Intel® OpenSource HD Graphics PRM

Volume 2 Part 1: *3D/Media – 3D Pipeline (Electronic Only)*

For the all new 2010 Intel Core Processor Family Programmer's Reference Manual (PRM)

March 2010

Revision 1.0

Doc Ref #: IHD_OS_V2Pt1_3_10

Creative Commons License

You are free:

to Share — to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

No Derivative Works. You may not alter, transform, or build upon this work.

Revision History

Document Number	Revision Number De	scription	Revision Date
IHD_OS_V2Pt1_3_10	1.0.	First Release.	March 2010

Contents

1. Introduction

This Programmer's Reference Manual (PRM) describes the architectural behavior and programming environment of the Havendale/Auburndale chipset family, Intel® 965 Chipset family and Intel® G35 Express Chipset GMCH graphics devices (see Table 1-1). The GMCH's Graphics Controller (GC) contains an extensive set of registers and instructions for configuration, 2D, 3D, and Video systems. The PRM describes the register, instruction, and memory interfaces and the device behaviors as controlled and observed through those interfaces. The PRM also describes the registers and instructions and provides detailed bit/field descriptions.

The Programmer's Reference Manual is organized into five volumes:

PRM, Volume 1: Graphics Core

Volume 1, Part 1, 2, 3, 4 and 5 covers the overall Graphics Processing Unit (GPU), without much detail on 3D, Media, or the core subsystem. Topics include the command streamer, context switching, and memory access (including tiling). The Memory Data Formats can also be found in this volume.

The volume also contains a chapter on the Graphics Processing Engine (GPE). The GPE is a collective term for 3D, Media, the subsystem, and the parts of the memory interface that are used by these units. Display, blitter and their memory interfaces are *not* included in the GPE.

PRM, Volume 2: 3D/Media

Volume 2, Part 1, 2 covers the 3D and Media pipelines in detail. This volume is where details for all of the "fixed functions" are covered, including commands processed by the pipelines, fixed-function state structures, and a definition of the inputs (payloads) and outputs of the threads spawned by these units.

This volume also covers the single Media Fixed Function, VLD. It describes how to initiate generic threads using the thread spawner (TS). It is generic threads which will be used for doing the majority of media functions. Programmable kernels will handle the algorithms for media functions such IDCT, Motion Compensation, and even Motion Estimation (used for encoding MPEG streams).

PRM, Volume 3: Display Registers

Volume 3, Parts 1, 2, 3 describes the control registers for the display. The overlay registers and VGA registers are also cover in this volume.

PRM, Volume 4: Subsystem and Cores

Volume 4, Part 1 and 2 describes the GMCH programmable cores, or EUs, and the "shared functions", which are shared by more than one EU and perform functions such as I/O and complex math functions.

The shared functions consist of the sampler, [**Pre-DevSNB**]: extended math unit, data port (the interface to memory for 3D and media), Unified Return Buffer (URB), and the Message Gateway which is used by EU threads to signal each other. The EUs use messages to send data to and receive data from the subsystem; the messages are described along with the shared functions although the generic message send EU instruction is described with the rest of the instructions in the Instruction Set Architecture (ISA) chapters.

This latter part of this volume describes the GMCH core, or EU, and the associated instructions that are used to program it. The instruction descriptions make up what is referred to as an Instruction Set Architecture, or ISA. The ISA describes all of the instructions that the GMCH core can execute, along with the registers that are used to store local data.

Device Tags and Chipsets

Device "Tags" are used in various parts of this document as aliases for the device names/steppings, as listed in the following table. Note that stepping info is sometimes appended to the device tag, e.g., [**DevBW-C**]. Information without any device tagging is applicable to all devices/steppings.

Table 1-1. Supported Chipsets

Chipset Family Name	Device Name	Device Tag
Intel® Q965 Chipset Intel® Q963 Chipset Intel® G965 Chipset	82Q965 GMCH 82Q963 GMCH 82G965 GMCH	[**DevBW**]
Intel® G35 Chipset	82G35 GMCH	[**DevBW-E**]
Intel® GM965 Chipset Intel® GME965 Chipset	GM965 GMCH GME965 GMCH	[**DevCL**]
Mobile Intel® GME965 Express Chipset Mobile Intel® GM965 Express Chipset Mobile Intel® PM965 Express Chipset Mobile Intel® GL960 Express Chipset		[**DevCL**]
[Cantiga A-step (not productized)]		[**DevCTG**], **[DevCTG-A]**
[Cantiga B-step/Eaglelake converged core (not productized)]		**[DevCTG-B],**
[Havendale/Auburndale]		**[DevILK]**

NOTES:

1. Unless otherwise specified, the information in this document applies to all of the devices mentioned in Table 1-1. For Information that does not apply to all devices, the Device Tag is used.
2. Throughout the PRM, references to "All" in a project field refters to all devices in Table 1-1.
3. Throughout the PRM, references to [DevBW] apply to both [DevBW] and [DevBW-E]. [DevBW-E] is referenced specifically for information that is [DevBW] only.
4. Stepping info is sometimes appended to the device tag (e.g., [DevBW-C]). Information without any device tagging is applicable to all devices/steppings.
5. A shorthand is used to (a) identify all devices/steppings prior to the device/stepping that the item pertains (e.g., "[Pre-DevSNB"], and (b) identify all devices/steppings following the device/stepping to which the item pertains."

1.1 Notations and Conventions

1.1.1 Reserved Bits and Software Compatibility

In many register, instruction and memory layout descriptions, certain bits are marked as “Reserved”. When bits are marked as reserved, it is essential for compatibility with future devices that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:
Do not depend on the states of any reserved bits when testing values of registers that contain such bits. Mask out the reserved bits before testing. Do not depend on the states of any reserved bits when storing to instruction or to a register.
When loading a register or formatting an instruction, always load the reserved bits with the values indicated in the documentation, if any, or reload them with the values previously read from the register.

1.2 Terminology

Term	Abbr.	Definition
3D Pipeline	--	One of the two pipelines supported in the GPE. The 3D pipeline is a set of fixed-function units arranged in a pipelined fashion, which process 3D-related commands by spawning EU threads. Typically this processing includes rendering primitives. See *3D Pipeline*.
Adjacency	--	One can consider a single line object as existing in a strip of connected lines. The neighboring line objects are called “adjacent objects”, with the non-shared endpoints called the “adjacent vertices.” The same concept can be applied to a single triangle object, considering it as existing in a mesh of connected triangles. Each triangle shares edges with three other adjacent triangles, each defined by an non-shared adjacent vertex. Knowledge of these adjacent objects/vertices is required by some object processing algorithms (e.g., silhouette edge detection). See *3D Pipeline*.
Application IP	AIP	Application Instruction Pointer. This is part of the control registers for exception handling for a thread. Upon an exception, hardware moves the current IP into this register and then jumps to SIP.
Architectural Register File	ARF	A collection of architecturally visible registers for a thread such as address registers, accumulator, flags, notification registers, IP, null, etc. ARF should not be mistaken as just the address registers.

Term	Abbr.	Definition
Array of Cores	--	Refers to a group of Gen4 EUs, which are physically organized in two or more rows. The fact that the EUs are arranged in an array is (to a great extent) transparent to CPU software or EU kernels.
Binding Table	--	Memory-resident list of pointers to surface state blocks (also in memory).
Binding Table Pointer	BTP	Pointer to a binding table, specified as an offset from the Surface State Base Address register.
Bypass Mode	--	Mode where a given fixed function unit is disabled and forwards data down the pipeline unchanged. Not supported by all FF units.
Byte	B	A numerical data type of 8 bits, B represents a signed byte integer.
Child Thread		A branch-node or a leaf-node thread that is created by another thread. It is a kind of thread associated with the media fixed function pipeline. A child thread is originated from a thread (the parent) executing on an EU and forwarded to the Thread Dispatcher by the TS unit. A child thread may or may not have child threads depending on whether it is a branch-node or a leaf-node thread. All pre-allocated resources such as URB and scratch memory for a child thread are managed by its parent thread.
Clip Space	--	A 4-dimensional coordinate system within which a clipping frustum is defined. Object positions are projected from Clip Space to NDC space via “perspecitive divide” by the W coordinate, and then viewport mapped into Screen Space
Clipper	--	3D fixed function unit that removes invisible portions of the drawing sequence by discarding (culling) primitives or by “replacing” primitives with one or more primitives that replicate only the visible portion of the original primitive.
Color Calculator	CC	Part of the Data Port shared function, the color calculator performs fixed-function pixel operations (e.g., blending) prior to writing a result pixel into the render cache.
Command	--	Directive fetched from a ring buffer in memory by the Command Streamer and routed down a pipeline. Should not be confused with instructions which are fetched by the instruction cache subsystem and executed on an EU.
Command Streamer	CS or CSI	Functional unit of the Graphics Processing Engine that fetches commands, parses them and routes them to the appropriate pipeline.
Constant URB Entry	CURBE	A UE that contains “constant” data for use by various stages of the pipeline.
Control Register	CR	The read-write registers are used for thread mode control and exception handling for a thread.
Degenerate Object	--	Object that is invisible due to coincident vertices or because does not intersect any sample points (usually due to being tiny or a very thin sliver).
Destination	--	Describes an output or write operand.
Destination Size		The number of data elements in the destination of a Gen4 SIMD instruction.
Destination Width		The size of each of (possibly) many elements of the destination of a Gen4 SIMD instruction.
Double Quad word (DQword)	DQ	A fundamental data type, DQ represents 16 bytes.
Double word (DWord)	D or DW	A fundamental data type, D or DW represents 4 bytes.

Term	Abbr.	Definition
Drawing Rectangle	--	A screen-space rectangle within which 3D primitives are rendered. An objects screen-space positions are relative to the Drawing Rectangle origin. See *Strips and Fans*.
End of Block	EOB	A 1-bit flag in the non-zero DCT coefficient data structure indicating the end of an 8x8 block in a DCT coefficient data buffer.
End Of Thread	EOT	a message sideband signal on the Output message bus signifying that the message requester thread is terminated. A thread must have at least one SEND instruction with the EOT bit in the message descriptor field set in order to properly terminate.
Exception	--	Type of (normally rare) interruption to EU execution of a thread's instructions. An exception occurrence causes the EU thread to begin executing the System Routine which is designed to handle exceptions.
Execution Channel	--	
Execution Size	ExecSize	Execution Size indicates the number of data elements processed by a GEN4 SIMD instruction. It is one of the GEN4 instruction fields and can be changed per instruction.
Execution Unit	EU	Execution Unit. An EU is a multi-threaded processor within the GEN4 multi-processor system. Each EU is a fully-capable processor containing instruction fetch and decode, register files, source operand swizzle and SIMD ALU, etc. An EU is also referred to as a GEN4 Core.
Execution Unit Identifier	EUID	The 4-bit field within a thread state register (SR0) that identifies the row and column location of the EU a thread is located. A thread can be uniquely identified by the EUID and TID.
Execution Width	ExecWidth	The width of each of several data elements that may be processed by a single Gen4 SIMD instruction.
Extended Math Unit	EM	A Shared Function that performs more complex math operations on behalf of several EUs.
FF Unit	--	A Fixed-Function Unit is the hardware component of a 3D Pipeline Stage. A FF Unit typically has a unique FF ID associated with it.
Fixed Function	FF	Function of the pipeline that is performed by dedicated (vs. programmable) hardware.
Fixed Function ID	FFID	Unique identifier for a fixed function unit.
FLT_MAX	fmax	The magnitude of the maximum representable single precision floating number according to IEEE-754 standard. FLT_MAX has an exponent of 0xFE and a mantissa of all one's.
Gateway	GW	See Message Gateway.
GEN4 Core		Alternative name for an EU in the GEN4 multi-processor system.
General Register File	GRF	Large read/write register file shared by all the EUs for operand sources and destinations. This is the most commonly used read-write register space organized as an array of 256-bit registers for a thread.
General State Base Address	--	The Graphics Address of a block of memory-resident "state data", which includes state blocks, scratch space, constant buffers and kernel programs. The contents of this memory block are referenced via offsets from the contents of the General State Base Address register. See *Graphics Processing Engine*.

Term	Abbr.	Definition
Graphics Address		The GPE virtual address of some memory-resident object. This virtual address gets mapped by a GTT or PGTT to a physical memory address. Note that many memory-resident objects are referenced not with Graphics Addresses, but instead with offsets from a “base address register”.
Graphics Processing Engine	GPE	Collective name for the Subsystem, the 3D and Media pipelines, and the Command Streamer.
Guardband	GB	Region that may be clipped against to make sure objects do not exceed the limitations of the renderer’s coordinate space.
Horizontal Stride	HorzStride	The distance in element-sized units between adjacent elements of a Gen4 region-based GRF access.
Immediate floating point vector	VF	A numerical data type of 32 bits, an immediate floating point vector of type VF contains 4 floating point elements with 8-bit each. The 8-bit floating point element contains a sign field, a 3-bit exponent field and a 4-bit mantissa field. It may be used to specify the type of an immediate operand in an instruction.
Immediate integer vector	V	A numerical data type of 32 bits, an immediate integer vector of type V contains 8 signed integer elements with 4-bit each. The 4-bit integer element is in 2’s compliment form. It may be used to specify the type of an immediate operand in an instruction.
Index Buffer	IB	Buffer in memory containing vertex indices.
In-loop Deblocking Filter	ILDB	The deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe.
Instance		In the context of the VF unit, an instance is one of a sequence of sets of similar primitive data. Each set has identical vertex data but may have unique instance data that differentiates it from other sets in the sequence.
Instruction	--	Data in memory directing an EU operation. Instructions are fetched from memory, stored in a cache and executed on one or more Gen4 cores. Not to be confused with commands which are fetched and parsed by the command streamer and dispatched down the 3D or Media pipeline.
Instruction Pointer	IP	The address (really an offset) of the instruction currently being fetched by an EU. Each EU has its own IP.
Instruction Set Architecture	ISA	The GEN4 ISA describes the instructions supported by a GEN4 EU.
Instruction State Cache	ISC	On-chip memory that holds recently-used instructions and state variable values.
Interface Descriptor	--	Media analog of a State Descriptor.
Intermediate Z	IZ	Completion of the Z (depth) test at the front end of the Windower/Masker unit when certain conditions are met (no alpha, no pixel-shader computed Z values, etc.)
Inverse Discrete Cosine Transform	IDCT	the stage in the video decoding pipe between IQ and MC
Inverse Quantization	IQ	A stage in the video decoding pipe between IS and IDCT.

Term	Abbr.	Definition
Inverse Scan	IS	A stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS for MPEG-2.
Jitter		Just-in-time compiler.
Kernel	--	A sequence of Gen4 instructions that is logically part of the driver or generated by the jitter. Differentiated from a Shader which is an application supplied program that is translated by the jitter to Gen4 instructions.
Least Significant Bit	LSB	
MathBox	--	See Extended Math Unit
Media	--	Term for operations that are normally performed by the Media pipeline.
Media Pipeline	--	Fixed function stages dedicated to media and “generic” processing, sometimes referred to as the generic pipeline.
Message	--	Messages are data packages transmitted from a thread to another thread, another shared function or another fixed function. Message passing is the primary communication mechanism of GEN4 architecture.
Message Gateway	--	Shared function that enables thread-to-thread message communication/synchronization used solely by the Media pipeline.
Message Register File	MRF	Write-only registers used by EUs to assemble messages prior to sending and as the operand of a send instruction.
Most Significant Bit	MSB	
Motion Compensation	MC	Part of the video decoding pipe.
Motion Picture Expert Group	MPEG	MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc.
Motion Vector Field Selection	MVFS	A four-bit field selecting reference fields for the motion vectors of the current macroblock.
Multi Render Targets	MRT	Multiple independent surfaces that may be the target of a sequence of 3D or Media commands that use the same surface state.
Normalized Device Coordinates	NDC	Clip Space Coordinates that have been divided by the Clip Space “W” component.
Object	--	A single triangle, line or point.
Open GL	OGL	A Graphics API specification associated with Linux.
Parent Thread	--	A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Pipeline Stage	--	A abstracted element of the 3D pipeline, providing functions performed by a combination of the corresponding hardware FF unit and the threads spawned by that FF unit.
Pipelined State Pointers	PSP	Pointers to state blocks in memory that are passed down the pipeline.

Term	Abbr.	Definition
Pixel Shader	PS	Shader that is supplied by the application, translated by the jitter and is dispatched to the EU by the Windower (conceptually) once per pixel.
Point	--	A drawing object characterized only by position coordinates and width.
Primitive	--	Synonym for object: triangle, rectangle, line or point.
Primitive Topology	--	A composite primitive such as a triangle strip, or line list. Also includes the objects triangle, line and point as degenerate cases.
Provoking Vertex	--	The vertex of a primitive topology from which vertex attributes that are constant across the primitive are taken.
Quad Quad word (QQword)	QQ	A fundamental data type, QQ represents 32 bytes.
Quad Word (QWord)	QW	A fundamental data type, QW represents 8 bytes.
Rasterization		Conversion of an object represented by vertices into the set of pixels that make up the object.
Region-based addressing	--	Collective term for the register addressing modes available in the EU instruction set that permit discontiguous register data to be fetched and used as a single operand.
Render Cache	RC	Cache in which pixel color and depth information is written prior to being written to memory, and where prior pixel destination attributes are read in preparation for blending and Z test.
Render Target	RT	A destination surface in memory where render results are written.
Render Target Array Index	--	Selector of which of several render targets the current operation is targeting.
Root Thread	--	A root-node thread. A thread corresponds to a root-node in a thread generation hierarchy. It is a kind of thread associated with the media fixed function pipeline. A root thread is originated from the VFE unit and forwarded to the Thread Dispatcher by the TS unit. A root thread may or may not have child threads. A root thread may have scratch memory managed by TS. A root thread with children has its URB resource managed by the VFE.
Sampler	--	Shared function that samples textures and reads data from buffers on behalf of EU programs.
Scratch Space	--	Memory allocated to the subsystem that is used by EU threads for data storage that exceeds their register allocation, persistent storage, storage of mask stack entries beyond the first 16, etc.
Shader	--	A Gen4 program that is supplied by the application in a high level shader language, and translated to Gen4 instructions by the jitter.
Shared Function	SF	Function unit that is shared by EUs. EUs send messages to shared functions; they consume the data and may return a result. The Sampler, Data Port and Extended Math unit are all shared functions.
Shared Function ID	SFID	Unique identifier used by kernels and shaders to target shared functions and to identify their returned messages.
Single Instruction Multiple Data	SIMD	The term SIMD can be used to describe the kind of parallel processing architecture that exploits data parallelism at instruction level. It can also be used to describe the instructions in such architecture.

Term	Abbr.	Definition
Source	--	Describes an input or read operand
Spawn	--	To initiate a thread for execution on an EU. Done by the thread spawner as well as most FF units in the 3D pipeline.
Sprite Point	--	Point object using full range texture coordinates. Points that are not sprite points use the texture coordinates of the point's center across the entire point object.
State Descriptor	--	Blocks in memory that describe the state associated with a particular FF, including its associated kernel pointer, kernel resource allowances, and a pointer to its surface state.
State Register	SR	The read-only registers containing the state information of the current thread, including the EUID/TID, Dispatcher Mask, and System IP.
State Variable	SV	An individual state element that can be varied to change the way given primitives are rendered or media objects processed. On Gen4 state variables persist only in memory and are cached as needed by rendering/processing operations except for a small amount of non-pipelined state.
Stream Output	--	A term for writing the output of a FF unit directly to a memory buffer instead of, or in addition to, the output passing to the next FF unit in the pipeline. Currently only supported for the Geometry Shader (GS) FF unit.
Strips and Fans	SF	Fixed function unit whose main function is to decompose primitive topologies such as strips and fans into primitives or objects.
Sub-Register		Subfield of a SIMD register. A SIMD register is an aligned fixed size register for a register file or a register type. For example, a GRF register, *r2*, is 256-bit wide, 256-bit aligned register. A sub-register, *r2.3:d*, is the fourth dword of GRF register *r2*.
Subsystem	--	The Gen4 name given to the resources shared by the FF units, including shared functions and EUs.
Surface	--	A rendering operand or destination, including textures, buffers, and render targets.
Surface State	--	State associated with a render surface including
Surface State Base Pointer	--	Base address used when referencing binding table and surface state data.
Synchronized Root Thread	--	A root thread that is dispatched by TS upon a 'dispatch root thread' message.
System IP	SIP	There is one global System IP register for all the threads. From a thread's point of view, this is a virtual read only register. Upon an exception, hardware performs some bookkeeping and then jumps to SIP.
System Routine	--	Sequence of Gen4 instructions that handles exceptions. SIP is programmed to point to this routine, and all threads encountering an exception will call it.
Thread		An instance of a kernel program executed on an EU. The life cycle for a thread starts from the executing the first instruction after being dispatched from Thread Dispatcher to an EU to the execution of the last instruction – a send instruction with EOT that signals the thread termination. Threads in GEN4 system may be independent from each other or communicate with each other through Message Gateway share function.

Term	Abbr.	Definition
Thread Dispatcher	TD	Functional unit that arbitrates thread initiation requests from Fixed Functions units and instantiates the threads on EUs.
Thread Identifier	TID	The field within a thread state register (SR0) that identifies which thread slots on an EU a thread occupies. A thread can be uniquely identified by the EUID and TID.
Thread Payload		Prior to a thread starting execution, some amount of data will be pre-loaded in to the thread's GRF (starting at r0). This data is typically a combination of control information provided by the spawning entity (FF Unit) and data read from the URB.
Thread Spawner	TS	The second and the last fixed function stage of the media pipeline that initiates new threads on behalf of generic/media processing.
Topology		See Primitive Topology.
Unified Return Buffer	URB	The on-chip memory managed/shared by GEN4 Fixed Functions in order for a thread to return data that will be consumed either by a Fixed Function or other threads.
Unsigned Byte integer	UB	A numerical data type of 8 bits.
Unsigned Double Word integer	UD	A numerical data type of 32 bits. It may be used to specify the type of an operand in an instruction.
Unsigned Word integer	UW	A numerical data type of 16 bits. It may be used to specify the type of an operand in an instruction.
Unsynchronized Root Thread	--	A root thread that is automatically dispatched by TS.
URB Dereference	--	
URB Entry	UE	URB Entry: A logical entity stored in the URB (such as a vertex), referenced via a URB Handle.
URB Entry Allocation Size	--	Number of URB entries allocated to a Fixed Function unit.
URB Fence	Fence	Virtual, movable boundaries between the URB regions owned by each FF unit.
URB Handle	--	A unique identifier for a URB entry that is passed down a pipeline.
URB Reference	--	
Variable Length Decode	VLD	The first stage of the video decoding pipe that consists mainly of bit-wide operations. GEN4 supports hardware VLD acceleration in the VFE fixed function stage.
Vertex Buffer	VB	Buffer in memory containing vertex attributes.
Vertex Cache	VC	Cache of Vertex URB Entry (VUE) handles tagged with vertex indices.
Vertex Fetcher	VF	The first FF unit in the 3D pipeline responsible for fetching vertex data from memory. Sometimes referred to as the Vertex Formatter.
Vertex Header	--	Vertex data required for every vertex appearing at the beginning of a Vertex URB Entry.
Vertex ID	--	Unique ID for each vertex that can optionally be included in vertex attribute data sent down the pipeline and used by kernel/shader threads.

Term	Abbr.	Definition
Vertex URB Entry	VUE	A URB entry that contains data for a specific vertex.
Vertical Stride	VertStride	The distance in element-sized units between 2 vertically-adjacent elements of a Gen4 region-based GRF access.
Video Front End	VFE	The first fixed function in the GEN4 generic pipeline; performs fixed-function media operations.
Viewport	VP	
Windower IZ	WIZ	Term for Windower/Masker that encapsulates its early (“intermediate”) depth test function.
Windower/Masker	WM	Fixed function triangle/line rasterizer.
Word	W	A numerical data type of 16 bits, W represents a signed word integer.

2. 3D Pipeline

2.1 Introduction

This section covers the programming details for the 3D fixed functions.

