Texture Coordinate Unit using feedback and the weighted accumulation to be done here. This has the advantage that the convolution weights and accumulation can all be done in floating point.

The program would consist of a series of subroutine calls to fetch the sample points and do the weighted accumulation. The **GRegBaseReg** is set on each call to sequence through the weights. There will be one subroutine call per convolution weight.

10.3.2 High Order or Multi-tap Filters

The previous example showed how such a filter could be written, albeit using a trivial example, but how does this cope with a more complex filter such as a bicubic filter? With such a filter you calculate a weighting using an equation (for 1D) like:

$$w(u) = au^3 + bu^2 + cu + d$$

where a, b, c and d are the cubic factors and u is the distance from the centre of the sample. If the 2D filter is separable then a similar equation exists but parameterised in v (for displacement in y) and the product of the u and v weights is used. These equations can be programmed quite easily, particularly if reuse is made of weight as successive samples along one axis are made. The calculation of these weights has an approximate budget of 30 or so cycles so is probably not going to be the limiting factor.

If the equations become more complicated or are not separable the calculation of the weights may become a problem. Basically we are trying to evaluate a function of two variables and the input variables have a very limited input range. This can be converted to a texture map lookup so the complexity of the function, even if it is asymmetric, is suddenly an non issue. The granularity of the function is determined by the size of the texture map and if the weights are limited to 8 bits this can be further improved by using bilinear filtering to do piece wise linear approximation of the function.

In this example we will take the 4x4 pixels in the neighbourhood of the centre sample point and look up in a texture map the weight (or height) of the filter. Note the centre sample point is not aligned on a fixed grid as we may be zooming or applying some warping function. The original texture map will be sampled using nearest neighbour filtering and the filter texture map is bilinearly filtered and the weighted sum is done in the Shading Unit via the multi pass method. Ideally the two texture maps are assigned to their own bank in the Primary Texture Cache to reduce the amount of cache misses.

This type of filter takes two cycles per sample point to run (one to read the original image sample and one to read the corresponding filter weight) so at 200MHz will have an output fragment rate of 6.25M per texture pipe. This is not blindingly fast compared to normal bilinear filtering but it doesn't cost any gates to do and is fast enough for video or photoshop applications.

```
// To look up the appropriate weight the uv distance from the sample point
// to the centre sample point (in the range -2...+2)is scaled and biased.

TextureSample4x4:

// Calculate sample point from plane equation.
reg[s] = MAdd (dPdx[@S], x, dPdy[@S], y);
```

```
reg[t] = MAdd (dPdx[@T], x, dPdy[@T], y);

// Add in the origin value and scale the coordinates to be in texture map
// units.
reg[s] = Add (reg[S], PStart[@S], planeScale[@S]);
reg[t] = Add (reg[T], PStart[@T], planeScale[@T]);

// Run shading program to reset texture accumulators.
// TexID = 0, destReg = 0, run 'first' program.
SendCommand (Nop, 0, 0, loadShade, noFeedback, firstProg);

// Calculate the distance from the centre sample point to the origin sample
// point.
reg[ds] = Fract (reg[s]);
reg[dt] = Fract (reg[t]);

// Move s and t back to the origin of the 4x4 kernel and force the fraction
// part to be reset so the nearest sampling will always sample the lower
// coordinate.  We may need to add small epsilon to prevent rounding
// errors, or FloatToInt followed by IntToFloat.
reg[s] = Sub (reg[s], 1.0);
reg[t] = Sub (reg[t], 1.0);
reg[s] = Sub (reg[s], reg[ds]);
reg[t] = Sub (reg[t], reg[dt]);

reg[sSaved] = Pass (reg[s]);
reg[dsSaved] = Pass (reg[ds]);

// Visit the 16 neighbouring pixels and calculate the weights
loopCount[0] = 4;
loopCount[1] = 4;

loop:

Call ProcessSample;

reg[s] = Add (reg[s], One);
reg[ds] = Add (reg[ds], One),

// Dec loop counter 0 and jump if not zero to loop.
DJNZ (0, loop);

// If here then we have finished a row so reset the s values for the next
// row.
reg[s] = Pass (reg[sSaved]);
reg[ds] = Pass (reg[dsSaved]);
loopCount[0] = 4;
```

```
// Move on to the next row
reg[t] = Add (reg[t], One);
reg[dt] = Add (reg[dt], One);

// Dec loop counter 0 and jump if not zero to loop.
DJNZ (1, loop);

// Scale the result and output.
SendCommand (Nop, 0, 0, noLoadShade, noFeedback, lastProg),
Done;

--------------------------------
// Process sample subroutine.  Using s, t, ds and dt two textures will be
// read.  The first one is a pixel from
// the image being filtered, and the second one is the corresponding weight
// from the filter kernel texture map.
ProcessSample:

output[0] = Wrap (reg[s], One);
output[1] = Wrap (reg[t], One);

// Store texture sample in destReg 0. Don't run a program
// until the weight is also loaded.
SendCommand (FilterTexture, 0, 0, loadShade, noFeedback, defaultProg);

// Compute the weight texture map address.  The ds and dt values are the
// coordinates, but in the
// range 0…4 so these are scaled to be in the 0…1 range expected.
// the wrap where a scale of 6 will be used (log2 (map size) - log2(4)).
reg[filterS] = Scale (reg[ds], -2);
reg[filterT] = Scale (reg[dt], -2);

output[0] = Wrap (reg[filterS], One);
output[1] = Wrap (reg[filterT], One),
SendCommand (FilterTexture, 1, 1, loadShade, noFeedback, middleProg);
return;
```

10.3.3 Ray Casting

Ray casting is a common operation when voxel processing. Without going into too much detail about how and where ray casting is used the basic algorithm is to send a ray into the 2D or 3D environment and walk along it querying the environment. Based on what if found or on some accumulation along the ray a colour is calculated, and/or the ray terminated.