2.2 3D Pipeline Overview

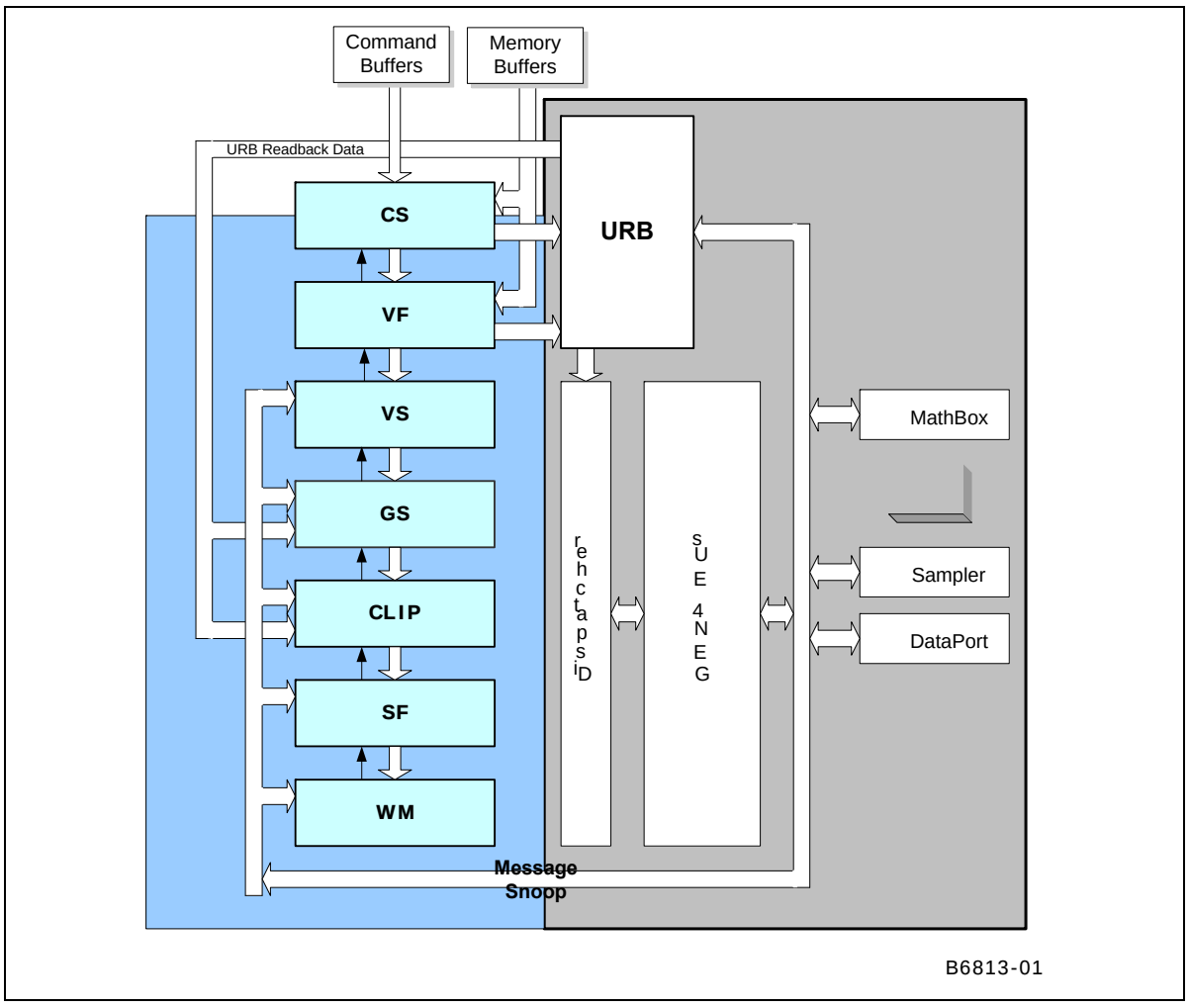


Figure 2-1 3D Pipeline Diagram [Pre-DevSNB]

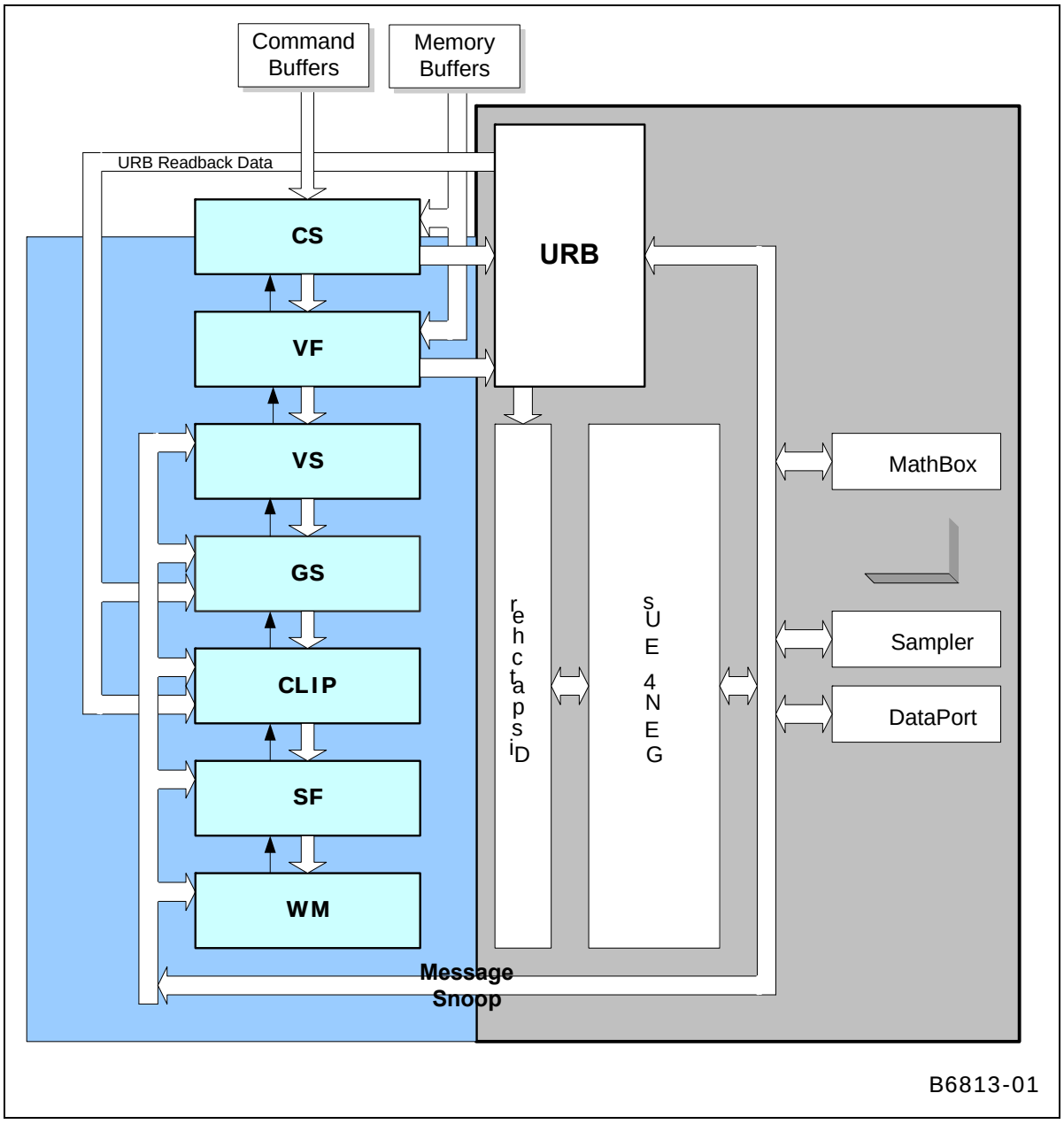


Figure 2-2 3D Pipeline Diagram [Pre-DevSNB]

2.2.1 3D Pipeline Stages

The following table lists the various stages of the 3D pipeline and describes their major functions.

Pipeline Stage	Functions Performed
Command Stream (CS)	The Command Stream stage is responsible for managing the 3D pipeline and passing commands down the pipeline. In addition, the CS unit reads "constant data" from memory buffers and places it in the URB. Note that the CS stage is shared between the 3D and Media pipelines.
Vertex Fetch (VF)	The Vertex Fetch stage, in response to 3D Primitive Processing commands, is responsible for reading vertex data from memory, reformatting it, and writing the results into Vertex URB Entries. It then outputs primitives by passing references to the VUEs down the pipeline.
Vertex Shader (VS)	The Vertex Shader stage is responsible for processing (shading) incoming vertices by passing them to VS threads.
Geometry Shader (GS)	The Geometry Shader stage is responsible for processing incoming objects by passing each object's vertices to a GS thread.
Clipper (CLIP)	The Clipper stage performs clip test on incoming objects and, if required, clips objects via CLIP threads [Pre-DevSNB]
Strip/Fan (SF)	The Strip/Fan stage performs object setup via use of spawned SF threads (aka Setup threads) for [Pre-DevSNB]
Windower/Masker (WM)	The Windower/Masker performs object rasterization and spawns WM thread (aka PS thread) to process (shade) the object pixels.

2.3 3D Primitives Overview

The 3DPRIMITIVE command (defined in the *VF Stage* chapter) is used to submit 3D primitives to be processed by the 3D pipeline. Typically the processing results in the rendering of pixel data into the render targets, but this is not required.

Note: *Terminology Note*: There is considerable confusion surrounding the term 'primitive', e.g., is a triangle strip a 'primitive', or is a triangle within a triangle strip a 'primitive'? In this spec, we will try to avoid ambiguity by using the term 'object' to represent the basic shapes (point, line, triangle), and 'topology' to represent input geometry (strips, lists, etc.). Unfortunately, terms like '3DPRIMITIVE' must remain for legacy reasons.

The following table describes the basic primitive topology types supported in the 3D pipeline.

Notes:

- There are several variants of the basic topologies. These have been introduced to allow slight variations in behavior without requiring a state change.
- Number of vertices:
  - **Dangling Vertices**: Topologies have an "expected" number of vertices in order to form complete objects within the topologies (e.g., LINELIST is expected to have an even number of vertices). The actual number of vertices specified in the 3DPRIMITIVE command, and as output from the GS unit, is allowed to deviate from this expected number --- in which case any "dangling" vertices are discarded. The removal of dangling vertices is initially performed in the VF unit. In order to filter out dangling vertices emitted by GS threads, the CLIP unit also performs dangling-vertex removal at its input. However, the CLIP unit is required to output the expected number ([Pre-DevSNB]: based on the assumption that the clipping kernel is thoroughly validated). [Pre-DevSNB]: If a CLIP thread violates this restriction, behavior is UNDEFINED.

Table 2-1. 3D Primitive Topology Types

3D Primitive Topology Type (ordered alphabetically)	Description
LINELIST	A list of independent line objects (2 vertices per line). **Programming Restrictions:** • Normal usage expects a multiple of 2 vertices, though incomplete objects are silently ignored.
LINELIST_ADJ	A list of independent line objects with adjacency information (4 vertices per line). **Programming Restrictions:** • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread. • Before issuing primitives of this type, if the GS unit is DISABLED, the CLIP unit must be ENABLED to cause adjacent vertices to be removed. The CLIP unit will discard the adjacent vertices and convert the PrimType to the corresponding no-adjacency PrimType.
LINELOOP	Similar to a 3DPRIM_LINESTRIP, though the last vertex is connected back to the initial vertex via a line object. . The LINELOOP topology is converted to LINESTRIP topology at the beginning of the 3D pipeline. **Programming Restrictions:** • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored. (The 2-vertex case is required by OGL). • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).

3D Primitive Topology Type (ordered alphabetically)	Description
LINESTRIP	A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define a connected line object. **Programming Restrictions:** • Normal usage expects at least 2 vertices, though incomplete objects are silently ignored.
LINESTRIP_ADJ	A list of vertices connected such that, after the first vertex, each additional vertex is associated with the previous vertex to define connected line object. The first and last segments are adjacent–only vertices. **Programming Restrictions:** • Normal usage expects at least 4 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread. • Before issuing primitives of this type, if the GS unit is DISABLED, the CLIP unit must be ENABLED to cause adjacent vertices to be removed. The CLIP unit will discard the adjacent vertices and convert the PrimType to the corresponding no-adjacency PrimType.
LINESTRIP_BF	Similar to LINESTRIP, except treated as “backfacing’ during rasterization (stencil test). This can be used to support “line” polygon fill mode when two-sided stencil is enabled.
LINESTRIP_CONT	Similar to LINESTRIP, except LineStipple (if enabled) is continued (vs. reset) at the start of the primitive topology. This can be used to support line stipple when the API-provided primitive is split across multiple tolopologies.
LINESTRIP_CONT_BF	Combination of LINESTRIP_BF and LINESTRIP_CONT variations.
POINTLIST	A list of point objects (1 vertex per point).
POINTLIST_BF	Similar to POINTLIST, except treated as “backfacing’ during rasterization (stencil test). This can be used to support “point” polygon fill mode when two-sided stencil is enabled.
POLYGON	Similar to TRIFAN, though the first vertex always provides the “flat-shaded” values (vs. this being programmable through state). **Programming Restrictions:** • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.

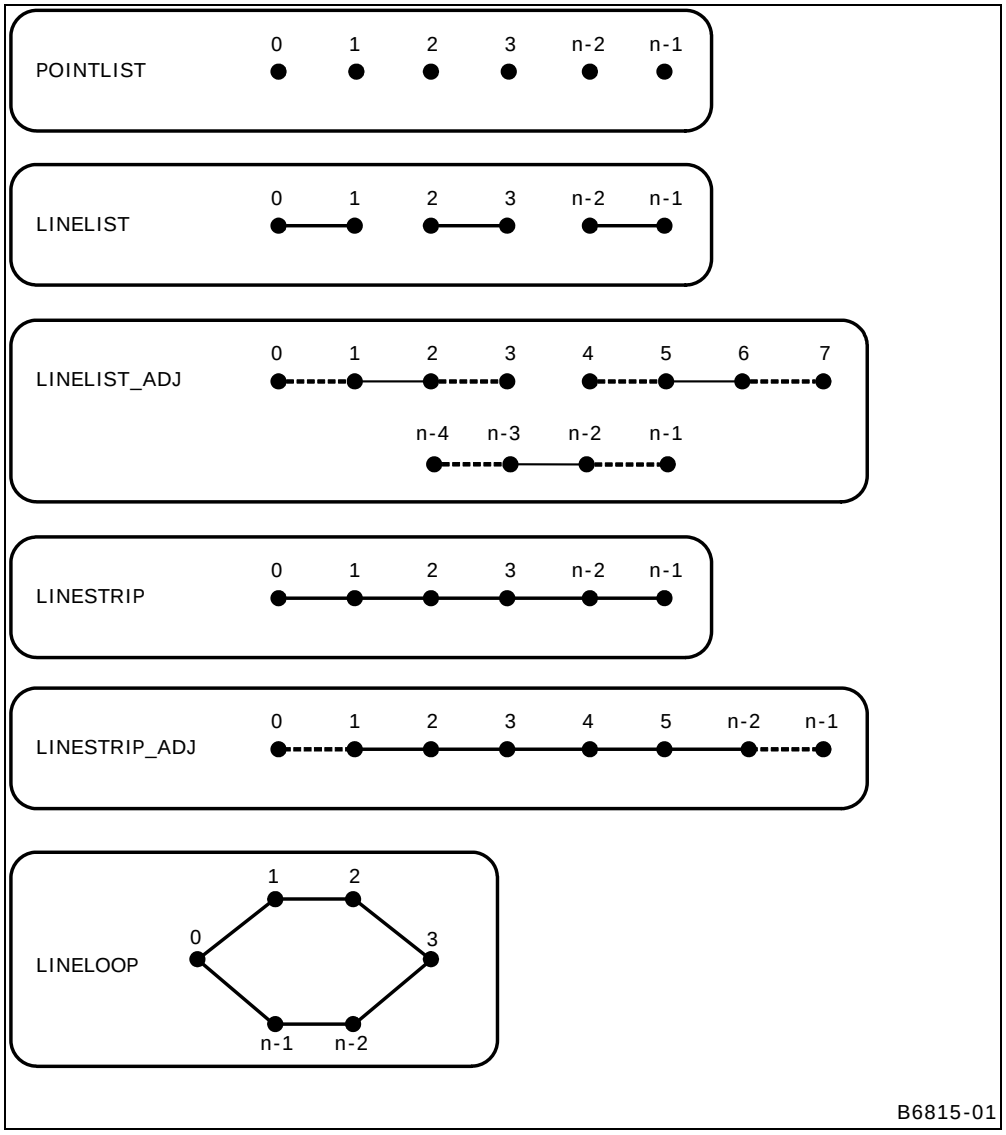
3D Primitive Topology Type (ordered alphabetically)	Description
QUADLIST	A list of independent quad objects (4 vertices per quad). The QUADLIST topology is converted to POLYGON topology at the beginning of the 3D pipeline. **Programming Restrictions:** • Normal usage expects a multiple of 4 vertices, though incomplete objects are silently ignored. • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).
QUADSTRIP	A list of vertices connected such that, after the first two vertices, each additional pair of vertices are associated with the previous two vertices to define a connected quad object. The QUADSTRIP topology is converted to POLYGON topology at the beginning of the 3D pipeline. **Programming Restrictions:** • Normal usage expects an even number (4 or greater) of vertices, though incomplete objects are silently ignored. • Not valid after the GS stage (i.e., must be converted by a GS thread to some other primitive type).

RECTLIST	A list of independent rectangles, where only 3 vertices are provided per rectangle object, with the fourth vertex implied by the definition of a rectangle. V0=LowerRight, V1=LowerLeft, V2=UpperLeft. Implied V3 = V0-V1+V2.
	Programming Restrictions:
	• Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.
	• The RECTLIST primitive is supported specifically for 2D operations (e.g., BLTs and "stretch" BLTs) and not as a general 3D primitive. Due to this, a number of restrictions apply to the use of RECTLIST:
	  • Must utilize "screen space" coordinates (VPOS_SCREENSPACE) when the primitive reaches the CLIP stage. The W component of position must be 1.0 for all vertices. The 3 vertices of each object should specify a screen-aligned rectangle (after the implied vertex is computed).
	  • Clipping: Must not require clipping or rely on the CLIP unit's ClipTest logic to determine if clipping is required. Either the CLIP unit should be DISABLED, or the CLIP unit's **Clip Mode** should be set to a value other than CLIPMODE_NORMAL.
	  • Viewport Mapping must be DISABLED (as is typical with the use of screen-space coordinates).

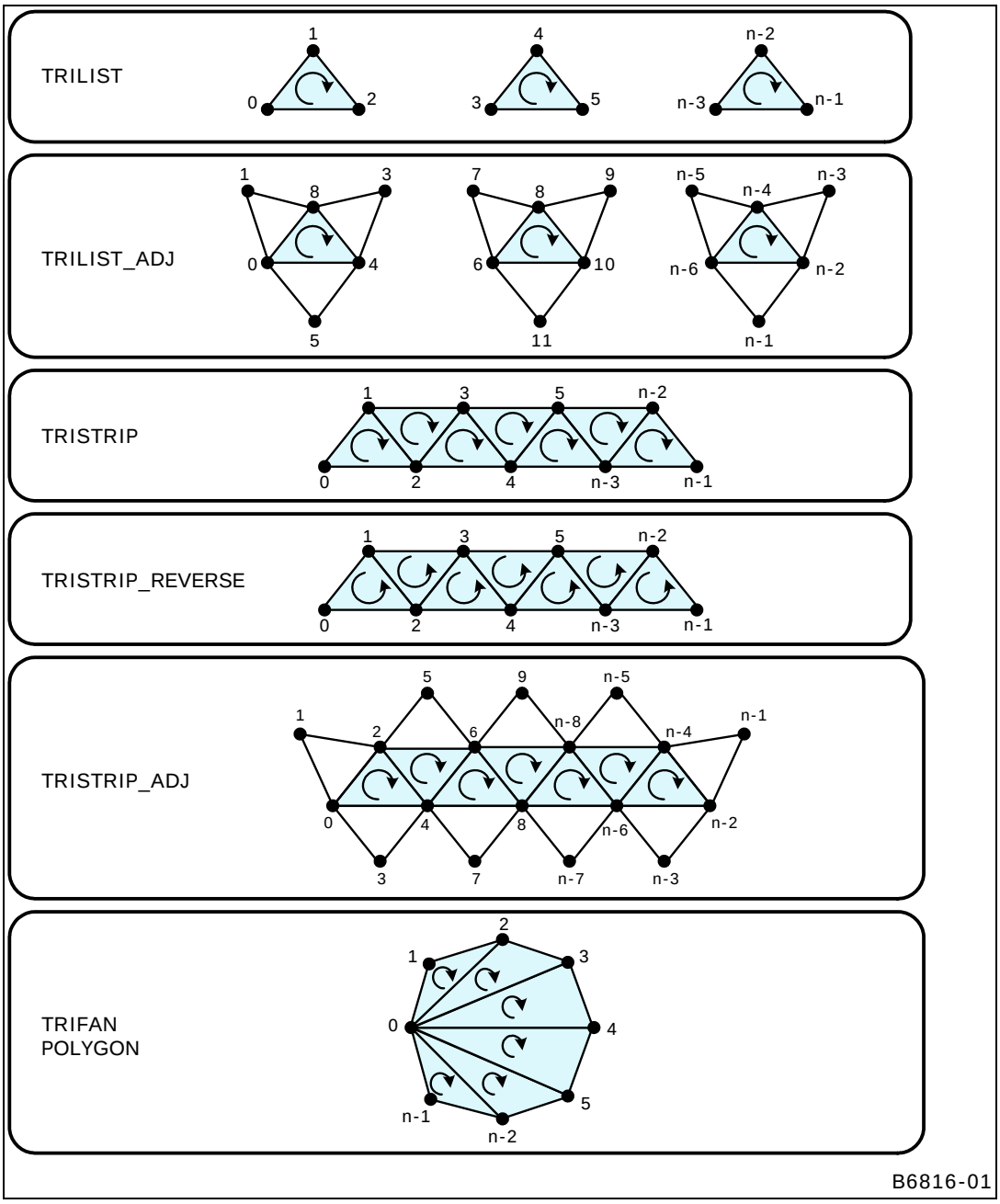
3D Primitive Topology Type (ordered alphabetically)	Description
TRIFAN	Triangle objects arranged in a fan (or polygon). The initial vertex is maintained as a common vertex. After the second vertex, each additional vertex is associated with the previous vertex and the common vertex to define a connected triangle object. Programming Restrictions: • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.
TRIFAN_NOSTIPPLE	Similar to TRIFAN, but poylgon stipple is not applied (even if enabled). This can be used to support “point” polygon fill mode, under the combination of the following conditions: (a) when the frontfacing and backfacing polygon fill modes are different (so the final fill mode is not known to the driver), (b) one of the fill modes is “point” and the other is “solid”, (c) point mode is being emulated by converting the point into a trifan, (d) polygon stipple is enabled. In this case, polygon stipple should not be applied to the points-emulated-as-trifans.
TRILIST	A list of independent triangle objects (3 vertices per triangle). **Programming Restrictions**: • Normal usage expects a multiple of 3 vertices, though incomplete objects are silently ignored.
TRILIST_ADJ	A list of independent triangle objects with adjacency information (6 vertices per triangle). **Programming Restrictions:** • Normal usage expects a multiple of 6 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread. • Before issuing primitives of this type, if the GS unit is DISABLED, the CLIP unit must be ENABLED to cause adjacent vertices to be removed. The CLIP unit will discard the adjacent vertices and convert the PrimType to the corresponding no-adjacency PrimType.
TRISTRIP	A list of vertices connected such that, after the first two vertices, each additional vertex is associated with the last two vertices to define a connected triangle object. **Programming Restrictions:** • Normal usage expects at least 3 vertices, though incomplete objects are silently ignored.

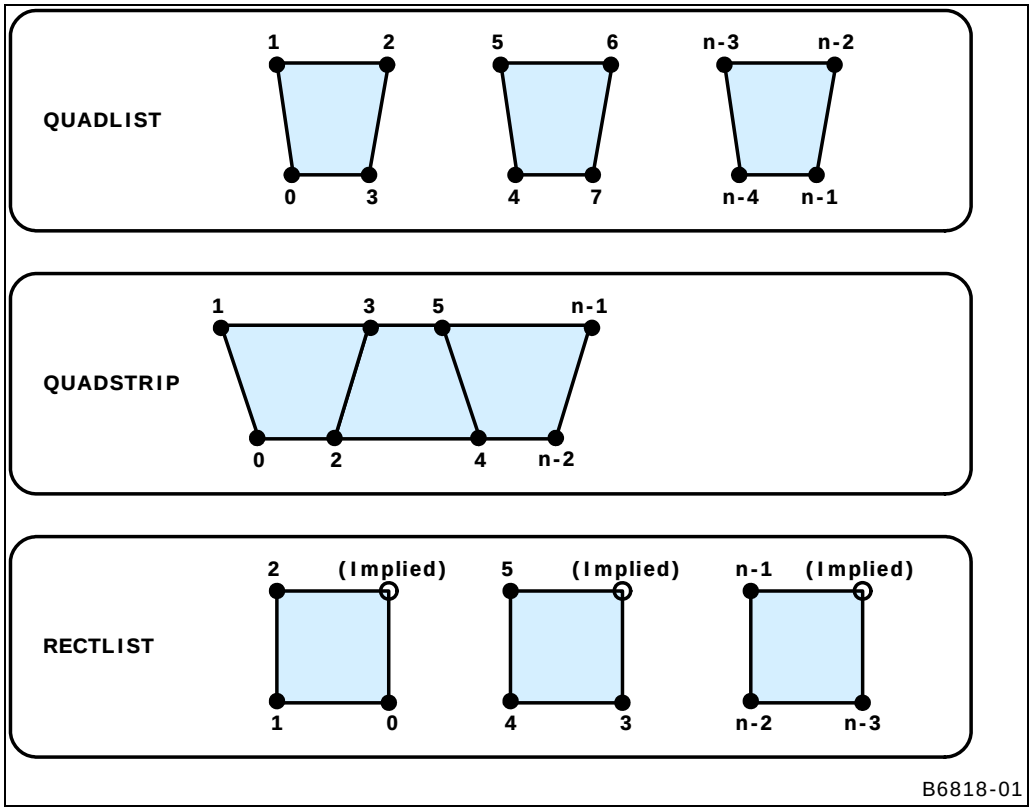
3D Primitive Topology Type (ordered alphabetically)	Description
TRISTRIP_ADJ	A list of vertices where the even-numbered (including 0th) vertices are connected such that, after the first two vertex pairs, each additional even-numbered vertex is associated with the last two even-numbered vertices to define a connected triangle object. The odd-numbered vertices are adjacent-only vertices. **Programming Restrictions:** • Normal usage expects at least 6 vertices, though incomplete objects are silently ignored. • Not valid as output from GS thread. • Before issuing primitives of this type, if the GS unit is DISABLED, the CLIP unit must be ENABLED to cause adjacent vertices to be removed. The CLIP unit will discard the adjacent vertices and convert the PrimType to the corresponding no-adjacency PrimType.
TRISTRIP_REVERSE	Similar to TRISTRIP, though the sense of orientation (winding order) is reversed – this allows SW to break long tristrips into smaller pieces and still maintain correct face orientations.

The following diagrams illustrate the basic 3D primitive topologies. (Variants are not shown if they have the same definition with respect to the information provided in the diagrams).


POINTLIST
0 1 2 3 n-2 n-1
LINELIST
0 1 2 3 n-2 n-1
LINELIST_ADJ
0 1 2 3 4 5 6 7
n-4 n-3 n-2 n-1
LINESTRIP
0 1 2 3 n-2 n-1
LINESTRIP_ADJ
0 1 2 3 4 5 n-2 n-1
LINELOOP
1 2
0 3
n-1 n-2
B6815-01

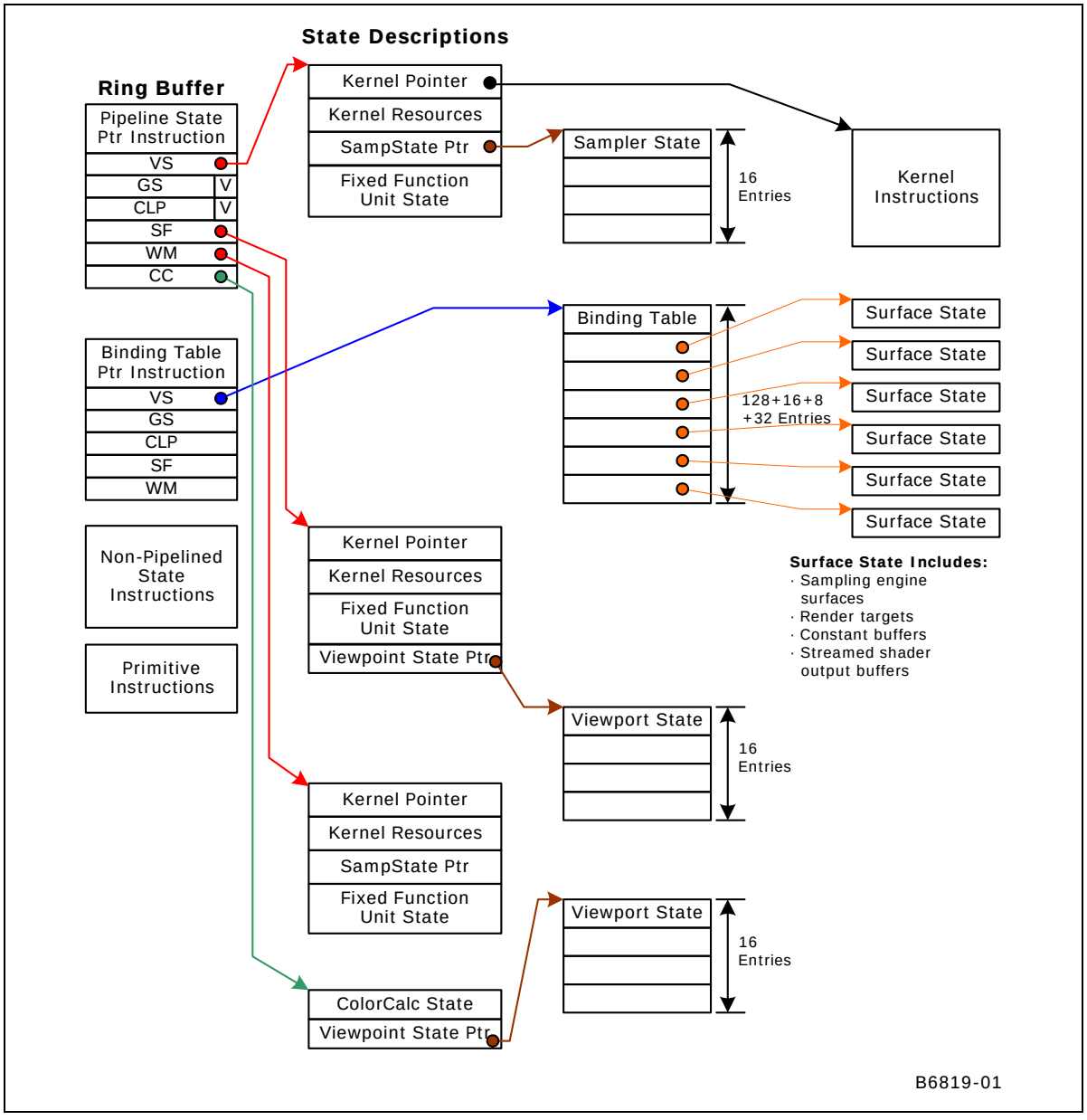
A note on the arrows you see below: These arrows are intended to show the vertex ordering of triangles that are to be considered having “clockwise” winding order in screen space. Effectively, the arrows show the order in which vertices are used in the cross-product (area, determinant) computation. Note that for TRISTRIP, this requires that either the order of odd-numbered triangles be reversed in the cross-product or the sign of the result of the normally-ordered cross-product be flipped (these are identical operations).




QUADLIST
1
2
0
3
5
6
4
7
n-3
n-2
n-4
n-1
QUADSTRIP
1
3
5
n-1
0
2
4
n-2
RECTLIST
2
(Implied)
1
0
5
(Implied)
4
3
n-1
(Implied)
n-2
n-3
B6818-01

2.4 3D Pipeline State Overview

2.4.1.1 [Pre-DevSNB]



2.4.2 3DSTATE_PIPELINED _POINTERS [Pre-DevSNB]

The 3DSTATE_PIPELINED_POINTERS command is used to set up the pointers to the 3D fixed function state. It is also used to disable the GS and/or CLIP units and make them pass-through (input flows through to output). The other units are (by definition) "enabled", meaning they will fetch and use the associated pipelined state to control the unit's functions.

[DevBW-A,B] Errata BWT007: State data pointed at by offsets from **General State Base** must be contained within 32-bit physical address space (that is, must entirely map to memory pages under 4GB.)

[DevILK] A pipeline flush must occur before clearing the **GS Enable** if the **GS Enable** was set on the previous 3DSTATE_PIPELINED_POINTERS.

3DSTATE_PIPELINED_POINTERS

Project:	[Pre-DevSNB]	**Length Bias:**	2

The 3DSTATE_PIPELINED_POINTERS command is used to set up the pointers to the 3D fixed function state. It is also used to disable the GS and/or CLIP units and make them pass-through (input flows through to output). The other units are (by definition) "enabled", meaning they will fetch and use the associated pipelined state to control the unit's functions.

[DevBW-A,B] Errata BWT007: State data pointed at by offsets from **General State Base** must be contained within 32-bit physical address space (that is, must entirely map to memory pages under 4GB.)

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 0h 3DSTATE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 00h 3DSTATE_PIPELINED_POINTERS Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 5h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:5	**Pointer VS_STATE** Project: All Address: GeneralStateOffset[31:5] Surface Type: VS_STATE Specifies the 32-byte aligned offset of the VS_STATE. This offset is relative to the **General State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ

3DSTATE_PIPELINED_POINTERS		
2	31:5	**Pointer to GS_STATE** Project: All Address: GeneralStateOffset[31:5] Surface Type: GS_STATE Specifies the 32-byte aligned offset of the GS_STATE. This offset is relative to the **General State Base Address**.
	4:1	**Reserved** Project: All Format: MBZ
	0	**GS Enable** Project: All Format: Enable FormatDesc Specifies whether the GS function is enabled or disabled (pass-through). If this bit is set to DISABLED, the pointer to GS_STATE is ignored. **Programming Note:** When enabling the GS stage that may generate incomplete objects, the CLIP stage also needs to be ENABLED in order to filter out any incomplete objects. See *Clipper* chapter.
3	31:5	**Pointer to CLIP_STATE** Project: All Address: GeneralStateOffset[31:5] Surface Type: CLIP_STATE Specifies the 32-byte aligned offset of the CLIP_STATE. This offset is relative to the **General State Base Address**.
	4:1	**Reserved** Project: All Format: MBZ
	0	**CLIP Enable** Project: All Format: Enable FormatDesc Specifies whether the CLIP function is enabled or disabled (pass-through). If this bit is set to ENABLED, the pointer to CLIP_STATE is ignored. **Programming Note:** When enabling the GS stage that may generate incomplete objects, the CLIP stage also needs to be ENABLED in order to filter out any incomplete objects. See *Clipper* chapter. A MI_FLUSH needs to be sent prior to disabling Clip function.
4	31:5	**Pointer to SF_STATE** Project: All Address: GeneralStateOffset[31:5] Surface Type: SF_STATE Specifies the 32-byte aligned offset of the SF_STATE. This offset is relative to the **General State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ

3DSTATE_PIPELINED_POINTERS		
5	31:5	**Pointer to WM_STATE** Project: All Address: GeneralStateOffset[31:5] Surface Type: WM_STATE Specifies the 32-byte aligned offset of the WM_STATE. This offset is relative to the **General State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
6	31:6	**Pointer to COLOR_CALC_STATE** Project: All Address: GeneralStateOffset[31:6] Surface Type: COLOR_CALC_STATE Specifies the 64-byte aligned offset of the COLOR_CALC_STATE. This offset is relative to the **General State Base Address**.
	5:0	**Reserved** Project: All Format: MBZ

2.4.3 3DSTATE_BINDI NG_TABLE_POINTERS

3DSTATE_BINDING_TABLE_POINTERS		
Project:	**Length Bias:**	2

The 3DSTATE_BINDING_TABLE_POINTERS command is used to define the location of fixed functions' BINDING_TABLE_STATE. Only some of the fixed functions utilize binding tables.

[DevBW-A,B] Errata BWT007: Surface State data pointed at by offsets from Surface State Base must be contained within 32-bit physical address space (that is, must entirely map to memory pages under 4G.)

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 0h 3DSTATE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 01h 3DSTATE_BINDING_TABLE_POINTERS Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 4h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	4:0	**Reserved** Project: All Format: MBZ
2	31:5	**Pointer to GS Binding Table** Project: All Address: SurfaceStateOffset[31:5] Surface Type: BINDING_TABLE_STATE Specifies the 32-byte aligned address offset of the GS function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
3	31:5	**Pointer to CLIP Binding Table** Project: All Address: SurfaceStateOffset[31:5] Surface Type: BINDING_TABLE_STATE^256 Specifies the 32-byte aligned address offset of the CLIP function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.

3DSTATE_BINDING_TABLE_POINTERS		
	4:0	**Reserved** Project: All Format: MBZ
4	31:5	**Pointer to SF Binding Table** Project: All Address: SurfaceStateOffset[31:5] Surface Type: BINDING_TABLE_STATE^256 Specifies the 32-byte aligned address offset of the SF function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ
5	31:5	**Pointer to PS Binding Table** Project: All Address: SurfaceStateOffset[31:5] Surface Type: BINDING_TABLE_STATE^256 Specifies the 32-byte aligned address offset of the PS (Windower) function's BINDING_TABLE_STATE. This offset is relative to the **Surface State Base Address**.
	4:0	**Reserved** Project: All Format: MBZ

2.5 Vertex Data Overview

The 3D pipeline FF stages (past VF) receive input 3D primitives as a stream of vertex information packets. (These packets are not directly visible to software). Much of the data associated with a vertex is passed indirectly via a VUE handle. The information provided in vertex packets includes:

- The **URB Handle** of the VUE: This is used by the FF unit to refer to the VUE and perform any required operations on it (e.g., cause it to be read into the thread payload, dereference it, etc.).
- **Primitive Topology Information**: This information is used to identify/delineate primitive topologies in the 3D pipeline. Initially, the VF unit supplies this information. GS and CLIP threads must supply this information with each vertex they produce (via the URB_WRITE message). If a FF unit directly outputs vertices (that were not generated by a thread they spawned), that FF unit is responsible for providing this information.
  - **PrimType**: The type of topology, as defined by the corresponding field of the 3DPRIMITIVE command.
  - **StartPrim**: TRUE only for the first vertex of a topology.
  - **EndPrim**: TRUE only for the last vertex of a topology.
- (Possibly, depending on FF unit) Data read back from the **Vertex Header** of the VUE.

2.5.1 Vertex URB Entry (VUE) Formats

In general, vertex data is stored in Vertex URB Entries (VUEs) in the URB, processed by CLIP threads, and only referenced by the pipeline stages indirectly via VUE handles. Therefore (for the most part) the contents/format of the vertex data is not exposed to 3D pipeline hardware – the FF units are typically only aware of the handles and sizes of VUEs.

VUEs are written in two ways:

- At the top of the 3D Geometry pipeline, the VF's InputAssembly function creates VUEs and initializes them from data extracted from Vertex Buffers as well as internally-generated data.
- VS, GS, and CLIP threads can compute, format and write new VUEs as thread output.

There are only two points in the 3D FF pipeline where the FF units are exposed to the VUE data. Otherwise the VUE remains opaque to the 3D pipeline hardware.

- Just prior to the CLIP stage, all VUEs are read-back:
  - [**Pre-DevILK**] Readback of the Vertex Header (first 256 bits of the VUE)
  - [**DevILK**] Readback of the Vertex Header (first 512 bits of the VUE)
  - [**DevILK**] Optional readback of User Clip distances if the User Clip Planes are enabled.
- Just after the CLIP stage, on clip-generated VUEs are read-back:
  - Readback of the Vertex Header (first 256 bits of the VUE)

Software must ensure that any VUEs subject to readback by the 3D pipeline start with a valid Vertex Header. This extends to all VUEs with the following exceptions listed below:

- If the VS function is enabled, the VF-written VUEs are not required to have Vertex Headers, as the VS-incoming vertices are guaranteed to be consumed by the VS (i.e., the VS thread is responsible for overwriting the input vertex data).

- If the GS FF is enabled, neither VF-written VUEs nor VS thread-generated VUEs are required to have Vertex Headers, as the GS will consume all incoming vertices.
- (There is a pathological case where the CLIP state can be programmed (There is a pathological case where the CLIP state can be programmed to guarantee that all CLIP-incoming vertices are consumed – regardless of the data read back prior to the CLIP stage – and therefore only the CLIP thread-generated vertices would require Vertex Headers).

The folllowing table defines the Vertex Header. The Position fields are described in further detail below.

Figure 2-3. VUE Vertex Header ([Pre-Dev-ILK])

DWord	Bit	Description
D0	31:0	Reserved: MBZ
D1	31:11	Reserved: MBZ
	10:0	**Render Target Array Index.** This value is (eventually) used to index into a specific element of an "array" Render Target. It is read back by the GS unit (for all exiting vertices) and the Clip unit (for all clip-generated vertices), subsequently routed into the PS thread payload, and eventually included in the RTWrite DataPort message header for use by the DataPort shared function. Software is responsible for ensuring this field is zero whenever a programmable index value is not required. When a programmable index value is required software must ensure that the correct 11-bit value is written to this field. Specifically, the kernels must perform a range check of computed index values against [0,2047], and output zero if that range is exceeded. Note that the unmodified "renderTargetArrayIndex" must be maintained in the VUE outside of the Vertex Header. Downstream, the DataPort range-checks the 11-bit index values against the range [**MinimumArrayElement**, **Depth**] state variables (SURFACE_STATE) associated with the specified render target surface. Format: 0-based U11 index value
D2	31:0	**Viewport Index.** This value is used to select one of a possible 16 sets of viewport (VP) state parameters in the Clip unit's VertexClipTest function and in the SF unit's ViewportMapping and Scissor functions. The GS unit (even if disabled) will read back this value for all vertices exiting the GS stage and entering the Clip stage. When enabled, the GS unit will range-check the value against [0,**Maximum VPIndex]** (see GS_STATE) and use a value of zero if out-of-range. When disabled, the GS unit instead uses the range [0,15]. After this range-check the values are sent down the pipeline and used in the Clip unit's VertexClipTest function. For vertices passing through the Clip stage, these values will also be sent to the SF unit for use in ViewportMapping and Scissor functions. The Clip unit (if enabled) will read back this value only for vertices generated by CLIP threads. Unlike the GS unit, the Clip unit will not apply any range check and instead just use the lower 4 bits. No hardware clamping is performed on these read-back values – the read-back values will be used unmodified by the SF unit. The CLIP kernel is therefore responsible for performing any required clamping on this value prior to writing the VUE Vertex Header. Software is responsible for ensuring this field is zero whenever a programmable index value is not required. Format: 0-based U32 index value
D3	31:19	**Reserved**: MBZ

DWord Bit		Description
	18:8	**Point Width.** This field specifies the width of POINT objects in screen-space pixels. It is used only for vertices within POINTLIST and POINTLIST_BF primitive topologies, and is ignored for vertices associated with other primitive topologies. This field is read back by both the GS and Clip units. Format: U8.3 pixels
	7:0	**User Clip Codes.** These are 'outside' status bits associated with the vertex element components marked as CullDistance or ClipDistance. The JITTER is required to generate code to compute and pack these bits. If a Cull/ClipDistance value is negative or a NaN value, its corresponding User Clip Code bit should be set. Up to eight values/bits are supported. The CLIP unit supports the **UserClipFlag ClipTest Enable Bitmask** (CLIP_STATE) which is applied to this field before being used in ClipTest. This field is read back only by the GS unit. This field is ignored for CLIP thread-generated vertices, as this information is only relevant to CLIP input vertices. Format: BITMASK8
D4	31:0	**Vertex Position X Coordinate.** If this is a PREMAPPED vertex, this field contains the X component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the X component of the vertex's NDC space position (i.e., the clip space X component divided by the clip space W component). Format: FLOAT32
D5	31:0	**Vertex Position Y Coordinate.** If this is a PREMAPPED vertex, this field contains the Y component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the Y component of the vertex's NDC space position (i.e., the clip space Y component divided by the clip space W component). Format: FLOAT32
D6	31:0	**Vertex Position Z Coordinate.** If this is a PREMAPPED vertex, this field contains the Z component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the Z component of the vertex's NDC space position (i.e., the clip space Z component divided by the clip space W component). Format: FLOAT32
D7	31:0	**Vertex Position RHW Coordinate.** This field contains the reciprocal of the vertex's clip space W coordinate. Format: FLOAT32
(D8-Dn)	31:0	**(Remainder of Vertex Elements).** While DWords D0-D7 are exposed to the device (i.e., read back by FF units), DWords D8-Dn of vertices written (by threads) are opaque to the device. Software is free to format/use these DWords as desired. The absolute maximum size limit on this data is specified via a maximum limit on the amount of data that can be read from a VUE (including the Vertex Header) (**Vertex Entry URB Read Length** has a maximum value of 63 256-bit units). Therefore the Remainder of Vertex Elements has an absolute maximum size of 62 256-bit units. Of course the actual allocated size of the VUE can and will limit the amount of data in a VUE.

Figure 2-4. VUE Vertex Header ([Dev-ILK])

DWord	Bit	Description
D0	31:0	**Reserved**: MBZ
D1	31:11	**Reserved**: MBZ
	10:0	**Render Target Array Index.** This value is (eventually) used to index into a specific element of an "array" Render Target. It is read back by the GS unit (for all exiting vertices) and the Clip unit (for all clip-generated vertices), subsequently routed into the PS thread payload, and eventually included in the RTWrite DataPort message header for use by the DataPort shared function. Software is responsible for ensuring this field is zero whenever a programmable index value is not required. When a programmable index value is required software must ensure that the correct 11-bit value is written to this field. Specifically, the kernels must perform a range check of computed index values against [0,2047], and output zero if that range is exceeded. Note that the unmodified "renderTargetArrayIndex" must be maintained in the VUE outside of the Vertex Header. Downstream, the DataPort range-checks the 11-bit index values against the range [**MinimumArrayElement**, **Depth**] state variables (SURFACE_STATE) associated with the specified render target surface. Format: 0-based U11 index value
D2	31:0	**Viewport Index.** This value is used to select one of a possible 16 sets of viewport (VP) state parameters in the Clip unit's VertexClipTest function and in the SF unit's ViewportMapping and Scissor functions. The GS unit (even if disabled) will read back this value for all vertices exiting the GS stage and entering the Clip stage. When enabled, the GS unit will range-check the value against [0,**Maximum VPIndex]** (see GS_STATE) and use a value of zero if out-of-range. When disabled, the GS unit instead uses the range [0,15]. After this range-check the values are sent down the pipeline and used in the Clip unit's VertexClipTest function. For vertices passing through the Clip stage, these values will also be sent to the SF unit for use in ViewportMapping and Scissor functions. The Clip unit (if enabled) will read back this value only for vertices generated by CLIP threads. Unlike the GS unit, the Clip unit will not apply any range check and instead just use the lower 4 bits. No hardware clamping is performed on these read-back values – the read-back values will be used unmodified by the SF unit. The CLIP kernel is therefore responsible for performing any required clamping on this value prior to writing the VUE Vertex Header. Software is responsible for ensuring this field is zero whenever a programmable index value is not required. Format: 0-based U32 index value
D3	31:19	Reserved: MBZ
	18:8	**Point Width.** This field specifies the width of POINT objects in screen-space pixels. It is used only for vertices within POINTLIST and POINTLIST_BF primitive topologies, and is ignored for vertices associated with other primitive topologies. This field is read back by both the GS and Clip units. Format: U8.3 pixels

DWord Bit		Description
	7:0	**User Clip Codes.** These are the sign bits of the vertex element components marked as CullDistance or ClipDistance. The JITTER is required to assemble these sign bits. A negative value (sign bit set) indicates that the vertex is on the "outside" of the corresponding user clip plane. Up to eight sign bits (clip flags) are supported. The CLIP unit supports a mask that is applied to this field before being used in ClipTest. This field is read back only by the GS unit. This field is ignored for CLIP thread-generated vertices, as this information is only relevant to CLIP input vertices. Format: BITMASK8
D4	31:0	**Vertex Position X Coordinate.** If this is a PREMAPPED vertex, this field contains the X component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the X component of the vertex's NDC space position (i.e., the clip space X component divided by the clip space W component). Format: FLOAT32
D5	31:0	**Vertex Position Y Coordinate.** If this is a PREMAPPED vertex, this field contains the Y component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the Y component of the vertex's NDC space position (i.e., the clip space Y component divided by the clip space W component). Format: FLOAT32
D6	31:0	**Vertex Position Z Coordinate.** If this is a PREMAPPED vertex, this field contains the Z component of the vertex's screen space position. If this is an UNMAPPED vertex, this field contains the Z component of the vertex's NDC space position (i.e., the clip space Z component divided by the clip space W component). Format: FLOAT32
D7	31:0	**Vertex Position RHW Coordinate.** This field contains the reciprocal of the vertex's clip space W coordinate.
D8	31:0	**Vertex Position X Coordinate.** This field contains the X component of the vertex's 4D space position. Format: FLOAT32
D9	31:0	**Vertex Position Y Coordinate.** This field contains the Y component of the vertex's 4D space position Format: FLOAT32
D10	31:0	**Vertex Position Z Coordinate.** This field contains the Z component of the vertex's NDC space position Format: FLOAT32
D11	31:0	**Vertex Position W Coordinate.** This field contains the Z component of the vertex's 4D space position Format: FLOAT32
D12	31:0	**User Clip Distance to Plane0.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane0 Format: FLOAT32

DWord	Bit	Description
D13	31:0	**User Clip Distance to Plane1.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane1 Format: FLOAT32
D14	31:0	**User Clip Distance to Plane2.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane2 Format: FLOAT32
D15	31:0	**User Clip Distance to Plane3.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane3 Format: FLOAT32
D16	31:0	**User Clip Distance to Plane4.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane4 Format: FLOAT32
D17	31:0	**User Clip Distance to Plane5.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane5 Format: FLOAT32
D18	31:0	**User Clip Distance to Plane6.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane6 Format: FLOAT32
D19	31:0	**User Clip Distance to Plane7.** If the User Clip Plane0 is enabled,This field contains distance from the vertex to the User Clip Plane7 Format: FLOAT32
(D20-Dn)	31:0	(Remainder of Vertex Elements). While DWords D0-D19 are exposed to the device (i.e., read back by FF units), DWords D20-Dn of vertices written (by threads) are opaque to the device. Software is free to format/use these DWords as desired. The absolute maximum size limit on this data is specified via a maximum limit on the amount of data that can be read from a VUE (including the Vertex Header) (Vertex Entry URB Read Length has a maximum value of 63 256-bit units). Therefore the Remainder of Vertex Elements has an absolute maximum size of 62 256-bit units. Of course the actual allocated size of the VUE can and will limit the amount of data in a VUE.

2.5.2 Vertex Positions

(For the sake of brevity, the following discussion will use the term *map* as a shorthand for "compute screen space coordinate via perspective divide followed by viewport transform".)

The "Position" fields of the Vertex Header are the only vertex position coordinates exposed to the 3D Pipeline. The CLIP and SF units are the only FF units which perform operations using these positions. The VUE will likely contain other position attributes for the vertex outside of the Vertex Header, though this information is not directly exposed to the FF units. For example, the Clip Space position will likely be required in the VUE (outside of the Vertex Header) in order to perform correct and robust 3D Clipping in the CLIP thread.

In the CLIP unit, the read-back Position fields are interpreted as being in one of two coordinate systems, depending on the **CLIP_STATE.VertexPositionSpace** bit. The CLIP unit will modify its VertexClipTest function depending on the coordinate space of the incoming vertices.

- **[Pre-DevSNB]:VPOS_NDCSPACE (Normalized Device Coordinate Space position, post-perspective division)**: This is the typical coordinate space in which vertex positions are defined upon input to the CLIP unit. A *speculative* perspective-division will have been performed, though the viewport map transformation will not have been applied (as this is provided by the downstream SF FF unit). An advantage of clip-testing in NDC space is that the View Volume has canonical unit dimensions (i.e., it's cheap to test against). The "speculative" nature of the perspective divide is discussed below.

- **VPOS_SCREENSPACE (Screen Space position)**: Under certain circumstances, the position in the Vertex Header will contain the screen-space (pixel) coordinates (post viewport mapping).

The SF unit does not have a state bit defining the coordinate space of the incoming vertex positions. Software must use the Viewport Mapping function of the SF unit in order to ensure that screen-space coordinates are available after that function. If screen space coordinates are passed into SF, then software will likely turn off the Viewport Mapping function.

The following subsections briefly describe the three relevant coordinate spaces.

2.5.2.1 Clip Space Position

The *clip-space* position of a vertex is defined in a homogeneous 4D coordinate space where, after perspective projection (division by W), the visible "view volume" is some canonical (3D) cuboid. Typically the X/Y extents of this cuboid are [-1,+1], while the Z extents are either [-1,+1] or [0,+1]. The API's VS or GS shader program will include geometric transforms in the computation of this clip space position such that the resulting coordinate is positioned properly in relation to the view volume (i.e., it will include a "view transform" in this computation path).

Note that, under typical perspective projections, the clip-space W coordinate is equal to the view-space Z coordinate.

A vertex's clip-space coordinates must be maintained in the VUE up to 3D clipping, as this clipping is performed in clip space.

- In [**Pre-DevSNB**], clip-space positions are stored outside of (beyond) the Vertex Header. VS/GS/Clip kernels must perform perspective projection internally and subsequently store the post-projected (NDC-space, see below) position in the Vertex Header for use by the FF pipeline.

2.5.2.2 NDC Space Position

A perspective divide operation performed on a clip-space position yields a [X,Y,Z,RHW] NDC (Normalized Device Coordinates) space position. Here "normalized" means that visible geometry is located within the [-1,+1] or [0,+1] extent view volume cuboid (see clip-space above).

- The NDC X,Y,Z coordinates are the clip-space X,Y,Z coordinates (respectively) divided by the clip-space W coordinate (or, more correclty, the clip-space X,Y,Z coordinates are multiplied by the reciprocal of the clip space W coordinate).
    - Note that the X,Y,Z coordinates may contain INFINITY or NaN values (see below).
- The NDC RHW coordinate is the reciprocal of the clip-space W coordinate and therefore, under normal perspective projections, it is the reciprocal of the view-space Z coordinate. Note that NDC space is really a 3D coordinate space, where this RHW coordinate is retained in order to perform perspective-correct interpolation, etal. Note that, under typical perspective projections.
    - Note that the RHW coordinate make contain an INFINITY or NaN value (see below).

Speculative Perspective Divide [Pre-DevSNB]

When operating in VPOS_NDCSPACE mode, the CLIP stage requires a 'speculative' PerspectiveDivide to have been performed on all incoming vertices. This places a requirement on software (the JITTER) to cause the NDC coordinates to be computed and stored prior to the CLIP stage, in addition to any shader functions which may be required. In the case where the application simply inputs clip space positions without any intervening processing prior to the CLIP stage, software must cause the speculative PerspectiveDivide function to be performed in the VS/GS thread.

This PerspectiveDivide function is considered speculative in that the results may not be used, i.e., in the case where the vertex lies outside the clipping boundaries. Note that, when performing PerspectiveDivide before 3DClipping, the resulting NDC coordinates may not even be representable. For example, the clip-space W coordinate may be zero or close enough to zero to cause the X/W, Y/W or Z/W operation to result in an INFINITE value. However, in these cases, the PerspectiveDivide results will not be used, and instead the corresponding clip-space coordinates will be used as input to the 3DClipping function (assuming the object is not trivially rejected).

NaN Values in NDC Coordinate Components

There are cases where a speculative PerspectiveDivide can produce NaN results. The following table shows these cases for the computation of X/W (same holds true for Y/W and Z/W).