The ray vector is just a texture coordinate (two or three D). Probing the environment is a 2D or 3D texture map access. The data can be returned to the Texture Coordinate Unit to play some part in terminating the ray (and hence program if all fragments reach this

condition) and/or passed to the Shading Unit for the colour computation. Walking along the ray is an addition to the texture coordinates.

10.3.4 Bump Mapping

Bump mapping uses a gradient field[8] stored in one texture map to perturb the texture coordinates used to access a second texture map. The second texture coordinate set and texture map is typically set up to return the specular colour and the perturbations provided by the first texture map gives the surface a wrinkled or bumpy look.

The gradient field needs to be aligned or rotated with the texture coordinate (representing the surface normal) before it can perturb it and this is done using a matrix multiplication. The matrix can be held constant over the triangle or it can be varied to give a more realistic look, but at added cost. How these matrices are set up is beyond the scope of this example and we will just assume a fixed matrix (as in DX7).

This example uses the DX7 method where the lod of the second texture map is calculated prior to the coordinates being perturbed (and avoids coping with a non analytically differentiable texture function).

BumpMap

```
// Set up the base addresses (used with @ operator) for texture 0.
// Lod is calculated but not used by Texture Index - could avoid
// calculating it but this uses existing code for compactness. Bump maps
// don't work well with mip map filtering.
PlaneBaseReg= 0,
GRegBaseReg = 0,
Call (TextureWithLod);

// TexID = 0, destReg = 0, no program run (not end of tile).
SendCommand (FilterTexture, 0, 0, noLoadShade, feedback, defaultProg);

// Second texture.
PlaneBaseReg= 4,
GRegBaseReg = 4,
Call (TextureWithBump);

// TexID = 1, destReg = 0, run default program (now end of tile).
SendCommand (FilterTexture, 1, 0, loadShade, noFeedback, defaultProg),
Done;

-----------------------------------------------
TextureWithBump:

// Calculate sample point from plane equation for the texture coordinate we
// are going to perturb. We do this while we are waiting for the feedback
// data to come back.
```

[8] The original paper by Blinn used a height field and derived the local gradients as needed, but this step can be avoided by storing the gradient information in the texture map.

```
reg[Q] = MAdd (dPdx[@Q], x, dPdy[@Q], y), savedp;
reg[S] = MAdd (dPdx[@S], x, dPdy[@S], y);
reg[T] = MAdd (dPdx[@T], x, dPdy[@T], y);

// Add in the origin value
reg[Q] = Add (reg[Q], PStart[@Q]);
reg[S] = Add (reg[S], PStart[@S]);
reg[T] = Add (reg[T], PStart[@T]);

DivResult = Div (reg[%Q]);

// Calculate the level of detail - see earlier for more comments.
lod.Load = MSub (dPdx[@S], reg[Q], dxSaved, reg[S], planeScale[@S]);
lod.MergeMax = MSub (dPdx[@T], reg[Q], dxSaved, reg[T], planeScale[@T]);
lod.MergeMax = MSub (dPdy[@S], reg[Q], dySaved, reg[S], planeScale[@S]);
lod.MergeMax = MSub (dPdy[@T], reg[Q], dySaved, reg[T], planeScale[@T]);

// Do the perspective division.
reg[s] = Mult (DivResult, reg[S]);
reg[t] = Mult (DivResult, reg[T]);

output[3] = lod;

// We will stall here waiting for all the feedback data to be returned, but // we have managed to do
        approximately 16 cycles of useful work.
WaitForFeedback;

// Read and store the two perturbation values in registers as floating
// point numbers.
reg[dx] = IntToFloat (feedback (Byte, 0, Signed));
reg[dy] = IntToFloat (feedback (Byte, 1, Signed))
FinishedWithFeedbackData;

// Transform input values using a matrix held in the global registers.
// Note to (mat00, mat01) and (mat10, mat11) are stored in the same 64 bit
// word so they can be accessed simultaneously.
reg[dxt] = MultAdd (reg[dx], GReg[mat00], reg[dy], GReg[mat01]);
reg[dyt] = MultAdd (reg[dx], GReg[mat10], reg[dy], GReg[mat11]);
reg[dxt] = Add (reg[dxt], GReg[mat20]);
reg[dyt] = Add (reg[dyt], GReg[mat21]);

// Add in transformed and scaled perturbation to texture coordinates.
reg[s] = Add (reg[s], reg[dxt]);
reg[t] = Add (reg[t], reg[dyt]);

// If we were worried about high repeat counts we could have multiplied the
// perturbation by Q, added it to S and T and then used the Wrap
```

```
// instruction to do the multiply by the reciprocal.
output[0] = Wrap (reg[s], one);
output[1] = Wrap (reg[t], one),
return;
```

10.4 Pixel applications

Pixel Unit applications include gradient fills with or without texture co-ordinate application, convolution, multipass with Pixel address, multibuffers, Game of Life etc.

11

Glossary & Index

PAU	Pixel Address Unit
TCU	Texture Coordinate Unit

INDEX