W RHW		Clip X	NDC X = X*RHW	Comments
NaN	NaN	d/c	NaN	Clip space position not representable (W is NaN)
d/c	d/c	NaN	NaN	Clip space position not representable (X is NaN)
+/-INF	+/-0	+/-INF	NaN	Clip space position is representable, but 3D clipping will not yield valid results.
+/-0 or denorm	+/-INF	+/-0 or denorm	NaN	Clip space postiion is representable. This is a case where a NDC X,Y,Z component can be NaN even when the Clip space position is representable. 3D Clipping can yield valid results.
+/-INF	+/-0	Not +/-INF	+/-0	This is the case of infinite perspective, where the vertex collapses to the NDC origin.
+/-0 or denorm	+/-INF	Not (+/-0 or denorm)	+/-INF	This is a case where an infinite NDC coordinate is generated, though 3D Clipping will be able to produce valid results.

During VertexClipTest, any vertex with an NaN NDC RHW coordinate will be marked as “BAD”. During ClipDetermination, any object containing a ‘BAD’ vertex will be trivially rejected.

2.5.2.3 Screen-Sp ace Position

Screen-space coordinates are defined as:

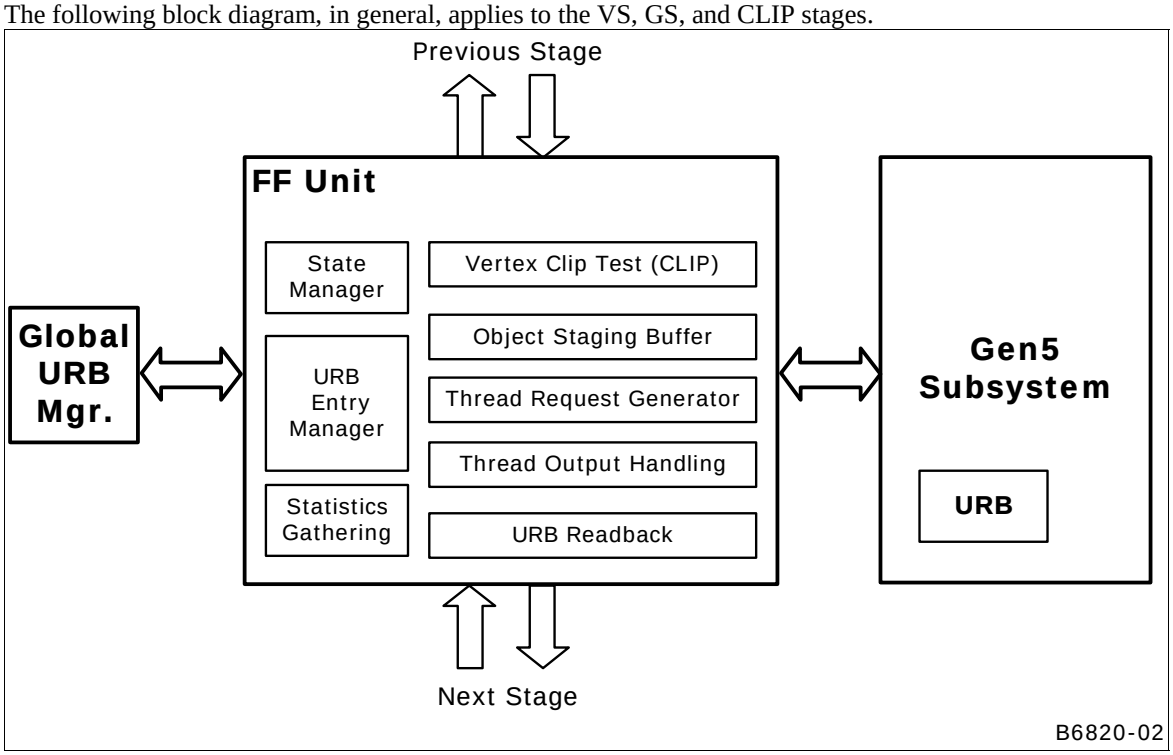
- X,Y coordinates are in absolute screen space (pixel coordinates, upper left origin). See Vertex X,Y Clamping and Quantization in the SF section for a discussion of the limitations/restrictions placed on screenspace X,Y coordinates.
- Z coordinate has been mapped into the range used for DepthTest.
- RHW coordinate is actually the reciprocal of clip-space W coordinate (typically the reciprocal of the view-space Z coordinate).

2.6 3D Pipeline Stage Overview

The fixed-function (FF) stages of the 3D pipeline share some common functionality, specifically related to the creation and management of threads. This chapter is intended to describe the behavior and programming model of these common functions, in an effort to not replicate this information for each pipeline stage. Stage-specific exceptions to the information provided here will be included in the stage-specific chapters to follow.

2.6.1 Generic 3D FF Unit Block Diagram

The following block diagram, in general, applies to the VS, GS, and CLIP stages.



2.6.2 Common 3D FF Unit Functions

A major role of the FF stages is in managing the GENx threads that perform the majority of the processing on the vertex/pixel data. (In general, the amount of non-thread processing performed by the 3DPIPE stages increases towards the end of the pipeline.) In a generic sense, the key functions included are:

- Bypass Mode
- URB Entry Management
- Thread Initiation Management
- Thread Request Data Generation
  - Thread Control Information Generation
  - Thread Payload Header Generation
  - Thread Payload Data Generation
- Thread Output Handling
- URB Entry Readback
- Statistics Gathering

The following table lists the various state variables used to control the common FF functions:

State Variable	Programmed Via	Generic Functions Affected
<stage> Enable	[Pre-DevSNB]: 3DSTATE_PIPELINED_POINTERS	Bypass Mode
Kernel Start Pointer	[Pre-DevSNB]: Pipeline State Descriptor	Thread Request Data Gen.
GRF Register Block Count		Thread Request Data Gen.
Single Program Flow		Thread Request Data Gen.
Thread Priority		Thread Request Data Gen.
Floating Point Mode		Thread Request Data Gen.
Exceptions Enable		Thread Request Data Gen.
Scratch Space Base Pointer		Thread Request Data Gen.
Per Thread Scratch Space		Thread Request Data Gen.
Constant URB Entry Read Length		Payload Data Gen.
Constant URB Entry Read Offset		Payload Data Gen.
Vertex URB Entry Read Length		Payload Data Gen.
Vertex URB Entry Read Offset		Payload Data Gen.
Dispatch GRF Start Register for URB Data		Payload Data Gen.
Maximum Number of Threads		Thread Resource Alloc. Scratch Space Mgt.

State Variable	Programmed Via	Generic Functions Affected
<stage> Fence	URB_FENCE_POINTER	URB Entry Mgt.
URB Entry Allocation Size	[Pre-DevSNB]: Pipeline State Descriptor	URB Entry Mgt.
Number of URB Entries		URB Entry Mgt.
Sampler State Pointer	[**Pre-DevSNB**]: Pipeline State Descriptor[	Payload Header Gen.
<stage> Binding Table Pointer	3DSTATE_BINDING_TABLE_ POINTERS	This gets routed directly to shared functions (transparent to software).
Sampler Count	[Pre-DevSNB]: Pipeline State Descriptor	Thread Request Data Gen.
Binding Table Entry Count		Thread Request Data Gen.
Statistics Enable		Statistics Gathering

2.6.3 Thread Initiation Management

Those FF stages that can spawn threads must have buffered the input (URB entries) available to supply a thread, and then ensure that there are sufficient resources (within the domain of the 3D pipeline) to make the thread request.

Once a FF stage determines a thread request can be submitted, (a) all input data required to initiate the thread is generated, (b) this information is submitted to the common thread dispatcher, (c) the thread dispatcher will spawn the thread as soon as an EU with sufficient GRF resources becomes available, and finally (d) the thread will start execution. With respect to concurrent threads, steps (c) and (d) can proceed out of order (i.e., a threads are not necessarily dispatched in the order that the thread requests are submitted to the thread dispatcher).

2.6.3.1 Thread Input Buffering

Each FF stage varies with regard to thread input requirements, and so this will not be discussed in this chapter other than the overview information provided in the following table:

FF Stage	Thread Input Requirements
CS	N/A (does not spawn threads)
VF	N/A (does not spawn threads)
GS	All the vertices associated with an object must be buffered before a GS thread can be initiated to process the object.
VS	Normally, two vertices are buffered before a VS thread is spawned to shade the pair in parallel. Under some circumstances (e.g., a flush, state change, etc.) a single vertex will be shaded.
CLIP	[Pre-DevSNB]: All the vertices associated with an object must be buffered before a CLIP thread can be initiated to process the object.
SF	[Pre-DevSNB]: All the vertices associated with an object must be buffered before a SETUP thread can be initiated to process the object.
WM	Threads spawned as required by the rasterization algorithm.

2.6.3.2 Thread Resource Allocation [Pre-Dev-ILK]

Once a FF stage that spawn threads has sufficient input to initiate a thread, it must guarantee that it is safe to request the thread initiation. For all these FF stages, this check is based on :

- The **availability of output URB entries**:
  - GS: At least one output URB entry must be available to serve as the initial output vertex from the GS thread. However, software must guarantee that additional URB entries will eventually become available to allow the pipeline to make forward progress and not deadlock. There are two considerations here:
    - Single GS Threads (**Maximum Number of Threads** == 1): There must be enough GS output URB entries allocated to allow the GS thread to make progress (call this number P). P must include enough vertices to allow the next enabled stage to make progress, i.e., must contain enough vertices for the worst-case object within a primitive. For example, the system would hang if the GS stage was only allocated 2 URB entries and the GS thread tried to output a TRILIST. In this case the GS stage would need to be allocated at least 3 URB entries – the GS thread would output the first 3 vertices, then would stall on the allocation of the 4th vertex until the rest of the pipeline consumed that first triangle and dereferenced the first vertex. The clipper, when enabled, imposes additional requirements on the number of output URB entries allocated to the GS. Because of the way the clipper processes strip/fan primitives, it will not release the URB entries for the vertices of a given object until it has finished processing the *next* object in the primitive. The minimum number of handles that must be allocated to the GS for strip/fan –type primitives is thus increased according to the following table:

Topology	Minimum GS Handles
LINESTRIP, LINESTRIP_BF, LINESTRIP_CONT, LINESTRIP_CONT_BF	3
POLYGON, TRIFAN, TRIFAN_NOSTIPPLE	4
TRISTRIP, TRISTRIP_REV	5

    - Dual GS threads: If two concurrent GS thread are permitted, software must account for the possibility that the subsequent GS thread completes before the preceding GS thread outputs its first vertex. Therefore there must be enough URB entries allocated to satisfy the above minimums for both threads.
  - CLIP: Same considerations as GS (above)
  - SF: An output URB entry must be available to store the results of the SETUP thread.
  - WM: N/A (does not output to URB)
- The **Maximum Number of Threads** state variable. This state variable limits the number of concurrent threads a FF stage can have executing. As long as the FF stage is operating below this limit, it can make additional thread initiation requests.
- In addition, the WM unit utilizes a **scoreboard** mechanism to ensure proper ordering of operations – and this mechanism can postpone the initiation of new threads. (See Windower chapter).

Software is responsible for programming of **Maximum Number of Threads** to ensure the correct and optimal operation of the 3D pipeline.

The considerations for programming **Maximum Number of Threads** are summarized below:

1. **URB Allocation**: (See discussion above)

2. **Scratch Space Allocation**: When the current kernel of an enable stage requires use of scratch space (for API-defined temporary storage, register spill/fill, overflow stacks, etc.), software must limit the number of concurrent threads (via **Maximum Number of Threads**) such that the total scratch space requirement is satisfied by the amount of scratch space memory allocated to the FF stage.

3. **Stream Output Serialization**: If a kernel is required to output a serialized stream of data to a memory, threads for that stage must be serialized by SW only allowing (**Maximum Number of Threads** == 1).

4. **Performance**: In general, a larger number of possibly-concurrent threads will better ensure the GENx cores are fully utilized.

(*Note*: The 3D pipeline can function correctly with (**Maximum Number of Threads** == 1) set at each enabled stage, given that there are sufficient resources to run this single thread (scratch space, etc). However, this will certainly not be an optimal configuration. See *Graphics Processing Engine* for a discussion of URB Allocation Requirements and Guidelines which includes information on programming the Number Of Threads for the various FF units.)

2.6.3.3 Thread Resource Allocation [Dev-ILK]

In general, the considerations listed in the preceding Pre-Dev-ILK section are relevant, with the following exceptions:

- The **availability of output URB entries**:
  - GS: Although up to 32 concurrent GS threads are allowed, only two can be outputting URB entries. Therefore, the considerations listed in the preceding Pre-Dev-ILK section are relevant.
  - CLIP: Same considerations as GS (above), except only 16 concurrent Clip threads are allowed.

2.6.4 Thread Request Generation

Once a FF unit determines that a thread can be requested, it must gather all the information required to submit the thread request to the Thread Dispatcher. This information is divided into several categories, as listed below and subsequently described in detail.

- **Thread Control Information**: This is the information required (from the FF unit) to establish the execution environment of the thread. Note that some information affecting the thread execution state is programmed external to the 3D pipeline (e.g., Exception Handler IP, Breakpoint IP, etc.)
- **Thread Payload Header**: This is the first portion of the thread payload passed in the GRF, starting at GRF R0. This is information passed directly from the FF unit. It precedes the Thread Payload Input URB Data.
- **Thread Payload Input URB Data**: This is the second portion of the thread payload. It is read from the URB using entry handles supplied by the FF unit.

2.6.4.1 Thread Control Information

The following table describes the various state variables that a FF unit uses to provide information to the Thread Dispatcher and which affect the thread execution environment. Note that this information is not directly passed to the thread in the thread payload (though some fields may be subsequently accessed by the thread via architectural registers).

Table 2-2. State Variables Included in Thread Control Information

State Variable	Usage	FFs
Kernel Start Pointer	This field, together with the **General State Pointer**, specifies the starting location (1st GENx core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Pointer**.	All FFs spawning threads
GRF Register Block Count	Specifies, in 16-register blocks, how many GRF registers are required to run the kernel. The Thread Dispatcher will only seek candidate EUs that have a sufficient number of GRF register blocks available. Upon selecting a target EU, the Thread DIspatcher will generate a logical-to-physical GRF mapping and provide this to the target EU.	All FFs spawning threads
Single Program Flow (SPF)	Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). See CR0 description in *ISA Execution Environment*.	All FFs spawning threads
Thread Priority	The Thread Dispatcher will give priority to those thread requests with Thread Priority of HIGH_PRIORITY over those marked as LOW_PRIORITY. Within these two classes of thread requests, the Thread Dispatcher applies a priority order (e.g., round-robin --- though this algorithm is considered a device implementation-dependent detail).	All FFs spawning threads
Floating Point Mode	This determines the initial value of the **Floating Point Mode** bit of the EU's CR0 architectural register that controls floating point behavior in the EU core. (See ISA.)	All FFs spawning threads
Exceptions Enable	This bitmask controls the exception hanlding logic in the EU. (See ISA.)	All FFs spawning threads
Sampler Count	This is a hint which specifies how many indirect SAMPLER_STATE structures should be prefetched concurrent with thread initiation. It is recommended that software program this field to equal the number of samplers, though there may be some minor performance impact if this number gets large. This value should not exceed the number of samplers accessed by the thread as there would be no performance advantage. Note that the data prefetch is treated as any other memory fetch (with respect to page faults, etc.).	All stages supporting sampling (VS, GS, WM)
Binding Table Entry Count	This is a hint which specifies how many indirect BINDING_TABLE_STATE structures should be prefetched concurrent with thread initiation. (The comments included in Sampler Count (above) also apply to this field).	All FFs spawning threads

2.6.4.2 Thread Payload Generation

FF units are responsible for generating a thread *payload* – the data pre-loaded into the target EU's GRF registers (starting at R0) that serves as the primary direct input to a thread's kernel. The general format of these payloads follow a similar structure, though the exact payload size/content/layout is unique to each stage. This subsection describes the common aspects – refer to the specific stage's chapters for details on any differences.

The payload data is divided into two main sections: the *payload header* followed by the *payload URB data*. The payload header contains information passed directly from the FF unit, while the payload URB data is obtained from URB locations specified by the FF unit.

NOTE: The first 256 bits of the thread payload (the initial contents of R0, aka "the R0 header") is specially formatted to closely match (and in some cases exactly match) the first 256 bits of thread-generated *messages* (i.e., the message header) accepted by shared functions. In fact, the send instruction supports having a copy of a GR's contents (such as R0) used as the message header. Software must take this intention into account (i.e., "don't muck with R0 unless you know what you're doing"). This is especially important given the fact that several fields in the R0 header are considered opaque to SW, where use or modification of their contents might lead to UNDEFINED results.

The payload header is further (loosely) divided into a leading *fixed payload header* section and a trailing, variable-sized *extended payload header* section. In general the size, content and layout of both payload header sections are FF-specific, though many of the fixed payload header fields are common amongst the FF stages. The extended header is used by the FF unit to pass additional information specific to that FF unit. The extended header is defined to start after the fixed payload header and end at the offset defined by **Dispatch GRF Start Register for URB Data**. Software can cause use the **Dispatch GRF Start Register for URB Data** field to insert padding into the extended header in order to maintain a fixed offset for the start of the URB data.

2.6.4.2.1 Fixed Payload Header

The payload header is used to pass FF pipeline information required as thread input data. This information is a mixture of SW-provided state information (state table pointers, etc.), primitive information received by the FF unit from the FF pipeline, and parameters generated/computed by the FF unit. most of the fields of the fixed header are common between the FF stages. These non-FF-specific fields are described in Table 2-3. Note that a particular stage's header may not contain all these fields, so they are not "common" in the strictest sense.

Table 2-3. Fixed Payload Header Fields (non-FF-specific)

Fixed Payload Header Field (non-FF-specific)	Description FFs	
FF Unit ID	Function ID of the FF unit. This value identifies the FF unit within the GENx subsystem. The FF unit will use this field (when transmitted in a Message Header to the URB Function) to detect messages emanating from its spawned threads.	All FFs spawning threads
Thread ID	This field uniquely identifies this thread within the FF unit over some period of time.	All FFs spawning threads
Scratch Space Pointer	This is the starting location of the thread's allocated scratch space, specified as an offset from the **General State Base Address**. Note that scratch space is allocated by the FF unit on a per-thread basis, based on the **Scratch Space Base Pointer** and **Per-Thread Scratch Space Size** state variables. FF units will assign a thread an arbitrarily-positioned region within this space. The scratch space for multiple (API-visible) entities (vertices, pixels) will be interleaved within the thread's scratch space.	All FFs spawning threads
Dispatch ID	This field identifies this thread within the outstanding threads spawned by the FF unit. This field does not uniquely identify the thread over any significant period of time. *Implementation Note*: This field is effectively an "active thread index". It is used on a thread's URB allocation request to identify which thread's handle pool is to source the allocation. It is used upon thread termination to free up the thread's scratch space allocation.	All FFs spawning threads
Binding Table Pointer	This field, together with the **Surface State Base Pointer**, specifies the starting location of the Binding Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **Surface State Base Pointer**. See *Shared Functions* for a description of a Binding Table.	All FFs spawning threads

Fixed Payload Header Field (non-FF-specific)	Description FFs	
Sampler State Pointer	This field, together with the **General State Base Pointer**, specifies the starting location of the Sampler State Table used by threads spawned by the FF unit. It is specified as a 64-byte-granular offset from the **General State Base Pointer**. See *Shared Functions* for a description of a Sampler State Table.	All FFs spawning threads which sample (VS, GS, WM)
Per Thread Scratch Space	This field specifies the amount of scratch space allocated to each thread spawned by the FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space.	All FFs spawning threads
Handle ID <n>	This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. The FF unit will use this information upon detecting a URB_WRITE message issued by the thread. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS do not write to URB entries, and therefore this info is not supplied.	VS, GS,CLIP,SF
URB Return Handle <n>	This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry. Threads spawned by the VS, GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS, GS,CLIP,SF
Primitive Topology Type	As part of processing an incoming primitive, a FF unit is often required to spawn a number of threads (e.g., for each individual triangle in a TRIANGLE_STRIP). This field identifies the type of primitive which is being processed by the FF unit, and which has lead to the spawning of the thread. GENx kernels written to process different types of objects can use this value to direct that processing. E.g., when a CLIP kernel is to provide clipping for all the various primitive types, the kernel would need to examine the Primitive Topology Type to distinguish between point, lines, and triangle clipping requests. NOTE: In general, this field is identical to the Primitive Topology Type assoociated with the primitive vertices as received by the FF unit. Refer to the individual FF unit chapters for cases where the FF unit modifies the value before passing it to the thread. (E.g., certain units perform toggling of TRIANGLESTRIP and TRIANGLESTRIP_REV).	GS, CLIP, SF, WM

2.6.4.2.2 Extended Payload Header

The extended header is of variable-size, where inclusion of a field is determined by FF unit state programming.

In order to permit the use of common kernels (thus reducing the number of kernels required), the **Dispatch GRF Start Register for URB Data** state variable is supported in all FF stages. This SV is used to place the payload URB data at a specific starting GRF register, irrespective of the size of the extended header. A kernel can therefore reference the payload URB data at fixed GRF locations, while conditionally referencing extended payload header information.

2.6.4.2.3 Payload URB Data

In each thread payload, following the payload header, is some amount of URB-sourced data required as input to the thread. This data is divided into an optional *Constant URB Entry* (CURBE), following either by a Primitive URB Entry (WM) or a number of Vertex URB Entries (VS, GS, CLIP, SF). A FF unit only knows the location of this data in the URB, and is never exposed to the contents. For each URB entry, the FF unit will supply a sequence of handles, read offsets and read lengths to the GENx subsystem. The subsystem will read the appropriate 256-bit locations of the URB, optionally perform swizzling (VS only), and write the results into sequential GRF registers (starting at **Dispatch GRF Start Register for URB Data**).

Table 2-4. State Variables Controlling Payload URB Data

State Variable	Usage	FFs
Dispatch GRF Start Register for URB Data	This SV identifies the starting GRF register receiving payload URB data. Software is responsible for ensuring that URB data does not overwrite the Fixed or Extended Header portions of the payload.	FFs spawning threads
[Pre-DevSNB] Constant URB Entry Read Offset	This SV determines the starting offset with the CURBE from which constant URB data to be read and supplied in this stage's payloads. It is specified as a 256-bit offset into the current CURBE. As the CURBE is (optionallly) used by all pipeline stages to supply constant data, this SV is used by SW to select the constants to be used for a particular stage. The sources of constant data within the CURBE for different stages can overlap. Specifying a constant data source extending beyond the end of the CURBE is UNDEFINED.	
[Pre-DevSNB] Constant URB Entry Read Length	This SV determines the amount of data (starting from **Constant URB Entry Read Offset**) to be read from the CURBE and passed into the payload URB data. It is specified in 256-bit units. If zero, no constant data is read. SW must program a zero value whenever the Constant Buffer is invalid (i.e., the CURBE is unspecified). Specifying a constant data source extending beyond the end of the CURBE is UNDEFINED.	

State Variable	Usage	FFs
Vertex URB Entry Read Offset	This SV specifies the starting offset within VUEs from which vertex data is to be read and supplied in this stage's payloads. It is specified as a 256-bit offset into any and all VUEs passed in the payload. This SV can be used to skip over leading data in VUEs that is not required by the stage's threads (e.g., skipping over the Vertex Header data at the SF stage, as that information is not required for setup calculations). Skipping over irrelevant data can only help to improve performance. Specifying a vertex data source extending beyond the end of a vertex entry is UNDEFINED.	VS, GS, Pre-DevSNB: CLIP, SF
Vertex URB Entry Read Length	This SV determines the amount of vertex data (starting at **Vertex URB Entry Read Offset**) to be read from each VUEs and passed into the payload URB data. It is specified in 256-bit units. A zero value is INVALID (at very least one 256-bit unit must be read). Specifying a vertex data source extending beyond the end of a VUE is UNDEFINED.	

Programming Restrictions: (others may already been mentioned)

- The maximum size payload for any thread is limited by the number of GRF registers available to the thread, as determined by min(128, 16 * **GRF Register Block Count**). Software is responsible for ensuring this maximum size is not exceeded, taking into account:
  - The size of the Fixed and Extended Payload Header associated with the FF unit.
  - The **Dispatch GRF Start Register for URB** Data SV.
  - The amount of CURBE data included (via **Constant URB Entry Read Length**)
  - The number of VUEs included (as a function of FF unit, it's state programming, and incoming primitive types)
  - The amount of VUE data included for each vertex (via **Vertex URB Entry Read Length**)
  - (For WM-spawned PS threads) The amount of Primitive URB Entry data.
- For any type of URB Entry reads:
  - Specifying a source region (via Read Offset, Read Length) that goes past the end of the URB Entry allocation is illegal.
    - The allocated size of Vertex/Primitive URB Entries is determined by the **URB Entry Allocation Size** value provided in the pipeline state descriptor of the FF unit owning the VUE/PUE.
    - The allocated size of CURBE entries is determined by the **URB Entry Allocation Size** value provided in the CS_URB_STATE command.

2.6.5 Thread Output Handling

Those FF units spawning threads are responsible for monitoring and responding to certain events generated by their spawned threads. Such events are indirectly detected by these FF units monitoring messages sent from threads to the URB Shared Function. By snooping the Message Bus Sideband and Header information, a FF can detect when a particular spawned thread sends a message to the URB function. A subset of this information is then captured and acted upon. Refer to the *URB* chapter for more details (including a table of valid/invalid combinations of the **Complete**, **Used**, **Allocate**, and **EOT** bits)

The following subsections describe functions that FF units perform as part of Thread Output Handling.

2.6.5.1 URB Entry Output (VS, GS, [Pre-DevSNB]: CLIP, SF)

The following description is applicable only to the GS, and for [Pre-Dev-ILK], the CLIP and SF stages.

For these threads the main (if not only) output of the thread takes the form of data written to one or more destination VUEs. At very least this is the only form of thread output visible to the FF units.

When a thread sends a URB_WRITE message to the URB function with the **Complete** and **Used** bits set in the Message Description, the spawning FF unit recognizes this as the thread having completely written a destination UE. (In the typical case of a VS thread, a pair of UEs will be written in parallel). The thread must not target any additional URB messages to this UE (unless it gets reallocated to the thread). The FF unit marks this UE as complete and available for output.

In the case where multiple concurrent threads are supported at a given stage, the FF unit is responsible for outputing UEs down the pipeline in order. I.e., all VUE outputs of a spawned thread must be sent down the pipeline (in order of allocation to the thread) prior to any outputs from a subsequently-spawned thread. This is required even if the subsequent threads perform any/all of their output prior to the preceding thread producing any/some output.

2.6.5.2 VUE Allocation (GS, CLIP) [Pre-Dev-ILK

The following description is applicable only to the VS, GS, CLIP stages.

The GS and CLIP threads are passed a single, initial destination VUE handle. These threads may be required to output more than one destination VUE, and therefore they are provided with a mechanism to request additional handles.

When a GS or CLIP thread issues a URB_WRITE message with the **Allocate** bit set, the spawning FF unit will consider this a request for the allocation of an additional VUE handle. The thread must specify a destination GRF register for the message writeback data. The spawning FF unit will perform the allocation, and provide the writeback data (containing **Handle ID** and **URB Return Handle**) to the GENx subsystem, which will in turn deliver that data to the appropriate GRF register. (See the *URB* chapter for the definition of this writeback data).

The thread is allowed to proceed while the allocation is taking place (it is guaranteed to complete at some point). If the thread attempts to reference the writeback data before the allocation has completed, execution will be stalled in the same fashion any unfulfilled dependency is handled. It is therefore recommended that

SW (a) request the additional allocation as soon as possible, and (b) reference the writeback data as late as possible in order to keep the thread in a runnable state. (Refer to the following subsection to see how the thread is allowed to "allocate ahead" and give back unused VUE handles).

NOTE: GS and CLIP threads must write VUEs in the order they are allocated by the FF unit (in response to an allocation request from the thread), starting with the initial destination handle passed in the thread payload.

A GS or CLIP thread is restricted as to the number of URB handles it can retain. Here a "retained" handle refers to a URB handle that (a) has been pre-allocated or allocated and returned to the thread via the **Allocate** bit in the URB_WRITE message, and (b) has yet to be returned to the pipeline via the **Complete** bit in the URB_WRITE message.

- When operating in single-thread mode (**Maximum Number of Threads** == 1), the number of retained handles must not exceed min(16, **Number of URB Entries**).
- When operating in dual-thread mode (**Maximum Number of Threads** == 2), the number of retained handles must not exceed (**Number of URB Entries**/2).

This restriction is not expected to be significant in that most/all GS/CLIP threads are expected to retain only a few (<=4) handles.

2.6.5.3 VUE Allocation (GS, CLIP) [Dev-ILK]

The following description is applicable only to the GS, CLIP stages.

The threads are not passed an initial handle. Instead, they request a first handle (if any) via the URB shared function's FF_SYNC message (see Shared Functions). If additional handles are required, the URB_WRITE allocate mechanism (mentioned above) is used.

2.6.5.4 VUE Dereference (GS, [Pre-DevSNB]: CLIP)

The following description is applicable only to the GS stage, and for Pre-DevSNB, the CLIP stages.

It is possible and legal for a thread to produce no output or subsequently allocate a destination VUE that was not required (e.g., the thread allocated ahead). Therefore, there is a mechanism by which a thread can "give back" (dereference) an allocated VUE. This mechanism must be used if the VUE is not written before the thread terminates.

A kernel can explicitly dereference a VUE by issuing a URB_WRITE message (specifying the to-be-dereference handle) with the **Complete** bit set and the **Used** bit clear.

2.6.5.5 Thread Termination

All threads must explicitly terminate by executing a SEND instruction with the EOT bit set. (See *EU* chapters). When a thread spawned by a 3D FF unit terminates, the spawning FF unit detects this termination as a part of Thread Management. This allows the FF units to manage the number of concurrent threads it has spawned and also manage the resources (e.g., scratch space) allocated to those threads.

Programming Note: [Pre-Dev-ILK] GS and Clip threads must terminate by sending a URB_WRITE message (with EOT set) with the Complete bit also set (therein returning a URB handle marked as either used or un-used).

2.6.6 VUE Readback

Starting with the CLIP stage, the 3D pipeline requires vertex information in addition to the VUE handle. For example, the CLIP unit's VertexClipTest function needs the vertex position, as does the SF unit's functions. This information is obtained by the 3D pipeline reading a portion of each vertex's VUE data directly from the URB. This readback (effectively) occurs immediately before the CLIP VertexClipTest function, and immediately after a CLIP thread completes the output of a destination VUE.

The Vertex Header (first 256 bits) of the VUE data is read back. (See the previous *VUE Formats* subsection (above) for details on the content and format of the Vertex Header.) .

This readback occurs automatically and is not under software control. The only software implication is that the Vertex Header must be valid at the readback points, and therefore must have been previously loaded or written by a thread.

2.7 Synchronization of the 3D Pipeline

Two types of synchronizations are supported for the 3D pipe: top of the pipe and end of the pipe. Top of the pipe synchronization really enforces the read-only cache invaliadation. This synchronization guarantees that primitives rendered after such synchronization event fetches the latest read-only data from memory. End of the pipe synchronization enforces that the read and/or read-write buffers do not have outstanding hardware accesses. These are used to implement read and write fences as well as to write out certain statistics deterministically with respect to progress of primitives through the pipeline (and without requiring the pipeline to be flushed.) The PIPE_CONTROL command (see details below) is used to perform all of above synchronizations.

2.7.1 Top-of-Pipe Synchronization

The driver can use top-of-pipe synchronization to invalidate read-only caches in hardware. This operation is performed only after determining that no pending accesses from the hardware exist on these read-only buffers. PIPE-CONTROL aommnd described below allows for invalidating individual read-only buffer type. It is recommended that driver invalidates only the required caches on the need basis so that cache warm-up overhead can be reduced.

2.7.2 End-of-Pipe Synchronization

The driver can use end-of-pipe synchronization to know that rendering is complete (although not necessarily in memory) so that it can de-allocate in-memory rendering state, read-only surfaces, instructions, and constant buffers. An end-of-pipe synchronization point is also sufficient to guarantee that all pending depth tests have completed so that the visible pixel count is complete prior to storing it to memory. End-of-pipe completion is sufficient (although not necessary) to guarantee that read events are complete (a “read fence” completion). Read events are still pending if work in the pipeline requires any type of read except a render target read (blend) to complete.

Write synchronization is a special case of end-of-pipe synchronization that requires that the render cache and/or depth erlated caches are flushed to memory, where the data will become globally visible. This type of synchronization is required prior to SW (CPU) actually reading the result data from memory, or initiating an operation that will use as a read surface (such as a texture surface) a previous render target and/or depth/stencil buffer.

2.7.3 Sy nchronization Actions

In order for the driver to act based on a synchronization point (usually the whole point), the reaching of the synchronization point must be communicated to the driver. This section describes the actions that may be taken upon completion of a synchronization point which can achieve this communication.

2.7.3.1 Writing a Value to Memory

The most common action to perform upon reaching a synchronization point is to write a value out to memory. An immediate value (included with the synchronization command) may be written. In lieu of an immediate value, the 64-bit value of the PS_DEPTH_COUNT (visible pixel count) or TIMESTAMP register may be written out to memory. The captured value will be the value at the moment all primitives parsed prior to the synchronization commands have been completely rendered, and optionally after all said primitives have been pushed to memory. It is not required that a value be written to memory by the synchronization command.

Visible pixel or TIMESTAMP information is only useful as a delta between 2 values, because these counters are free-running and are not to be reset except at initialization. To obtain the delta, two PIPE_CONTROL commands should be initiated with the command sequence to be measured between them. The resulting pair of values in memory can then be subtracted to obtain a meaningful statistic about the command sequence.

2.7.3.1.1 PS_DEPTH_ COUNT

If the selected operation is to write the visible pixel count (PS_DEPTH_COUNT register), the synchronization command should include the **Depth Stall Enable** parameter. There is more than one point at which the global visible pixel count can be affected by the pipeline; once the synchronization command reaches the first point at which the count can be affected, any primitives following it are stalled at that point in the pipeline. This prevents the subsequent primitives from affecting the visible pixel count until all primitives preceding the synchronization point reach the end of the pipeline, the visible pixel count is accurate and the synchronization is completed. This stall has a minor effect on performance and should only be used in order to obtain accurate “visible pixel” counts for a sequence of primitives.

The PS_DEPTH_COUNT count can be used to implement an (API/DDI) “Occlusion Query” function.

2.7.3.2 Generating an Interrupt

The synchronization command may indicate that a “Sync Completion” interrupt is to be generated (if enabled by the MI Interrupt Control Registers – see *Memory Interface Registers*) once the rendering of all prior primitives is complete. Again, the completion of rendering can be considered to be when the internal render cache has been updated, or when the cache contents are visible in memory, as selected by the command options.

2.7.3.3 Invalidating of Caches

If software wishes to use the notification that a synchronization point has been reached in order to reuse referenced structures (surfaces, state, or instructions), it is not sufficient just to make sure rendering is complete. If additional primitives are initiated after new data is laid over the top of old in memory following a synchronization point, it is possible that stale cached data will be referenced for the subsequent rendering operation. In order to avoid this, the PIPE_CONTROL command must be used. (See PIPE_CONTROL description below).

2.7.4 PIPE_CONTROL Command

The PIPE_CONTROL command is used to effect the synchronization described above. Parsing of a PIPE_CONTROL command stalls 3D pipe only if the stall enable bit is set. Commands after PIPE_CONTROL will continue to be parsed and processed in the 3D pipeline. This may include additional PIPE_CONTROL commands. The implementation does enforce a practical upper limit [pre-DevCTG](4) [DevCTG][DevILK]on the number of PIPE_CONTROL commands that may be outstanding at once. Parsing of a PIPE_CONTROL command that causes this limit to be reached will stall the parsing of new commands until the first of the outstanding PIPE_CONTROL commands reaches the end of the pipe and retires.

Note that although PIPE_CONTROL is intended for use with the 3D pipe, it *is* legal to issue PIPE_CONTROL when the Media pipe is selected. In this case PIPE_CONTROL will stall at the top of the pipe until the Media FFs finish processing commands parsed before PIPE_CONTROL. Post-synchronization operations, flushing of caches and interrupts will then occur if enabled via PIPE_CONTROL parameters. Due to this stalling behavior, only one PIPE_CONTROL command can be outstanding at a time on the Media pipe.

[**DevCTG+**]: For the invalidate operation of the pipe control, the following pointers are affected. The invalidate operation affects the restore of these packets. If the pipe control invalidate operation is completed before the context save, the indirect pointers will not be restored from memory.

1. Pipeline State Pointer
2. Media State Pointer
3. Constant Buffer Packet

[pre-DevGT] PIPE_CONTROL will invalidate the Sampler and constant read caches unless the **Depth Stall Enable** bit is set. It will invalidate the Instruction/State cache if the **Instruction/State Cache Flush Enable** is set. .

If software wishes to access the rendered data in memory (for analysis by the application or to copy it to a new location to use as a texture, for examples), it must also ensure that the write cache (render cache) is flushed after the synchronization point is reached so that memory will be updated. This can be accomplished by setting the **Write Cache Flush Enable** bit. Note that the **Depth Stall Enable** bit must be clear in order for the flush of the render cache to occur. **Depth Stall Enable** is intended only for accurate reporting of the PS_DEPTH counter; the render cache cannot be flushed nor can the read caches be invalidated (except for the instruction/state cache) in conjunction with this operation.

[pre-DevGT] Both of the vertex caches will be flushed at the end of any PIPE_CONTROL operation regardless of how the control bits are set. [**DevCTG-B+**]: The **Texture Cache Flush Enable** bit controls whether the texture cache is flushed, irrespective of any other bit settings.

Table 2-5. Caches Invalidated/Flushed by PIPE_CONTROL Bit Settings

[Pre-DevCTG]

Depth Stall Enable	Write Cache Flush Enable	Inst/State Cache Flush Enable	Read (Sampler/Constant) Caches Inv'ed?	Write (Render) Cache Flushed?	Inst/State Cache Inv'ed?	Index-Based Vertex Cache Inv'ed?	VF Cache Inv'ed?	Stall Next Prim at Depth Stage?
0 0		0	Yes	No	No	Yes	Yes	No
0 0		1	Yes	No	Yes	Yes	Yes	No
0 1		0	Yes	Yes	No	Yes	Yes	No
0 1		1	Yes	Yes	Yes	Yes	Yes	No
1 X		0	No	No	No	Yes	Yes	Yes
1 X		1	No	No	Yes	Yes	Yes	Yes

[DevCTG]

Depth Stall Enable [13]	Write Cache Flush Enable [12]	Inst/State Cache Flush Enable [11]	Texture Cache FLush [10]	Read (Sampler/Constant) Caches Inv'ed? [1]	Write (Render) Cache Flushed?	Write (Depth) Cache flushed	Inst/State Cache Inv'ed?	Index-Based Vertex Cache Inv'ed?	VF Cache Inv'ed?	Stall Next Prim at Depth Stage?
0	0	0	0	No	No	No	No	No		No
0	0	0 1		Yes	No	No	No	Yes	Yes	No
0	0	1 x		Yes	No	No	Yes	Yes	Yes	No
0	1	0	x	Yes	Yes	Yes	No	Yes	Yes	No
0	1	1	x	Yes	Yes	Yes	Yes	Yes	Yes	No
1	X	x	x	No	No	No	No	Yes	Yes	Yes

[DevILK]

Depth Stall Enable	Write Cache Flush Enable	Inst/State Cache Flush Enable	Texture Cache FLush	Z-inhibit	FC-stall	Read (Sampler/Constant) Caches Inv'ed?	Write (Render) Cache Flushed?	Write (Depth) Cache flushed	Inst/State Cache Inv'ed?	Index-Based Vertex Cache Inv'ed?	Stall Next Primat thread dispatch?	VF Cache Inv'ed?	Stall Next Primat Depth Stage?
0	0	0 0		x 0		No	No	No	No	No	No	No	No
0	0	0 0		x 1		No	No	No	No	No	Yes	No	No
0	0	0 1		x x		Yes	No	No	No	Yes	No	Yes	No
0	0	1 x		x x		Yes	No	No	Yes	Yes	No	Yes	No
0	1	0 x		0 x		Yes	Yes	Yes	No	Yes	No	Yes	No
0	1	1 x		0 x		Yes	Yes	Yes	Yes	Yes	No	Yes	No
0	1	0 x		1 x		Yes	Yes	No	No	Yes	No	Yes	No
0	1	1 x		1 x		Yes	Yes	No	Yes	Yes	No	Yes	No
1	x	x x		x x		No	No	No	No	Yes	No	Yes	Yes

2.7.4.1 [Pre-DevSNB]

PIPE_CONTROL			
Project:	[Pre-DevSNB]	**Length Bias:**	2
The PIPE_CONTROL command is used to effect the synchronization described above.			

DWord	Bit	Description
0	31:29	**Command Type** **Default Value: 3h GFXPIPE Format: OpCode**
	28:27	**Command SubType** **Default Value: 3h GFXPIPE_3D Format: OpCode**
	26:24	**3D Command Opcode** **Default Value: 2h PIPE_CONTROL Format: OpCode**
	23:16	**3D Command Sub Opcode** **Default Value: 00h Format: OpCode**
	15:14	**Post-Sync Operation** **Project: All** **This field specifies an optional action to be taken upon completion of the synchronization operation.**

Value	Name	Description	Project
0h	No Write	No write occurs as a result of this instruction. This can be used to implement a "trap" operation, etc.	All
1h	QWord Write	Write the QWord containing Immediate Data Low, High DWs to the Destination Address	All
2h	PS Depth Count	Write the 64-bit PS_DEPTH_COUNT register to the Destination Address	All
3h	Timestamp	Write the 64-bit TIMESTAMP register to the Destination Address	All

Errata	Description	Project
#	PS_DEPTH_COUNT cannot be accurately sampled using this command. Setting this field to 2 will write an UNDEFINED value rather than the accurate PS_DEPTH_COUNT.	BW-A,B

PIPE_CONTROL		
	13	**Depth Stall Enable Project: All Format: Enable** **If ENABLED, the 3D pipeline will stall any subsequent primitives at the Depth Test stage until the Sync and Post-Sync operations complete.** **If DISABLED, the 3D pipeline will not stall subsequent primitives at the Depth Test stage.** **This bit should be set when obtaining a "visible pixel" count to preclude the possible inclusion in the PS_DEPTH_COUNT value written to memory of some fraction of pixels from objects initiated after the PIPE_CONTROL command.** **Programming Notes:** • **This bit should be DISABLED for operations other than writing PS_DEPTH_COUNT.** • **This bit will have no effect (besides preventing write cache flush) if set in a PIPE_CONTROL command issued to the Media pipe.**
	12	**Write Cache Flush Enable Project: All Format: Enable** **Setting this bit will force Render Cache to be flushed to memory prior to this synchronization point completing. This bit should be set for all write fence Sync operations to assure that results from operations initiated prior to this command are visible in memory once software observes this synchronization.** **This bit should be DISABLED for End-of-pipe (Read) fences, PS_DEPTH_COUNT or TIMESTAMP queries. This bit is ignored if Depth Stall Enable is set; the Render Cache will not be flushed even if Write Cache Flush Enable is set.**
	11	**Instruction/State Cache Flush Enable Project: All Format: Enable** **Setting this bit is independent of any other bit in this packet. This bit controls the invalidation of the L1 and L2 instruction/state caches after the completion of the flush.** **[DevILK] : This bit must not be set.**
	10	**Texture Cache Flush Enable Project: CTG+ Format: Enable** **Setting this bit is independent of any other bit in this packet. This bit controls the invalidation of the texture caches after the completion of the flush.**
	10	**Reserved Project: Pre-CTG Format: MBZ**
	9	**Indirect State Pointers Disable Project: CTG+ Format: Disable** **At the completion of the post-sync operation associated with this pipecontrol packet, the indirect state pointers in the hardware will be considered as invalid ie the indirect pointers will not be saved in the context. If any new indirect state commands are executed in the command stream while the pipe control is pending, the new indirect state commands will be preserved.**
	9	**Reserved Project: Pre-CTG Format: MBZ**

PIPE_CONTROL

	8	**Notify Enable** **Project: A ll** **Format: Enable** **If ENABLED, a Sync Completion Interrupt will be generated (if enabled by the MI Interrupt Control registers) once the sync operation is complete. See Interrupt Control Registers in Memory Interface Registers for details.**
	7:0	**DWord Length** **Default Value: 2h Excludes DWord (0,1)** **Format: =n Total Length - 2** **Project: A ll**
1	31:3	**Destination Address** **Project: A ll** **Address: Grap hicsAddress[31:3]** **QW-aligned graphics memory address at which data will be written when sync point occurs. Ignored if Post-Sync Operation is "No write".**
	2	**Destination Address Type** **Project: A ll** **Defines address space of Destination Address.** **Value Na me — Description — Project** **0h — Local PGTT — Use process local PGTT — All** **1h — Global GTT — Use Global GTT; valid only for privileged commands — All** **Programming Notes — Project** **Ignored if "No write" is the selected in Operation. — All**
	1:0	**Reserved Project: Pre-Dev-ILK Format: MBZ**
	1	**Stall At Pixel Scoreboard** **Project: De vILK** **Defines the behavior of PIPE_CONTROL command at the pixel scoreboard.** **Value Na me — Description — Project** **0h — Disabled — Stall at the pixel scoreboard is disabled. — DevILK** **1h — Enabled — Stall at the pixel scoreboard is enabled. — DevILK** **Programming Notes — Project** **This bit should be DISABLED for End-of-pipe (Read) fences, PS_DEPTH_COUNT or TIMESTAMP queries. This bit is ignored if Depth Stall Enable is set. Further the render cache is not flushed even if Write Cache Flush Enable bit is set. — DevILK**

	0	**Depth Cache Flush Inhibit** **Project: De vILK** **Setting this bit prevents flushing (i.e. writing back the dirty lines to memory and invalidating the tags) of depth related caches even if the Write Cache Flush is enabled. This bit applies to HiZ cache, Stencil cache and depth cache.** **Value Na me — Description — Project** **0h Flushed — Depth relates caches (HiZ, Stencil and Depth) are flushed when Write Flush is Enabled. — DevILK** **1h Not Flushed — Depth relates caches (HiZ, Stencil and Depth) are NOT flushed when Write Flush is Enabled. — DevILK** **Programming Notes — Project** **This bit is ignored when Write Cache Flush is not enabled. Ideally depth caches need to be flushed only when depth is required to be coherent in memory for later use as a texture, source or honoring CPU lock. — DevILK** Errata | Description | Project # | This bit must be disabled. | DevILK
2	31:0	**Immediate Data Low DW — Project: A II — Format: U32** **Low DW of QW value to write to memory at synchronization point. Ignored if Post-Sync Operation is "No write", "Write PS_DEPTH_COUNT" or "Write TIMESTAMP".**
3	31:0	**Immediate Data High DW — Project: A II — Format: U32** **High DW of QW value to write to memory at synchronization point. Ignored if Post-Sync Operation is "No write", "Write PS_DEPTH_COUNT" or "Write TIMESTAMP".**

2.7.4.1.1 Post-Sy nc Operation

These are arguments related to events that occur *after* the marker initiated by the PIPE_CONTROL command is completed. The table below shows the restrictions:

Arguments Bit		**Restrictions**
Protected Mem Enable	22	Requires stall bit ([20] of DW1) set.
Global Snapshot Count Reset	19	Requires stall bit ([20] of DW1) set.
Generic Media State Clear	16	Requires stall bit ([20] of DW1) set.
Indirect State Pointers Disable	9	Requires stall bit ([20] of DW1) set.
Store Data Index	21	Post-Sync Operation ([15:14] of DW1) must be set to something other than '0'
Sync GFDT	17	Post-Sync Operation ([15:14] of DW1) must be set to something other than '0' or 0x2520[13] must be set

Arguments Bit		Restrictions
TLB inv	18	Post-Sync Operation ([15:14] of DW1) must be set to something other than '0'. Already implied when 0x2520[13] is set
Post Sync Op	15:14	No Restriction
Notify En	8	No Restriction

2.7.4.1.2 Flush Types

These are arguments related to the type of read only invalidation or write cache flushing is being requested. Note that there is only intra-dependency. That is, it is not affected by the post-sync operation or the stall bit. The table below shows the restrictions:

Arguments Bit		Restrictions
Depth Stall	13	Following bits must be ***clear*** • Render Target Cache Flush Enable ([12] of DW1) • Depth Cache Flush Enable ([0] of DW1) • Stall at Pixel Scoreboard. ([1] of DW1)
Render Target Cache Flush	12	Depth Stall must be clear ([13] of DW1)
Depth Cache Flush	0	Depth Stall must be clear ([13] of DW1)
Stall Pixel Scoreboard	1	Depth Stall must be clear ([13] of DW1)
Inst invalidate.	11	No Restriction
Tex invalidate.	10	No Restriction
VF invalidate	4	No Restriction
Constant invalidate	3	No Restriction
State Invalidate	2	No Restriction

2.7.4.1.3 Stall

If the stall bit is set, the command streamer waits until the pipe is completely flushed.

Arguments Bit		Restrictions
Stall Bit	20	1 of the following must also be set • Render Target Cache Flush Enable ([12] of DW1) • Depth Cache Flush Enable ([0] of DW1) • Stall at Pixel Scoreboard ([1] of DW1) • Depth Stall ([13] of DW1) • Post-Sync Operation ([13] of DW1) • Notify Enable ([8] of DW1)
Errata	Description	Project
#	This bit must be disabled.	DevILK]

3. Vertex Fetch (VF) Stage

3.1 Vertex Fetch (VF) Stage Overview

The VF stage performs one major function: executing 3DPRIMITIVE commands. This is handled by the VF's InputAssembly function. The following subsections describe some high-level concepts associated with the VF stage.

3.1.1 Input Assembly

The VF's InputAssembly function includes (for each vertex generated):

- Generation of VertexIndex and InstanceIndex for each vertex, possibly via use of an Index Buffer.
- Lookup of the VertexIndex in the Vertex Cache (if enabled)
- If a cache miss is detected:
  - Use of computed indices to fetch data from memory-resident vertex buffers
  - Format conversion of the fetched vertex data
  - Assembly of the format conversion results (and possibly some internally generated data) to form the complete "input" (raw) vertex
  - Storing the input vertex data in a Vertex URB Entry (VUE) in the URB
  - Output of the VUE handle of the input vertex

3.1.1.1 Vertex Assembly

The VF utilizes a number of VERTEX_ELEMENT state structures to define the contents and format of the vertex data to be stored in Vertex URB Entries (VUEs) in the URB. See below for a detailed description of the command used to define these structures (3DSTATE_VERTEX_ELEMENTS).

Each active VERTEX_ELEMENT structure defines up to 4 contiguous DWords of VUE data, where each DWord is considered a "component" of the vertex element. The starting destination DWord offset of the vertex element in the VUE is specified, and the VERTEX_ELEMENT structures must be defined with monotonically increasing VUE offsets. For each component, the source of the component is specified. The source may be a constant (0, 0x1, or 1.0f), a generated ID (VertexID, InstanceID or PrimitiveID), or a component of a structure in memory (e.g,. the Y component of an XYZW position in memory). In the case of a memory source, the Vertex Buffer sourcing the data, and the location and format of the source data with that VB are specified.

The VF's Vertex Assembly process can be envisioned as the VF unit stepping through the VERTEX_ELEMENT structures in order, fetching and format-converting the source information (if memory resident), and storing the results in the destination VUE.

3.1.2 Vertex Cache

The Vertex Cache is strictly a performance-enhancing feature and has no impact on 3D pipeline results (other than a few statistics counters).

In addition, any non-trivial use of instancing (i.e., more than one instance per 3DPRIMITIVE command and the inclusion of instance data in the input vertex) will effectively invalidate the cache between instances, as the `InstanceIndex` is not included in the cache tag.

3.1.3 Input Data: Push Model vs. Pull Model

Given the programmability of the pipeline, and the ability of shaders to input (load/sample) data from memory buffers in an arbitrary fashion, the decision arises in whether to push instance/vertex data into the front of the pipeline or defer the data access (pull) to the shaders that require it. An incrementing *VertexID*, *InstanceID* and *PrimitiveID* are generated in the Input Assembly process, and these values can be declared as input to the "first enabled, relevant" shader. That shader can, for example, use the HW-generated ID as an index into a memory resource such as a constant buffer or vertex buffer. The GENx 3D pipeline supports these IDs as required by the API.

There are tradeoffs involved in deciding between these models. For vertex data, it is probably always better to push the data into the pipeline, as the VF hardware attempts to cover the latency of the data fetch. The decision is less clear for instance data, as pushing instance data leads to larger Vertex URB entries which will be holding redundant data (as the instance data for vertices of an object are by definition the same). .

3.1.4 Generated IDs

The VF generates InstanceID, VertexID, and PrimitiveID values as part of the InputAssembly process. No special considerations for OpenGL have been included.

Note that the generated IDs are considered separate from any offset computations performed by the VF unit, and are therefore described separately here.

The InstanceID, VertexID, and PrimitiveID values associated with each vertex can be stored in the vertex's VUE, via use of the **Component *n* Control** fields in the VERTEX_ELEMENT structure.

The definition/use of PrimitiveID is more complicated than the other auto-generated IDs. It is only available to the GS as a special non-vertex input, and the PS as a constant-interpolated attribute. The PrimitiveID therefore should be kept separate from the vertex data. Take for example a TRILIST primitive topology: It should be possible to share vertices between triangles in the list even though each triangle has a different PrimitiveID associated with it.

3.2 VF Stage Input

As a stage of the GENx 3D pipeline, the VF stage receives inputs from the previous (CS) stage.

The VF stage gets its state programmed directly via pipelined state commands. It does not support indirect state descriptors.

The following table lists the 3D pipeline commands processed by the VF stage. Other commands (not listed) are simply passed down the pipeline. Refer to *3D Overview* for an overview of the various types of input to a 3D Pipeline stage.

Command De	scription
Processing Commands	
3DPRIMITIVE	This primitive command is used to inject primitives into the 3D pipeline, where they will be processed according to the current context state settings. Most typically this processing will result in rendering to destination surfaces, though this is not required. This command is defined in the *VF Stage* chapter (as it is executed there), though the processing of this command includes the entire 3D pipeline.
Pipelined State Commands	
3DSTATE_INDEX_BUFFER	This pipelined state command is used to specify Index Buffer parameters used in the VF unit's InputAssembly function. An Index Buffer can be used to provide vertex indices when processing subsequent 3DPRIMITIVE commands. This command does not travel past the VF stage. See *Index Buffer* below for details on this command.
3DSTATE_VERTEX_BUFFERS	This pipelined state command is used to specify Vertex Buffer parameters used in the VF unit's InputAssembly function. Vertex Buffers provide vertex data when processing subsequent 3DPRIMITIVE commands. This command does not travel past the VF stage. See *Vertex Buffers* below for details on this command.
3DSTATE_VERTEX_ELEMENTS	This pipelined state command is used to specify Vertex Element parameters used in the VF unit's InputAssembly function. Vertex Element parameters specify how vertex data, extracted from Vertex Buffers, are format converted and stored in VUEs. This command does not travel past the VF stage. See *Input Vertex Data* below for details on this command.
3DSTATE_VF_STATISTICS	This pipelined state command is used to turn pipeline statistics gathering by the VF stage on or off. This command does not travel past the VF stage. See *Statistics Gathering* below for details on this command.

The VF stage also receives input directly from memory, in the form of Index Buffers and Vertex Buffers.

3.3 Index Buffer (IB)

The 3DSTATE_INDEX_BUFFER command is used to define an *Index Buffer* (IB) used in subsequent 3DPRIMITIVE commands.

The RANDOM access mode of the 3DPRIMITIVE command involves the use of a memory-resident IB. The IB, defined via the 3DSTATE_INDEX_BUFFER command described below, contains a 1D array of 8, 16 or 32-bit index values. These index values will be fetched by the InputAssembly function, and subsequently used to compute locations in VERTEXDATA buffers from which the actual vertex data is to be fetched. (This is opposed to the SEQUENTIAL access mode were the vertex data is simply fetched sequentially from the buffers).

Software is responsible for ensuring that accesses outside the IB do not occur. This is possible as software can compute the range of IB values referenced by a 3DPRIMITIVE command (knowing the **StartVertexLocation**, **InstanceCount**, and **VerticesPerInstance** values) and can then compare this range to the IB extent.

3.3.1 3DSTATE_INDEX_BUFFER

3DSTATE_INDEX_BUFFER

Project:	All	Length Bias:	2

This command is used to specify the current IB state used by the VF function. At most one IB is defined and active at any given time.

NOTES:

- The IB must be specified before any RANDOM 3D_PRIMITIVE commands are issued
- It is possible to have vertex elements source completely from generated ID values and therefore not require any Index Buffer accesses. In this case, VF function will simply ignore the Index Buffer state.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 0h 3DSTATE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 0Ah 3DSTATE_INDEX_BUFFER Format: OpCode
	11	**Reserved** Project: All Format: MBZ
	10	**Cut Index Enable** Project: All Format: Enable FormatDesc If ENABLED, the largest index value (0xFF,0xFFFF,0xFFFFFFFF, depending on Index Format) is interpreted as the “cut” index. (See description of this elsewhere in this section). If DISABLED, there is no special “cut” index value, and the largest index value is simply used as an index. (Expected OpenGL driver usage) This field can only be enabled for certain primitive topology types. Refer to the table later in this section for details.
	9:8	**Index Format** Project: All Format: U2 enumerated type FormatDesc This field specifies the data format of the index buffer. All index values are UNSIGNED.

Value Name	Description	Project
0h	INDEX_BYTE	All
1h	INDEX_WORD	All
2h	INDEX_DWORD	All

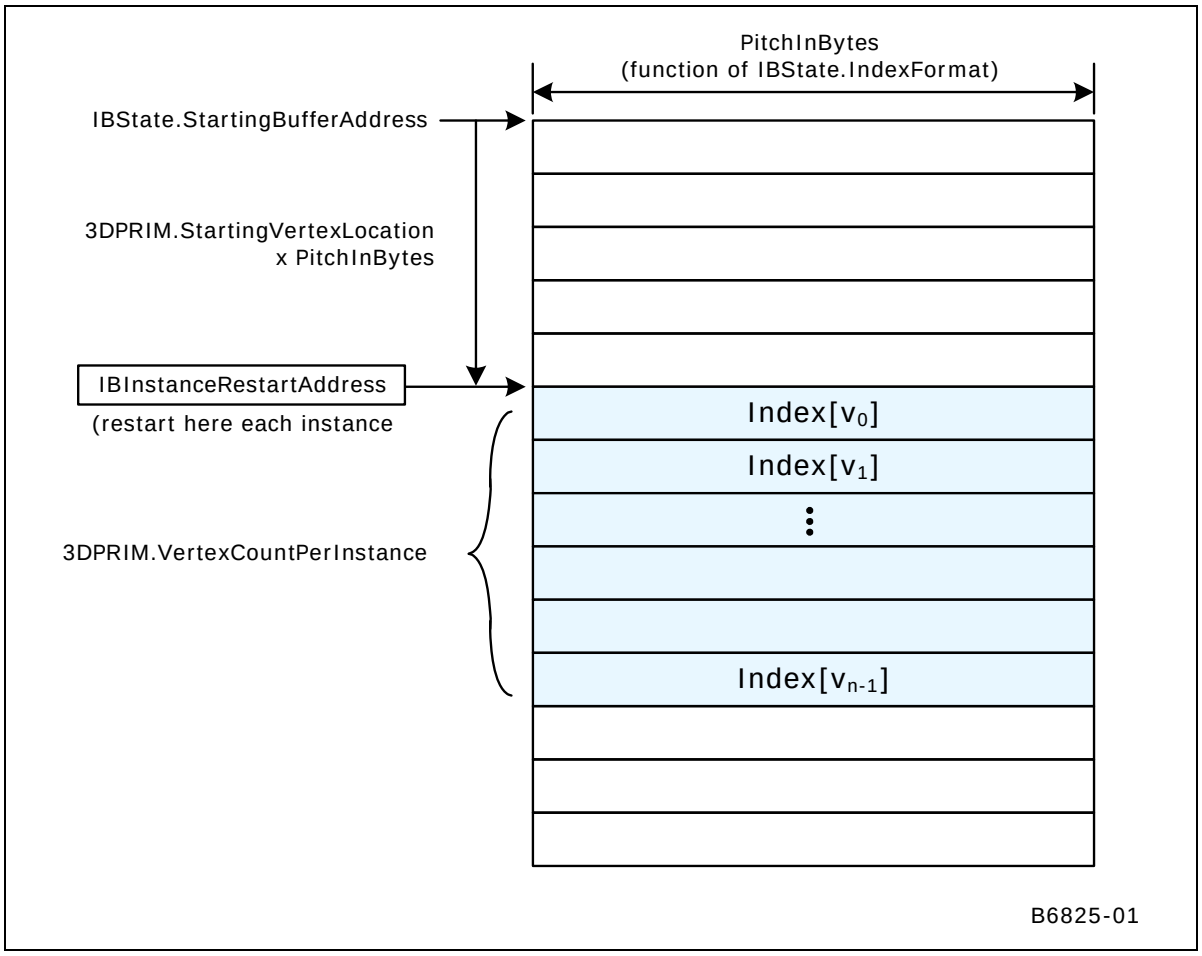
3DSTATE_INDEX_BUFFER		
	7:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:0	**Buffer Starting Address** Project: All Address: GraphicsAddress[31:0] Surface Type: Index Buffer Entry This field contains the size-aligned (as specified by **Index Format**) Graphics Address of the first element of interest within the index buffer. Software must program this value with the combination (sum) of the base address of the memory resource and the byte offset from the base address to the starting structure within the buffer. **Programming Notes** Index Buffers can only be allocated in linear (not tiled) graphics memory Index Buffers can only be mapped to Main Memory (UC). They must not be mapped to snooped System Memory, or UNPREDICTABLE values may be read
2	31:0	**Buffer Ending Address** Project: All Address: GraphicsAddress[31:0] If non-zero, this field contains the address of the last valid byte in the index buffer. Any index buffer reads past this address returns an index value of 0 (as if the index buffer was zero-extended). Software must guarantee that the buffer ends on an index boundary (e.g., for an INDEX_DWORD buffer, Bits [1:0] == 11b). [Pre-DevSNB]: If this field is zero, no bounds checking is performed. All indices will be read directly from memory. [**DevBW**, **DevCL**] Errata: In order to enable bounds checking, this field must be programmed to point to the last byte of a 64B cacheline (Bits[5:0] == 111111b), otherwise an UNDEFINED index value may be returned for accesses past the end address. Note, however, that Vertex Buffer data accesses using these UNDEFINED index values are still correctly clamped to the Max Index of the Vertex Buffer.

The following table lists which primitive topology types support the presence of Cut Indicies. When the Index Buffer has **Cut Index Enable** set, it is UNDEFINED to issue a 3DPRIMITIVE with a primitive topology type not supporting a Cut Index (even if no cut indicies are actually present in the index buffer).

Definition	Cut Index?
3DPRIM_POINTLIST	Y
3DPRIM_LINELIST	Y
3DPRIM_LINESTRIP	Y
3DPRIM_TRILIST	Y
3DPRIM_TRISTRIP	Y
3DPRIM_TRIFAN	N
3DPRIM_QUADLIST	N
3DPRIM_QUADSTRIP	N
3DPRIM_LINELIST_ADJ	Y
3DPRIM_LINESTRIP_ADJ	Y
3DPRIM_TRILIST_ADJ	Y
3DPRIM_TRISTRIP_ADJ	Y
3DPRIM_TRISTRIP_REVERSE	Y
3DPRIM_POLYGON	N
3DPRIM_RECTLIST	N
3DPRIM_LINELOOP	N
3DPRIM_POINTLIST_BF	Y
3DPRIM_LINESTRIP_CONT	Y
3DPRIM_LINESTRIP_BF	Y
3DPRIM_LINESTRIP_CONT_BF	Y
3DPRIM_TRIFAN_NOSTIPPLE	N

3.3.2 Index Buffer Access

The figure below illustrates how the Index Buffer is accessed.



3.4 Vertex Buffers (VBs)

The 3DSTATE_VERTEX_BUFFERs and 3DSTATE_INSTANCE_STEP_RATE commands are used to define *Vertex Buffers* (VBs) used in subsequent 3DPRIMITIVE commands.

Most input vertex data is sourced from memory-resident VBs. A VB is a 1D arrays of structures, where the size of the structure as defined by the VB's **BufferPitch**. VBs are accessed either as *VERTEXDATA buffers* or *INSTANCEDATA buffers*, as defined by the VB's **BufferAccessType**. The VB's access type will determine whether the VF-computed VertexIndex or InstanceIndex is used to access data in the VB.

Given that the RANDOM access mode of the 3DPRIMITIVE command utilizes an IB (possibly provided by an application) to compute VB index values, VB definitions contain a **MaxIndex** value used to detect accesses beyond the end of the VBs. Any access outside the extent of a VB returns 0.

3.4.1 3DSTATE_VERTEX_ BUFFERS

This command is used to specify VB state used by the VF function. From 1 to 17 VBs[Pre-DevSNB], 33..

NOTES:

- It is possible to have individual vertex elements sourced completely from generated ID values and therefore not require any vertex buffer accesses for that vertex element. In this case, VF function will simply ignore the VB state associated with that vertex element. If all enabled vertex elements have this characteristic, no VBs are required to process 3DPRIMITIVE commands. For example, this might arise when the user wants to perform all data lookups in the first shader, so only generated index values need to be passed down to it. In this extreme case, SW would not need to program any VB state, and therefore not need to issue any 3DSTATE_VERTEX_BUFFERS commands.

- For any 3DSTATE_VERTEX_BUFFERS command, at least one VERTEX_BUFFER_STATE structure must be included.

- VERTEX_BUFFER_STATE structures are 4 DWords for both VERTEXDATA buffers and INSTANCEDATA buffers.

- Inclusion of partial VERTEX_BUFFER_STATE structures is UNDEFINED.

The order in which VBs are defined within this command can be arbitrary, though a vertex buffer must be defined only once in any given command (otherwise operation is UNDEFINED).

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 03h
	28:16	**GFXPIPE Opcode =** 3DSTATE_VERTEX_BUFFERS GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 08h] (Pipelined)
	15:8	Reserved : MBZ
	7:0	**DWord Length** (excludes DWords 0,1) 4n-1 (where n = # of buffer states included)
1-4		**Vertex Buffer State [0]** Format: VERTEX_BUFFER_STATE
5-8		**Vertex Buffer State [1]**
…		…
(4n-3)-(4n)		**Vertex Buffer State [..]**

3.4.2 VERTEX_BUFFER_STATE Structure

VERTEX_BUFFER_STATE

Project: All

This structure is used in 3DSTATE_VERTEX_BUFFERS to set the state associated with a VB. The VF function will use this state to determine how/where to extract vertex element data for all vertex elements associated with the VB.

The VERTEX_BUFFER_STATE structure is 4 DWords for both INSTANCEDATA and VERTEXDATA buffers. A VB is defined as a 1D array of vertex data structures, accessed via a computed index value. The VF function therefore needs to know the starting address of the first structure (index 0) and size of the vertex data structure. [DevILKIf an index value outside of the range [0,Max Index] is used to access this vertex buffer, the value 0 is returned. [DevILK] Vertex element accesses which straddle or go past the VB's End Address will return 0's for all elements.

DWord	Bit	Description
0	31:27	**[Pre-DevGT]Vertex Buffer Index** Project: All Format: U5 index Range [0,16] This field contains an index value which selects the VB state being defined.

VERTEX_BUFFER_STATE

	31:26	**Vertex Buffer Index** Project: All Format: U6 index FormatDesc Address: GraphicsAddress[31:0] Range [0,32] This field contains an index value which selects the VB state being defined.
	26	**[Pre-DevGT]Buffer Access Type** Project: All This field determines how vertex element data is extracted from this VB. This control applies to all vertex elements associated with this VB. .
	25:21	**Reserved** Project: All Format: MBZ
	15	**Reserved** Project: All Format: MBZ
	13	**[DevILK] Null Vertex Buffer.** Format: Enable FormatDesc This field enabled causes any fetch for vertex data to return 0.

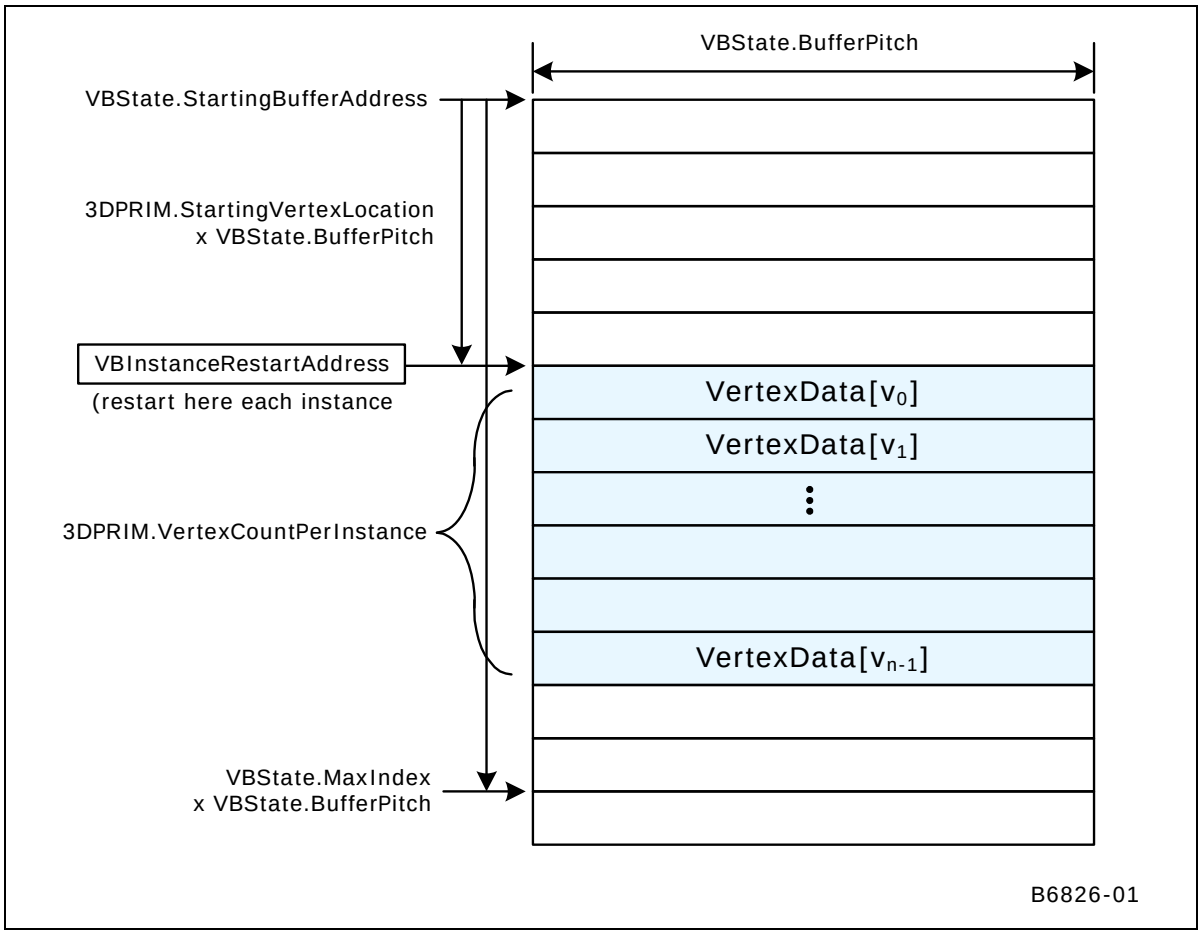
Value Name	Description	Project
0 VERTEXDATA	For SEQUENTIAL vertex access, each vertex of an instance is sourced from sequential structures within the VB. For RANDOM vertex access, each vertex of an instance is looked up (separately) via a computed index value.	All
1 NSTANCEDATA	Each vertex of an instance is sourced with the same (instance) data. Subsequent instances may be sourced with the same or different data, depending on **Instance Data Step Rate**.	All

VERTEX_BUFFER_STATE		
	12	**[DevCTG+]** Project: All Security: None Access: None Exists If: Always Default Value: 0h DefaultVaueDesc Mask: MMIO(0x2000)#16 Format: U32 FormatDesc Address: GraphicsAddress[31:0] Surface Type: U32 Range 0..2^32-1 Invalidate the Vertex overfetch cache when this bit is set. For multiple vertex buffer state structures in one packet, this bit may be set only once in the entire packet. [Pre-DevCTG]: Reserved
	11:0	**Buffer Pitch** Format: U12 FormatDesc Count of bytes Range [Pre-DevCTG]: [0,2047] Bytes [DevCTG+]: [0,2048] Bytes This field specifies the pitch in bytes of the structures accessed within the VB. This information is required in order to access elements in the VB via a structure index. **Programming Notes** Different VERTEX_BUFFER_STATE structures can refer to the same memory region using different Buffer Pitch values. See note on 64-bit float alignment in Buffer Starting Address.

VERTEX_BUFFER_STATE

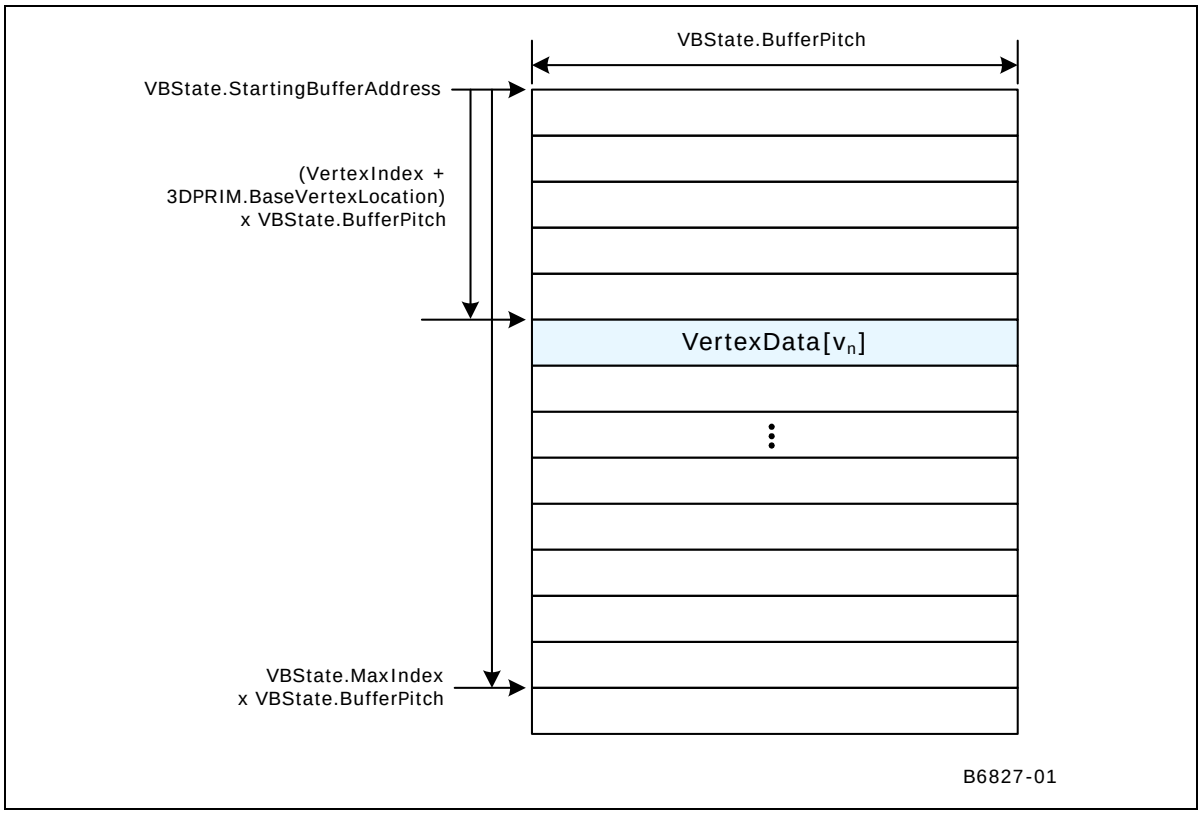
1	31:0	**Buffer Starting Address** Format: GraphicsAddress[31:0] FormatDesc Address: GraphicsAddress[31:0] This field contains the byte-aligned Graphics Address of the first element of interest within the VB. Software must program this value with the combination (sum) of the base address of the memory resource and the byte offset from the base address to the starting structure within the buffer.[**Programming Notes** 64-bit floating point values must be 64-bit aligned in memory, or UNPREDICTABLE data will be fetched. When accessing an element containing 64-bit floating point values, the Buffer Starting Address and Source Element Offset values must add to a 64-bit aligned address, and BufferPitch must be a multiple of 64-bits. VBs can only be allocated in linear (not tiled) graphics memory. As computed index values are, by definition, interpreted as unsigned values, there is no issue with accesses to locations before (lower address value) the start of the buffer. However, these wrapped indices are subject to Max Index checking (see below).
2	31:0	**End Address** Project: All Security: None Default Value: 0h DefaultVaueDesc Mask: MMIO(0x2000)#16 Format: U32 FormatDesc Address: GraphicsAddress[31:0] Surface Type: U32 Range 0..2^32-1 [HVNABD] This field defines the address of the last valid byte in this particular VB. Access of a vertex element which either straddles or is beyond this address will return 0's for any data read.
2	31:0	**Max Index [Pre-DevILK]** Format: U32 FormatDesc If non-zero (bounds-checking enabled), this field defines the maximum (inclusive) structure index accessible for this particular VB. Use of an index larger than the Max Index returns 0 for all components. This includes a "negative" computed index which, when viewed as an unsigned value, exceeds Max Index. If zero, bounds checking is disabled. A read from the vertex buffer memory is performed regardless of the computed index. **Programming Notes** The smallest vertex buffer that can be bounds-checked is a 2-entry buffer (where MaxIndex is programmed to 1).

VERTEX_BUFFER_STATE		
3	31:0	**Instance Data Step Rate:** Format: U32 FormatDesc This field only applies to INSTANCEDATA buffers – it is ignored (but still present) for VERTEXDATA buffers). This field determines the rate at which instance data for this particular INSTANCEDATA vertex buffer is changed in sequential instances. Only after the number of instances specified by this field is generated is new (sequential) instance data provided. This process continues for each group of instances defined in the draw command. For example, a value of 1 in this field causes new instance data to be supplied with each sequential (instance) group of vertices. A value of 2 causes every other instance group of vertices to be provided with new instance data. The special value of 0 causes all vertices of all instances generated by the draw command to be provided with the same instance data. (The same effect can be achieved by setting this field to its maximum value.)

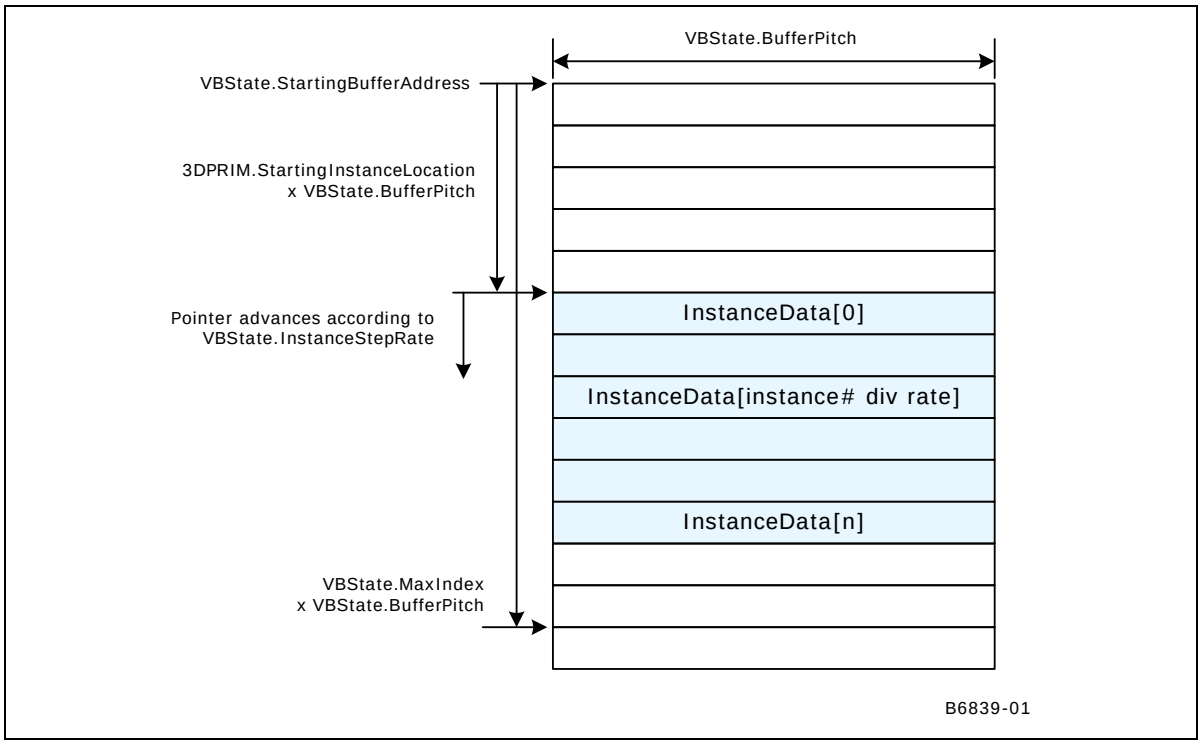


3.4.3 VERTEXDATA Buffers – SEQUENTIAL Access

3.4.4 VERTEXDATA Buffers – RANDOM Access



3.4.5 INSTANCEDATA Buffers



3.5 Input Vertex Definition

The 3DSTATE_VERTEX_ELEMENTS command is used to define the source and format of input vertex data and the format of how it is stored in the destination VUE as part of 3DPRIMITIVE processing in the VF unit.

Refer to *3DPRIMITIVE Processing* below for the general flow of how input vertices are input and stored during processing of the 3DPRIMITIVE command.

3.5.1 3DSTATE_VERTEX_ ELEMENTS

This is a variable-length command used to specify the active vertex elements (up to 18[PreDevSNB], 34.

RESTRICTIONS/NOTES:

- At least one VERTEX_ELEMENT_STATE structure must be included.
- **[Pre-DevILK]** Vertex elements must be ordered by increasing **Destination Element Offset**.
- Inclusion of partial VERTEX_ELEMENT_STATE structures is UNDEFINED.
- SW must ensure that at least one vertex element is defined prior to issuing a 3DPRIMTIVE command, or operation is UNDEFINED.
- There are no 'holes' allowed in the destination vertex: NOSTORE components must be overwritten by subsequent components unless they are the trailing DWords of the vertex. Software must explicitly chose some value (probably 0) to be written into DWords that would otherwise be 'holes'.
- (See additional restrictions listed in the command fields and VERTEX_ELEMENT_STATE description).
- **[DevILK+]** Element[0] must be valid.
- **[DevILK+]** All elements must be valid from Element[0] to the last valid element. (i.e. if Element[2] is valid then Element[1] and Element[0] must also be valid)
- **[DevILK+]** The pitch between elements packed in the URB will always be 128 bits.

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 03h
	28:16	**GFXPIPE Opcode =** 3DSTATE_VERTEX_ELEMENTS GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 09h] (Pipelined)
	15:8	Reserved : MBZ
	7:0	**DWord Length** (excludes DWords 0,1) Vertex Element Count = (**DWord Length** + 1) / 2
1-2		**Element[0]** Format: VERTEX_ELEMENT_STATE
[3-4]		**Element[1]**
…		…
[35-36]		**Element[17]**

3.5.2 VERTEX_ELEM ENT_STATE Structure

3.5.3 VERTEX_ELEM ENT_STATE Structure

3DSTATE_InstructionName

Project:	All	**Length Bias:**	2

This structure is used in 3DSTATE_VERTEX_ELEMENTS to set the state associated with a *vertex element*. A vertex element is defined as an entity supplying from 1 to 4 DWord vertex components to be stored in the vertex URB entry. Up to 18 vertex elements are supported. The VF function will use this state, and possibly the state of the associated vertex buffer, to fetch/generate the source vertex element data, perform any required format conversions, padding with zeros, and store the resulting destination vertex element data into the vertex URB entry.

DWord	Bit	Description
0	31:27	**Vertex Buffer Index** Project: [Pre-DevGT] Format: U5 FormatDesc Range [0,16] (Up to 17 VBs are supported) This field specifies which vertex buffer the element is sourced from. **Programming Notes** It is possible for a vertex element to include only internally-generated data (VertexID, etc.), in which case the associated vertex buffer state is ignored.
	26	**Valid** Project: [Pre-DevGT] Format: Boolean FormatDesc

Value Name	Description	Project
0h	True	this vertex element is used in vertex assembly	[Pre-DevGT]
1h	False	this vertex element is not used.	[Pre-DevGT]

25	**Reserved**	
15	**Reserved** Project: [Pre-DevGT]	Format: MBZ
14:11	**Reserved** Project: All	Format: MBZ

3DSTATE_InstructionName		
	10	**Source Element Offset (in bytes)** Project: All Format: U11 byte offset Range [0,2047 Byte offset of the source vertex element data in the structures comprising the vertex buffer. **Programming Notes** See note on 64-bit float alignment in Buffer Starting Address.
1	31	**Reserved** Project: All Format: MBZ

	30:28	**Component 0 Control** Project: All This field specifies which value is stored for component 0 of this particular vertex element.

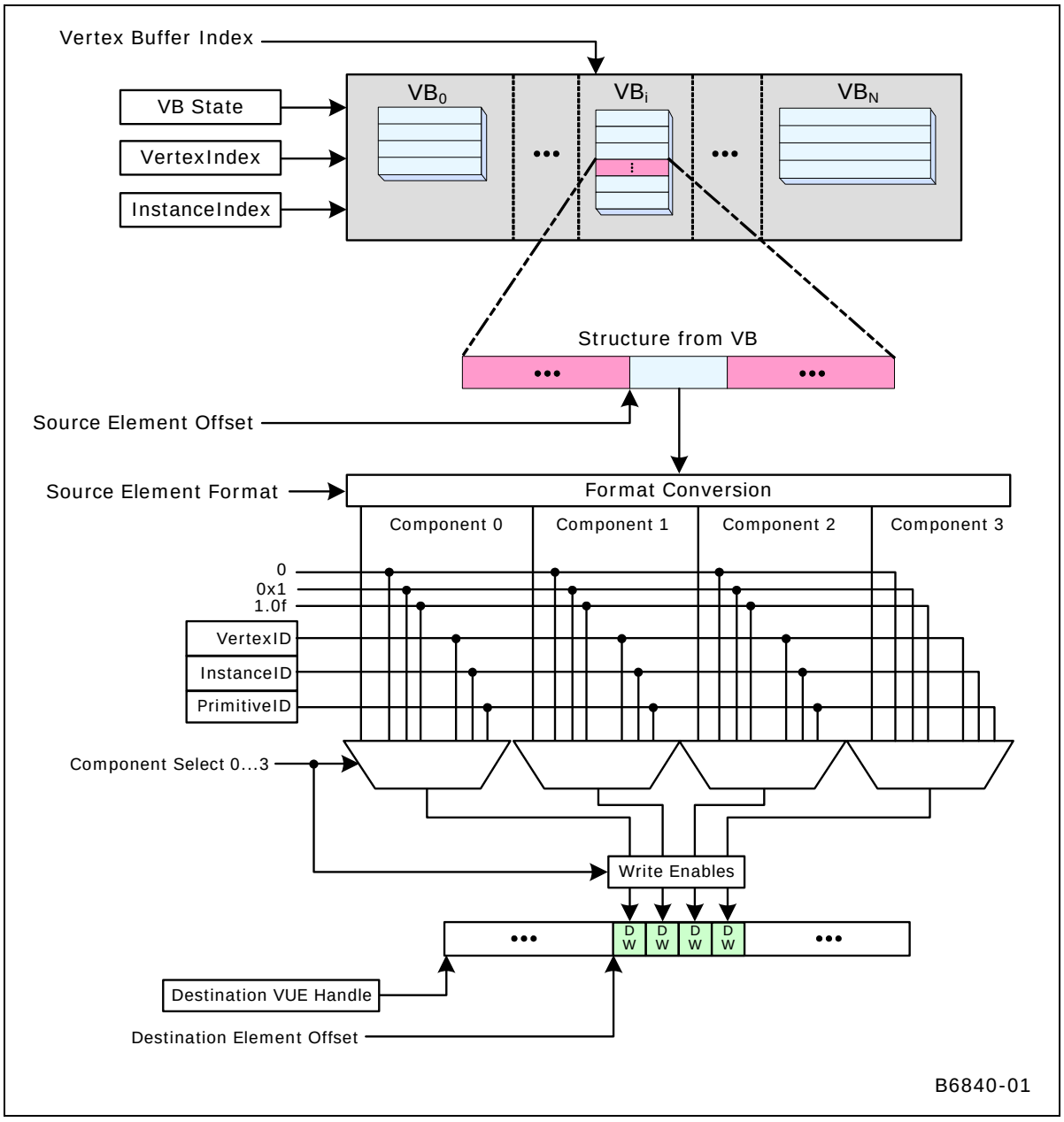
Value Name	Description	Project
0 VFCOMP_NOSTORE	Don't store this component. (Not valid for Component 0, but can be used for Component 1-3). Once this setting is used for a component, all higher-numbered components (if any) MUST also use this setting. (I.e., no holes within any particular vertex element). Also, there are no 'holes' allowed in the destination vertex: NOSTORE components must be overwritten by subsequent components unless they are the trailing DWords of the vertex. Software must explicitly chose some value (probably 0) to be written into DWords that would otherwise be 'holes'.	All
1 VFCOMP_STORE_SRC	Store corresponding component from format-converted source element. Storing a component that is not included in the Source Element Format results in an UNPREDICTABLE value being stored. Software should used the STORE_0 or STORE_1 encoding to supply default components.	All
2 VFCOMP_STORE_SRC	Store 0 (interpreted as 0.0f if accessed as a float value)	
3 VFCOMP_STORE_1_FP	Store 1.0f	
4 VFCOMP_STORE_1_INT	Store 0x1	
5 VFCOMP_STORE_VID	Store Vertex ID (as U32)	
6 VFCOMP_STORE_IID	Store Instance ID (as U32)	

	27	**Reserved** Project: All Format: MBZ
	26:24	**Component 1 Control**
	23	**Reserved** Project: All Format: MBZ

3DSTATE_InstructionName		
	22:20	**Component 2 Control**
	19	**Reserved** Project: All Format: MBZ
	18:16	**Component 3 Control**
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**Destination Element Offset** Project: DevILK This field specifies a DWord offset into the target URB Entry into which the converted vertex element is to be written. **Programming Notes** Element Alignment: Elements must be size-aligned within the URB entry. E.g., it is valid to pack two 2-component elements within one 128-bit region, but it is not valid to have a 4-component element to span two 128-bit regions. This restriction is imposed as a result of limitations on how logical elements can be accessed from GRF registers by the GEN4 EUs (it is not an restriction imposed by the VF unit). The data written to the URB will be padded (with 0) to a 256-bit boundary .
	31:0	**Reserved** Project: [DevILK] Format: MBZ

3.5.4 Vertex Element Data Path

The following diagram shows the path by which a vertex element within the destination VUE is generated and how the fields of the VERTEX_ELEMENT_STATE structure is used to control the generation.



3.6 3D Primitive Processing

3.6.1 3DPRIMITIVE Command

3DPRIMITIVE

Project:	All	Length Bias:	2

The 3DPRIMITIVE command is used to submit 3D primitives to be processed by the 3D pipeline. Typically the processing results in the rendering of pixel data into the render targets, but this result is not required.

The parameters passed in this command are forwarded to the Vertex Fetch function. The Vertex Fetch function will use this information to generate vertex data structures and store them in the URB. These vertices are then passed down the 3D pipeline for possible processing by the Vertex Shader, Geometry Shader and Clipper. If rendering is required, the computed vertices are passed down to the StripFan and WindowerMasker units.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 3h 3DPRIMITIVE Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 0h 3DPRIMITIVE Format: OpCode
	15	**Vertex Access Type** Project: All Format: VertexAccessType This field specifies how data held in vertex buffers marked as VERTEXDATA is accessed by Vertex Fetch.

Value Name	Description	Project
0h SEQUENTIAL	VERTEXDATA buffers are accessed sequentially	All
1h RANDOM	VERTEXDATA buffers are accessed randomly via an index obtained from the Index Buffer.	All

	14:10	**Primitive Topology Type** Project: All Format: 3D_PrimTopoType	See table below for encoding, see *3D Overview* for diagrams and general comments This field specifies the *topology type* of 3D primitive generated by this command. Note that a single primitive topology (list/strip/fan/etc.) can contain a number of basic objects (lines, triangles, etc.).

3DPRIMITIVE		
	9	**Reserved** Project: DevBW, DevCL Format: MBZ
	9	**Indirect Vertex Count** Project: DevCTG, DevILK Format: U1 If set, the Vertex Count Per Instance field contains the graphics memory address of the DWord containing the vertex count. If clear, the Vertex Count Per Instance field contains the count as immediate data.
	8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 4h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:0	**Vertex Count Per Instance** Project: DevBW, DevCL Format: U32 Count of Vertices Address: GraphicsAddress[31:0] Range: [0, 2^32-1] upper limit probably constrained by VB size This field specifies how many vertices are to be generated for each instance of the primitive topology. **[DevCTG]**: If **Indirect Vertex Count** is set: Format = DWord-aligned Graphics Memory Address of the count value (there the count value has the same Format/Range as listed below) If **Indirect Vertex Count** is clear: Format = U32 count of vertices Range = [0, 2^32-1] (upper limit probably constrained by VB size) **[DevILK+]:** Ignored if **Internal Vertex Count** is set. **Programming Notes** • This per-instance value should specify a valid number of vertices for the primitive topology type. E.g., for 3DPRIM_TRILIST_ADJ, this field should specify a multiple of 6 vertices. However, in cases where too few or too many vertices are provided, the unused vertices will be silently discarded by the pipeline. • A 0 value is this field effectively makes the command a 'no-operation'.

3DPRIMITIVE

	31:0	**Vertex Count Per Instance** Project: DevCTG, ,DevILK Format: U32 Count of Vertices Range: [0, 2^32-1] upper limit probably constrained by VB size Address: GraphicsAddress[31:0] Surface Type: U32*1 This field specifies how many vertices are to be generated for each instance of the primitive topology. If **Indirect Vertex Count** is set: Format = DWord-aligned Graphics Memory Address of the count value (there the count value has the same Format/Range as listed below) If **Indirect Vertex Count** is clear: Format = U32 count of vertices **Programming Notes** • This per-instance value should specify a valid number of vertices for the primitive topology type. E.g., for 3DPRIM_TRILIST_ADJ, this field should specify a multiple of 6 vertices. However, in cases where too few or too many vertices are provided, the unused vertices will be silently discarded by the pipeline. • A 0 value is this field effectively makes the command a 'no-operation'.
2	31:0	**Start Vertex Location** Project: All Format: U32 structure index This field specifies the "starting vertex" for each instance. This allows skipping over part of the vertices in a buffer if, for example, a previous 3DPRIMITIVE command had already drawn the primitives associated with the earlier entries. For SEQUENTIAL access, this field specifies, for each instance, a starting structure index into the vertex buffers For RANDOM access, this field specifies, for each instance, a starting index into the Index Buffer. **Programming Notes** Access of any data outside of the valid extent of a vertex or index buffer will return the value 0 (i.e., appears as if the data stored at the invalid location was 0).
3	31:0	**Instance Count** Project: All Format: U32 count of instances Range 1..2^32-1 This field specifies the number of instances by which the primitive topology is to be regenerated. A value of 0 is UNDEFINED. A value of 1 effectively specifies "non-instanced" operation, though vertex buffers will still be used to provide instance data, if so programmed.

3DPRIMITIVE		
4	31:0	**Start Instance Location** Project: All Format: U32 structure index This field specifies the “starting instance” for the command as an initial structure index into INSTANCEDATA buffers. Subsequent instances will access sequential instance data structures, as controlled by the **Instance Data Step Rate**. **Programming Notes** Access of any data outside of the valid extent of a vertex or index buffer will return the value 0 (i.e., appears as if the data stored at the invalid location was 0).
5	31:0	**Base Vertex Location** Project: All Format: S31 structure index bias This field specifies a signed bias to be added to values read from the index buffer. This allows the same index buffer values to access different vertex data for different commands. This field applies only to RANDOM access mode. This field is ignored for SEQUENTIAL access mode, where there Start Vertex Location can be used to specify different regions in the vertex buffers. **Programming Notes** Access of any data outside of the valid extent of a vertex or index buffer will return the value 0 (i.e., appears as if the data stored at the invalid location was 0).

3.6.2 Functional Overview

The following pseudocode summarizes the general flow of 3D Primitive Processing.

```
CommandInit
InstanceLoop {
      VertexLoop {
            VertexIndexGeneration
            if (CutFlag)
                  TerminatePrimitive
            else
                  OutputBufferedVertex
                  VertexCacheLookup
                  if (miss) {
                        VertexElementLoop {
                              SourceElementFetch
                              FormatConversion
                              DestinationComponentSelection
                              PrimitiveInfoGeneration
                              URBWrite
                        }
                  }
      }
      TerminatePrimitive
}
```

3.6.3 CommandInit

The `InstanceID` value is initialized to 0.

3.6.4 InstanceLoop

The InstanceLoop is the outmost loop, iterating through each instance of primitives. There is no special “non-instanced” mode – at a minimum there is one instance of primitives.

For SEQUENTIAL accessing, the `VertexID` value is initialized to 0 at the start of each instance. (For RANDOM accessing, there is no initial value for `VertexID`, as it is derived from the fetched IB value).

The `PrimitiveID` is also initialized to 0 at the start of each instance. `StartPrim` is initialized to TRUE.

The VertexLoop (see below) is then executed to iterate through the instance vertices and output vertices to the pipeline as required.

The end of each iteration of InstanceLoop includes an implied “cut” operation.

The `InstanceID` value is incremented at the end of each InstanceLoop. Note that each instance will produce the same vertex outputs with the exception of any data dependent on `InstanceID` (i.e., "instance data").

3.6.5 VertexLoop

The VertexLoop iterates `VertexNumber` through the `VertexCountPerInstance` vertices for the instance.

For each iteration, a number of processing steps are performed (see below) to generate the information that comprises a vertex. Note that, due to CutProcessing, each iteration does not necessarily output a vertex to the pipeline. When a vertex is to be output, the following information is generated for that vertex:

- `PrimitiveType` associated with the vertex. This is simply a copy of the PrimitiveTopologyType field of the 3DPRIMITIVE
- VUE handle at which the vertex data is stored
  - For a Vertex Cache hit, the VUE handle is marked with a `VCHit` boolean.
  - Otherwise, the VertexLoop will generate and store the input vertex data into the VUE referenced by this handle.
- The `PrimitiveID` associated with the vertex. See PrimitiveInfoGeneration.
- `PrimStart` and `PrimEnd` booleans associated with the vertex. See PrimitiveInfoGeneration.

(Note that a single vertex of buffering is required in order to associate `PrimEnd` with a vertex, as this information may not be known until the next iteration through the VertexLoop (see *OutputPrimitiveDelimiter*).

`VertexNumber` value is incremented by 1 at the end of the loop.

3.6.6 VertexIndexGeneration

A `VertexIndex` value needs to be derived for each vertex. With the exception of the "cut" index, this index value is used as the vertex cache tag and will be used as a structure index into all VERTEXDATA VBs.

For SEQUENTIAL accessing, the `VertexID` and `VertexIndex` value is derived as shown below:

```
VertexIndex = StartVertexLocation + VertexNumber

VertexID = VertexNumber
```

For RANDOM access, the `VertexID` and `VertexIndex` is derived from an `IBValue` read from the IB, as shown below:

```
IBIndex = StartVertexLocation + VertexNumber
VertexID = IB[IBIndex]
if (VertexID == ‘all ones’)
      CutFlag = 1
else
      VertexIndex = VertexID + BaseVertexLocation
      CutFlag = 0
endif
```

3.6.7 TerminatePrimitive

For RANDOM accessing, and when enabled via **Cut Index Enable**, a fetched `IBValue` of ‘all ones’ (0xFF, 0xFFFF, or 0xFFFFFFFF depending on **Index Format**) is interpreted as a ‘cut value’ and signals the termination of the current primitive and the possible start of the next primitive. This allows the application to specify an instance as a sequence of variable-sized strip primitives (though the cut value applies to any primitive type).

Also, there is an implied primitive termination at the end of each InstanceLoop (and so strip primitives cannot span multiple instances).

In either case, the currently-buffered vertex (if any) is marked with `EndPrim` and then flushed out to the pipeline.

The next-output vertex (if any) will be marked with `StartPrim`.

Whenever a primitive delimiter is encountered, the `PIDCounterS` and `PIDCounterR` counters are reset to 0. These counters control the incrementing (in PrimitiveInfoGeneration, below) of `PrimitiveID` within each primitive topology of an instance.

```
if (PIDCounterS != 0) // There is a buffered vertex
      if (primType == TRISTRIP_ADJ)
            if (PIDCounterS==6 || PIDCounterR==1)
                  PrimitiveID++
            endif
      endif
      PrimEnd = TRUE
      OutputBufferedVertex
endif
PrimEnd = FALSE
PrimStart = TRUE
```

3.6.8 VertexCacheLookup

The `VertexIndex` value is used as the tag value for the VertexCache (see *Vertex Cache*, above). If the Vertex Cache is enabled and the `VertexIndex` value hits in the cache, the VUE handle is read from the cache and inserted into the vertex stream. It is marked with a `VCHit`.

Otherwise, for Vertex Cache misses, a VUE handle is obtained to provide storage for the generated vertex data. VertexLoop processing then proceeds to iterate through the VEs to generate the destination VUE data.

3.6.9 VertexElementLoop

The VertexElementLoop generates and stores vertex data in the destination VUE one VE at a time.

[Pre-DevILK] Note that VEs must be defined (via 3DSTATE_VERTEX_ELEMENTS) in order of increasing **Destination Element Offset**, though architecturally the order by which VEs are processed is arbitrary (has no impact on the results).

3.6.10 SourceElementFetch

The following assumes the VE requires data from a VB, which is the typical case. In the case that the VE is completely comprised of constant and/or auto-generated IDs, the SourceElementFetch and FormatConversion steps are skipped.

The structure index within the VE's selected VB is computed as follows:

```
if (VB is a VERTEXDATA VB)
      VBIndex = VertexIndex
else // INSTANCEDATA VB
      VBIndex = StartInstanceLocation
      if (VB.InstanceDataStepRate > 0)
            VBIndex += InstanceID/VB.InstanceDataStepRate
endif
```

If `VBIndex` is invalid (i.e., negative or past **Max Index**), the data returned from the VB fetch is defined to be zero. Otherwise, the address of the source data required for the VE is then computed and the data is read from the VB. The amount of data read from the VB is determined by the **Source Element Format**.

```
if ( (VBIndex<0) || (VBIndex>VB.MaxIndex) )
      srcData = 0
else
      pSrcData = VB.BufferStartingAddress + (VBIndex *
VB.BufferPitch) + VE.SourceElementOffset
      srcData = MemoryRead( pSrcData, VE.SourceElementFormat )
endif
```

3.6.11 FormatConversion

Once the VE source data has been fetched, it is subjected to format conversion. The output of format conversion is up to 4 32-bit components, each either integer or floating-point (as specified by the **Source Element Format**). See *Sampler* for conversion algorithms.

The following table lists the valid **Source Element Format** selections, along with the format and availability of the converted components (if a component is listed as "-", it cannot be used as source of a VUE component). Note: This table is a subset of the list of supported surface formats defined in the *Sampler* chapter. Please refer to that table as the "master list". This table is here only to identify the components available (per format) and their format.

Table 3-1. Source Element Formats supported in VF Unit

Source Element Format	Converted Component				
	Format	0	1	2	3
256 bits					
R64G64B64A64_FLOAT	FLOAT	R	G	B	A
192 bits					
R64G64B64_FLOAT	FLOAT	R	G	B	A
128 bits					
R32G32B32A32_FLOAT	FLOAT	R	G	B	A
R32G32B32A32_SNORM	FLOAT	R	G	B	A
R32G32B32A32_UNORM	FLOAT	R	G	B	A
R32G32B32A32_SINT	SINT	R	G	B	A
R32G32B32A32_UINT	UINT	R	G	B	A
R32G32B32A32_SSCALED	FLOAT	R	G	B	A
R32G32B32A32_USCALED	FLOAT	R	G	B	A
R64G64_FLOAT	FLOAT	R	G	-	-
96 bits					
R32G32B32_FLOAT	FLOAT	R	G	B	-
R32G32B32_SNORM	FLOAT	R	G	B	-
R32G32B32_UNORM	FLOAT	R	G	B	-
R32G32B32_SINT	SINT	R	G	B	-
R32G32B32_UINT	UINT	R	G	B	-
R32G32B32_SSCALED	FLOAT	R	G	B	-
R32G32B32_USCALED	FLOAT	R	G	B	-
64 bits					
R16G16B16A16_FLOAT	FLOAT	R	G	B	A
R16G16B16A16_SNORM	FLOAT	R	G	B	A
R16G16B16A16_UNORM	FLOAT	R	G	B	A
R16G16B16A16_SINT	SINT	R	G	B	A

Source Element Format	Converted Component					
	Format	0	1	2	3	
R16G16B16A16_UINT	UINT	R	G	B	A	
R16G16B16A16_SSCALED	FLOAT	R	G	B	A	
R16G16B16A16_USCALED	FLOAT	R	G	B	A	
R32G32_FLOAT	FLOAT	R	G	-	-	
R32G32_SNORM	FLOAT	R	G	-	-	
R32G32_UNORM	FLOAT	R	G	-	-	
R32G32_SINT	SINT	R	G	-	-	
R32G32_UINT	UINT	R	G	-	-	
R32G32_SSCALED	FLOAT	R	G	-	-	
R32G32_USCALED	FLOAT	R	G	-	-	
R64_FLOAT	FLOAT	R	-	-	-	
48 bits						
R16G16B16_SNORM	FLOAT	R	G	B	-	
R16G16B16_UNORM	FLOAT	R	G	B	-	
R16G16B16_SSCALED	FLOAT	R	G	B	-	
R16G16B16_USCALED	FLOAT	R	G	B	-	
32 bits						
R10G10B10A2_UNORM	FLOAT	R	G	B	A	
R10G10B10A2_UINT	UINT	R	G	B	A	
R10G10B10X2_USCALED	FLOAT	R	G	B	-	
R10G10B10_SNORM_A2_UNORM	FLOAT	R	G	B	A	
B8G8R8A8_UNORM	FLOAT	B	G	R	A	
R8G8B8A8_SNORM	FLOAT	R	G	B	A	
R8G8B8A8_UNORM	FLOAT	R	G	B	A	
R8G8B8A8_SINT	SINT	R	G	B	A	
R8G8B8A8_UINT	UINT	R	G	B	A	
R8G8B8A8_SSCALED	FLOAT	R	G	B	A	
R8G8B8A8_USCALED	FLOAT	R	G	B	A	
R11G11B10_FLOAT	FLOAT	R	G	B	-	
R16G16_FLOAT	FLOAT	R	G	-	-	
R16G16_SNORM	FLOAT	R	G	-	-	
R16G16_UNORM	FLOAT	R	G	-	-	
R16G16_SINT	SINT	R	G	-	-	
R16G16_UINT	UINT	R	G	-	-	
R16G16_SSCALED	FLOAT	R	G	-	-	
R16G16_USCALED	FLOAT	R	G	-	-	
R32_FLOAT	FLOAT	R	-	-	-	

Source Element Format	Converted Component					
	Format	0	1	2	3	
R32_SINT	SINT	R	-	-	-	
R32_UINT	UINT	R	-	-	-	
R32_SSCALED	FLOAT	R	-	-	-	
R32_USCALED	FLOAT	R	-	-	-	
R32_SNORM	FLOAT	R	-	-	-	
R32_UNORM	FLOAT	R	-	-	-	
24 bits						
R8G8B8_SNORM	FLOAT	R	G	B	-	
R8G8B8_UNORM	FLOAT	R	G	B	-	
R8G8B8_SSCALED	FLOAT	R	G	B	-	
R8G8B8_USCALED	FLOAT	R	G	B	-	
16 bits						
R8G8_SNORM	FLOAT	R	G	-	-	
R8G8_UNORM	FLOAT	R	G	-	-	
R8G8_SINT	SINT	R	G	-	-	
R8G8_UINT	UINT	R	G	-	-	
R8G8_SSCALED	FLOAT	R	G	-	-	
R8G8_USCALED	FLOAT	R	G	-	-	
R16_FLOAT	FLOAT	R	-	-	-	
R16_SNORM	FLOAT	R	-	-	-	
R16_UNORM	FLOAT	R	-	-	-	
R16_SINT	SINT	R	-	-	-	
R16_UINT	UINT	R	-	-	-	
R16_SSCALED	FLOAT	R	-	-	-	
R16_USCALED	FLOAT	R	-	-	-	
8 bits						
R8_SNORM	FLOAT	R	-	-	-	
R8_UNORM	FLOAT	R	-	-	-	
R8_SINT	SINT	R	-	-	-	
R8_UINT	UINT	R	-	-	-	
R8_SSCALED	FLOAT	R	-	-	-	
R8_USCALED	FLOAT	R	-	-	-	

3.6.12 DestinationFormatSelectio n

The **Component Select 0..3** bits are then used to select, on a per-component basis, which destination components will be written and with which value. The supported selections are the converted source component, `VertexID`, `InstanceID`, `PrimitiveID`, the constants 0 or 1.0f, or nothing (VFCOMP_NO_STORE). If a converted component is listed as '-' (not available) in Table 3-1, it must not be selected (via VFCOMP_STORE_SRC), or an UNPREDICTABLE value will be stored in the destination component.

The selection process sequences from component 0 to 3. Once a **Component Select** of VFCOMP_NO_STORE is encountered, all higher-numbered **Component Select** settings must also be programmed as VFCOMP_NO_STORE. It is therefore not permitted to have 'holes' in the destination VE.

3.6.13 PrimitiveInfoGeneration

A `PrimitiveID` value and `PrimStart` boolean need to be associated with the vertex.

If the vertex is either the first vertex of an instance or the first vertex following a 'cut index', the vertex is marked with `PrimStart`.

`PrimitiveID` gets incremented such that subsequent per-object processing (i.e., in the GS or SF/WM) will see an incrementing value associated with each sequential object within an instance. The `PrimitiveID` associated with the provoking, non-adjacent vertex of an object is applied to the object.

The following pseudocode describe the logic used in the VertexLoop to compute the `PrimitiveID` value associated with the vertex. Recall that `PrimitiveID` is reset to 0 at the start of each InstanceLoop.

```
if (PIDCounterS < S[primType])
      PIDCounterS++
else
      if (PIDCounterR < R[primType])
            PIDCounterR++
      else
            PrimitiveID++
            PIDCounterR = 0
      endif
endif
```

Two counters are employed to control the incrementing of `PrimitiveID`. The counters are compared against two corresponding parameters associated with the primitive topology type.

The `PIDCounterS` is used to 'skip over' some number (possibly zero) initial vertices of the primitive topology. This counter gets reset to 0 after each primitive is terminated.

Then the `PIDCounterR` is used to periodically increment the `PrimitiveID`, where the incrementing interval (vertex count) is topology-specific.

The following table lists the `S[]` and `R[]` values associated with each primitive topology type.

PrimTopologyType	S, R	PrimitiveID Outputs
POINTLIST POINTLIST_BF	1, 0	**0,1,2,3, …**
LINELIST	1, 1	**0,0,1,1,2,2,3,3, …**
LINELIST_ADJ	1, 3	**0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3 …**
LINESTRIP LINESTRIP_BF LINESTRIP_CONT	2, 0	**0,0,1,2,3, …**
LINESTRIP_ADJ	3, 0	**0,0,0,1,2,3,…**
LINELOOP	2, 0	0,0,1,2,3,… Note: this breaks the usage model (as the initial vertex is the provoking vertex for the closing line, but it has an invalid PrimitiveID of 0), but is effectively a don't care as PrimitiveID is only required for D3D and LINELOOP is an OpenGL-only primitive.) The LINELOOP topology is converted to LINESTRIP topology at the beginning of the 3D pipeline.
TRILIST RECTLIST	1, 2	0,0,0,1,1,1,2,2,2,3,3,3,…
TRILIST_ADJ	1, 5	0,0,0,0,0,0,1,1,1,1,1,1,2,2,2,2,2,2,…
TRISTRIP TRISTRIP_REV	3, 0	0,0,0,1,2,3, …
TRISTRIP_ADJ	5, 1	0,0,0,0,0,0,1,1,2,2,3,3, …
TRIFAN TRIFAN_NOSTIPPLE POLYGON	3, 0	0,0,0,1,2,3, …
QUADLIST	1, 3	0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3, … Note: The QUADLIST topology is converted to POLYGON topology at the beginning of the 3D pipeline.
QUADSTRIP	3, 1	0,0,0,0,1,1,2,2,3,3, … Note: The QUADSTRIP topology is converted to POLYGON topology at the beginning of the 3D pipeline.

3.6.14 URBWrite

The selected destination components are written into the destination VUE starting at **Destination Offset Select**. See the description of 3DPRIMITIVE for restrictions on this field.

3.6.15 OutputBufferedVertex

In order to accommodate 'cut' processing, the VF unit buffers one output vertex. The generation of a new vertex or the termination of a primitive causes the buffered vertex to be output to the pipeline.

3.7 Dangling Vertex Removal

The last functional stage of processing of the 3DPRIMITIVE command is the removal of “dangling” vertices. This includes the discarding of primitive topologies without enough vertices for a single object (e.g., a TRISTRIP with only two vertices), as well as the discarding of trailing vertices that do not form a complete primitive (e.g., the last two vertices of a 5-vertex TRILIST).

This function is best described as a filter operating on the vertex stream emitted from the processing of the 3DPRIMITIVE. The filter inputs the PrimType, PrimStart and PrimEnd values associated with the generated vertices. The filter only outputs primitive topologies without dangling vertices. This requires the filter to (a) be able to buffer some number of vertices, and (b) be able to remove dangling vertices from the pipeline and dereference the associated VUE handles.

3.8 Other Vertex Fetch Functionality

3.8.1 Statistics Gathering

The VF stage tracks two pipeline statistics, the number of vertices fetched and the number of objects generated. VF will increment the appropriate counter for each when statistics gathering is enabled by issuing the 3DSTATE_VF_STATISTICS command with the **Statistics Enable** bit set.

DWord	Bit	Description
0	31:29	**Command Type =** GFXPIPE = 03h
	28:16	**GFXPIPE Opcode =** 3DSTATE_VF_STATISTICS **[DevBW], [DevCL]** GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 0Bh] (Pipelined) **[DevCTG+]** GFXPIPE[28:27 = 1h, 26:24 = 0h, 23:16 = 0Bh] (Pipelined, Single DW)
	15:1	Reserved : MBZ
	0	**Statistics Enable** If ENABLED, VF will increment the pipeline statistics counters IA_VERTICES_COUNT and IA_PRIMITIVES_COUNT for each vertex fetched and each object output, respectively, for 3DPRIMITIVE commands issued subsequently. If DISABLED, these counters will not be incremented for subsequent 3DPRIMITIVE commands. Format: Enable

3.8.1.1 Vertices Generated

VF will increment the IA_VERTICES_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) for each vertex it fetches, even if that vertex comes from a cache rather than directly from a vertex buffer in memory. Any “dangling” vertices (fetched vertices that are part of an incomplete object) should be included.

3.8.1.2 Objects Generated

VF will increment the IA_PRIMITIVES_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) for each object (point, line, triangle or quadrilateral) that it forwards down the pipeline. NOTE: For LINELOOP, the last (closing) line object is not counted.

3.9 HS Thread Execution

Input to HS threads is comprised of:

- Input Control Points (incoming patch vertices), pushed into the payload and/or passed indirectly via URB handles.
- Push Constants (common to all threads)
- Patch Data handle
- resources available via binding table entries (accessed through shared functions)
- miscellaneous payload fields (Instance Number, etc.)

Typically the only output of the HS threads is the Path URB Entry (patch record). All threads are passed the same patch record handle. As the (possibly concurrent) threads can both read and write the patch record, it is up to the kernels to ensure deterministic results. One approach would be to use the thread’s Instance Number as an index for URB write destinations.

3.9.1 Dispatch Mask

HS threads will be dispatched with the dispatch mask set to 0xFFFF. It is the responsibility of the kernel to modify the execution mask as required (e.g., if operating in SIMD4x2 mode but only the lower half is active, as would happen in one thread is the threads were computing an odd number of OCPs via SIMD4x2 operation).

3.10 ICP Dereferencing

If ICPs are only pushed in HS payloads (i.e., the **Include Vertex Handles** state bit is clear), the ICP handles will automatically be released after the last instance for the patch is dispatched.

If **Include Vertex Handles** is set (the HS thread(s) will be reading ICP data in from the URB, it is the responsibility of the HS thread instances to explicitly dereference all the ICP handles via use of the **Complete** bit in URB_READ_xxx commands.

- If only one instance is used, that instance can dereference the ICP handles as soon as they are no longer needed, by setting **Complete** in the last URB_READ from that handle. Otherwise all (or the remaining) ICP handles need to be explicitly dereferenced via (possibly null-response-length) URB_READ commands prior to thread EOT.

- If more than one instance is spawned, the last-terminating instance is responsible for dereferencing all the ICP handles before it terminates. Instances can detect that they are the last-terminating thread via use of the semaphore allocated to the patch (via the **Semaphore Handle** and **Semaphore Index** payload fields). A URB_ATOMIC_INC operation (URB_ATOMIC command) can be performed on this semaphore by each instance prior to terminating. Only the last-terminating thread will observe the value (InstanceCount – 1) as a return value. After dereferencing all the ICPs, the last-terminating thread must also reset the semaphore to 0 via the URB_ATOMIC_MOV operation.

3.11 Patch URB Entry (Patch Record) Output

For each patch, the HS thread(s) generate a single patch record, starting with a fixed 32B Patch Header . When the final thread instance terminates, the patch record handle is passed down the pipeline to the Tessellation Engine (TE).

3.11.1 Patch Header

The first 8 DWords of the patch record is defined as a “Patch Header”. The Patch Header is written by an HS thread and read by the TE stage. It normally contains up to six **Tessellation Factors** (TFs) that determine how finely the TE stage needs to tessellate a domain (if at all). In SW Tessellation mode, the header contains **Domain Point Count** and **Domain Point Buffer Starting Address** fields which identify the domain points generated by an HS thread. The following diagram shows the fixed layouts of the Patch Header, depending on DomainType and SW Tessellation Mode.

Table 3-2 Patch Header (QUAD Domain)

DWord	Bits	Description
7	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32
6	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
5	31:0	**UEQ1 Tessellation Factor** Format: FLOAT32
4	31:0	**VEQ1 Tessellation Factor** Format: FLOAT32

DWord	Bits	Description
3	31:0	**Inside U Tessellation Factor** Format: FLOAT32
2	31:0	**Inside V Tessellation Factor** Format: FLOAT32
1-0	31:0	Reserved : MBZ

Table 3-3 Patch Header (TRI Domain)

DWord	Bits	Description
7	31:0	**UEQ0 Tessellation Factor** Format: FLOAT32
6	31:0	**VEQ0 Tessellation Factor** Format: FLOAT32
5	31:0	**WEQ0 Tessellation Factor** Format: FLOAT32
4	31:0	**Inside Tessellation Factor** Format: FLOAT32
3-0	31:0	Reserved : MBZ

Table 3-4 Patch Header (ISOLINE Domain)

DWord	Bits	Description
7	31:0	**Line Detail Tessellation Factor** Format: FLOAT32
6	31:0	**Line Density Tessellation Factor** Format: FLOAT32
5-0	31:0	Reserved : MBZ

Table 3-5 Patch Header (SW Tessellation Mode)

DWord	Bits	Description
7	31:0	**Domain Point Count** Specifies the number of DOMAIN_POINT structures in the domain point list in memory. If 0, there are no domain points defined, the patch will considered "culled", and the TE stage will discard the patch. Otherwise the TS stage will send this number of domain points down the pipeline. Format: U32
6	31:6	**Domain Point Buffer Starting Address (DPBSA)** This field specifies the starting memory offset from SW Tessellation Base Address (set by the SWTESS_BASE_ADDRESS command) at which the HS thread has written a list of DOMAIN_POINT structures. This field is ignored if **Domain Point Count** is 0. Format: 64B-aligned offset from SW Tessellation Base Address
	5:0	Reserved : MBZ
5-0	31:0	Reserved: MBZ

3.11.2 DOMAIN_POINT Structure

In SW Tessellation Mode (i.e., when the TE State is SW_TESS), the TE stage will read a sequence of DOMAIN_POINT structures from memory, starting at the Domain Point Buffer Starting Address field of the patch header. (The DPBSA is treated as an offset from the SW Tessellation Base Address as set by the SWTESS_BASE_ADDRESS command).

Table 3-6. DOMAIN_POINT Memory Structure (SW Tessellation)

DWord	Bit	Description
0	31	**PrimStart** Set on the first domain point of the topology (e.g., first vertex in a TRISTRIP).
	30	**PrimEnd** Set on the last domain point of the topology (e.g., last vertex in a TRISTRIP). Programming note: Software must ensure that incomplete primitives are not output, or behavior is UNDEFINED.
	29	**PatchEnd** Set on the last domain point for the patch. By definition, PrimEnd must also be set. Programming Note: Software must ensure that the **Domain Point Count** coincides with the domain point marked with PatchEnd.
	28:24	**PrimType** This is the primitive topology type. Format: See 3DPRIMITIVE for encodings Valid values: POINTLIST, LINESTRIP, LINELIST, TRISTRIP, TRISTRIP_REV, TRILIST, TRIFAN.

DWord	Bit	Description
	23:19	Reserved
	18:17	**DS Tag [16:15]** This field provides bits [16:15] of the DS Tag value for this domain point. See **DS Tag [14:0]**. Format: U2
	16:0	**U Coordinate** Format: U1.16
1	31:17	**DS Tag [14:0]** This field provides bits [14:0] of the DS Tag value for this domain point. In order to utilize the DS cache, the 17-bit DS Tag must be unique for the associated U,V coordinate. If software cannot guarantee this, the DS cache must be disabled when in SW Tessellation mode. Format: U15
	16:0	**V Coordinate** Format: U1.16

3.12 Statistics Gathering

3.12.1 HS Invocations

The HS unit controls the HS_INVOCATIONS counter, which counts the number of patches processed by the HS stage.

4. Vertex Shader (VS) Stage

4.1 VS Stage Overview

The VS stage of the GEN4 3D Pipeline is used to perform processing ("shading") of vertices after being assembled and written to the URB by the VF function. The primary function of the VS stage is to pass vertices that miss in the Vertex Cache to VS threads, and then pass the VS thread-generated vertices down the pipeline. Vertices that hit in the Vertex Cache are passed down the pipeline unmodified.

When the VS stage is disabled, vertices flow through the unit unmodified (i.e., as written by the VF unit).

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Overview* chapter for a general description of a 3D pipeline stage, as much of the VS stage operation and control falls under these "common" functions. I.e., most stage state variables and VS thread payload parameters are described in *3D Overview*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the VS stage, and any exceptions the VS stage exhibits with respect to common FF unit functions.

4.1.1 Vertex Caching

The 3D Pipeline employs a Vertex Cache that is shared between the VF and VS units. (See *Vertex Fetch* chapter for additional information). The Vertex Cache may be explicitly DISABLED via the **Vertex Cache Disable** bit in VS_STATE. Even when explicitly ENABLED, the VS unit can (by default) implicitly disable and invalidate the Vertex Cache when it detects one of the following conditions:

1. Either VertexID or PrimitiveID is selected as part of the vertex data stored in the URB.
2. Sequential indices are used in the 3DPRIMITIVE command (though this is effectively a don't care as there wouldn't be any hits anyway).

The implicit disable will persist as long as one of these conditions persist. The **Vertex Cache Implicit Disable Inhibit** bit in the VFSKPD MI register is provided (as a "chicken bit") to inhibit the VS unit's implicit cache disable. If inhibited, software is responsible for explicitly enabling/disabling the vertex cache as required for correct operation.

Note: Even though use of VertexID causes an implicit cache disable, there is no known (good) reason why this is required. Software can therefore either (a) allow the implicit cache disable (the default action) and live with some possible performance penalty due to the too-often-disabled cache, or (b) use the chicken bit to allow the cache to remain enabled if VertexID is used. However, (b) will require software to explicitly disable the cache if PrimitiveID is used.

The Vertex Cache is implicitly invalidated between 3DPRIMITIVE commands and between instances within a 3DPRIMITIVE command – therefore use of InstanceID in a Vertex Element is not a condition underwhich the cache is implicitly disabled.

The following table summarizes the modes of operation of the Vertex Cache:

Vertex Cache	VS Function Enable	Mode of Operation
DISABLED (implicitly or explicitly)	***DISABLED***	***Vertex Cache is not used. VF unit will assemble all vertices and write them into the URB entry supplied by the VS unit. VS unit will pass references to these VUEs down the pipeline unmodified.*** ***Usage Model: This is an exceptional condition, only required when the VF-generated vertices contain InstanceID or PrimitiveID and more than one instance is produced. Otherwise the Vertex Cache should be enabled.***
DISABLED ***(implicitly or explicitly)***	***ENABLED***	***Vertex Cache is not used. VF unit will assemble all vertices and write them into the URB entry supplied by the VS unit. VS unit will spawn VS threads to process all vertices, overwriting the input data with the results. The VS unit pass references to these VUEs down the pipeline.*** ***Usage Model: This mode is only used when the VS function is required, but either (a) the input vertex contains InstanceID or PrimitiveID and more than one instance is generated or (b) the VS kernel produces a side effect (e.g., writes to a memory buffer) which requires every vertex to be processed by a VS thread.***
ENABLED	***DISABLED***	***Vertex Cache is used to provide reuse of VF-generated vertices. The VF unit will check the cache and only process (assemble/write) vertices that miss in the cache. In either case, the VS unit will pass references to vertices (that hit or miss) down the pipeline without spawning any VS threads.*** ***Usage Model: Normal operation when the VS function is not required. Note that there may be situations which require the VS function to be used even when not explicitly required by the API. E.g., perspective divide may be required for clip testing.***
ENABLED	***ENABLED***	***Vertex Cache is used to provide reuse of VS-processed vertices. The VF unit will check the cache and only process (assemble/write) vertices that miss in the cache. The VS unit will only process (shade) the vertices that missed in the cache. The VS unit sends references to hit or missed vertices down the pipeline in the correct order.*** ***Usage Model: Normal operation when the VS function is required and use of the Vertex Cache is permissible.***

4.2 VS Stage Input

As a stage of the GEN4 3D pipeline, the VS stage receives inputs from the previous (VF) stage. Refer to *3D Overview* for an overview of the various types of input to a 3D Pipeline stage. The remainder of this subsection describes the inputs specific to the VS stage.

4.2.1 State

4.2.1.1 URB_FENCE

Refer to *3D Overview* for a description of how the VS stage processes this command.

4.2.1.2 VS_STAT E [Pre-DevSNB]

The following table describes the format and contents of the VS_STATE structure referenced by the **Pointer to VS State** field of the 3DSTATE_PIPELINED_POINTERS command.

3DSTATE_VS			
Project:	[Pre-DevSNB]	**Length Bias:**	2

For **[Pre-DevSNB]**, the state used by VS is defined with this inline state packet.

DWord	Bit	Description
0	31:6	**Kernel Start Pointer** Project: All Format: **[Pre-DevIL]:** Format = GeneralStateOffset[31:6] FormatDesc **[DevIL]:** Format = InstructionBaseOffset[31:6] Address: GraphicsAddress[31:0] Surface Type: U32 Range 0..2^32-1 This field specifies the starting location (1st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Base Address [Pre-DevIL]** or **Instruction Base Address [DevIL]**. This field is ignored if **VS Function Enable** is DISABLED.

Errata De	scription	Project
[BWT007	Instructions pointed at by offsets from General State must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)	**[DevBW-A,B]**

5:4	**Reserved** Project: All	Format: MBZ

3DSTATE_VS

	3:1	**GRF Register Count** Project: All Security: None Access: None Exists If: Always Default Value: 0h DefaultVaueDesc Mask: MMIO(0x2000)#16 Format: U32 FormatDesc Address: GraphicsAddress[31:0] Surface Type: U32 Range 0..2^32-1 Defines the number of GRF Register Blocks used by the kernel. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. This field is ignored if **VS Function Enable** is DISABLED.
	0	**Reserved** Project: All Format: MBZ
1	31	**Single Program Flow (SPF)** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). If set, the VS unit will only dispatch 1-vertex thread payloads. See CR0 description in *ISA Execution Environment*.

Value Name	Description	Project
0 Multiple Program Flows	Multiple Program Flows (1- or 2-vertex threads spawned, operating under normal (SIMD4x2) mode)	All
1 Single Program Flow	Single Program Flow (only 1-vertex threads spawned, operating under SPF EU mode)	All

Programming Notes
This field is ignored if **VS Function Enable** is DISABLED.

	30:26	**Reserved** Project: All Format: MBZ

3DSTATE_VS

25:18 Binding Table Entry Coun

Project: All

Format: U8 FormatDesc

Range [0,255]

Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). If set, the VS unit will only dispatch 1-vertex thread payloads. See CR0 description in *ISA Execution Environment*.

Programming Notes

- This field is ignored if **VS Function Enable** is DISABLED.
- [DevILK:A], [DevILK:B] MBZ
- **Note:** For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache

17 Thread Priority

Project: All

Specifies the priority of the thread for dispatch:

Value Na	me	Description	Project
0	Normal	Normal priority	All
1	High	High priority	All

Programming Notes

- This field is ignored if **VS Function Enable** is DISABLED.
- **[Pre-DevIL]**: this field must be zero.

16 Floating Point Mode:

Project: All

Specifies the initial floating point mode used by the dispatched thread.

Value Na	me	Description	Project
0h	IEEE-754 rules	Use IEEE-754 Rules	All
1h	Alternate rules	Use alternate rules	All

Programming Notes

This field is ignored if **VS Function Enable** is DISABLED.

15:14 Reserved Project: All Format: MBZ

3DSTATE_VS		
	13	**Illegal Opcode Exception Enable** Project: All Format Enable FormatDesc This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. **Programming Notes** • This field is ignored if **VS Function Enable** is DISABLED.
	12	**Reserved** Project: All Format: MBZ
	11	**MaskStack Exception Enable** Project: All Format: Enable FormatDesc This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.
	10:8	**Reserved** Project: All Format: MBZ
	7	**Software Exception Enable** Project: All Format Enable FormatDesc This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.
	6:0	**Reserved** Project: All Format: MBZ

3DSTATE_VS		
2	31:10	**Scratch Space Base Of** Project: All Format: GeneralStateOffset[31:10] FormatDesc Range 0..2^32-1 Specifies the starting location of the scratch space area allocated to this FF unit as a 1K-byte aligned offset from the **General State Base Address**. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by **Per-Thread Scratch Space**. The computed offset of the thread-specific portion will be passed in the thread payload as **Scratch Space Offset**. The thread is expected to utilize "stateless" DataPort read/write requests to access scratch space, where the DataPort will cause the **General State Base Address** to be added to the offset passed in the request header. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.
	9:4	**Reserved** Project: All Format: MBZ
	3:0	**Per-Thread Scratch Space** Project: All Format: U4 power of 2 Bytes over 1K Bytes FormatDesc Range [0,11] indicating [1K Bytes, 2M Bytes] Specifies the amount of scratch space to be allocated to each thread spawned by this FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** can each get **Per-Thread Scratch Space** size without exceeding the driver-allocated scratch space. **Programming Notes** • This field is ignored if **VS Function Enable** is DISABLED. • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it.
3	31	**Reserved** Project: All Format: MBZ
	30:25	**Constant URB Entry Read Length** Project: All Format: U6 FormatDesc Range [0,63] Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.

3DSTATE_VS		
	24	**Reserved** Project: All Format: MBZ
	23:18	**Constant URB Entry Read Offset** Project: All Format: U6 FormatDesc Range [0,63] Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.
	17	**Reserved** Project: All Format: MBZ
	16:11	**Vertex URB Entry Read Length** Project: All Format: U6 FormatDesc Range [1,63] Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments. **Programming Notes** • This field is ignored if **VS Function Enable** is DISABLED. • It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread.
	10	**Reserved** Project: All Format: MBZ
	9:4	**Vertex URB Entry Read Offset** Project: All Format: U6 FormatDesc Range [0,63] Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.

3DSTATE_VS		
	3:0	**Dispatch GRF Start Register for URB Data** Project: All Format: U4 FormatDesc Range [0,15] indicating GRF [R0,R15] Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.
4	31	**Reserved** Project: All Format: MBZ
	30:25	**Maximum Number of Threads** Project: All Format: U5 representing thread count - 1 FormatDesc Range [Pre-DevCTG:B]Range = [0,15] indicating thread count of [1,16] [DevCTG:B+]Range = [0,31] indicating thread count of [1,32] [DevILK]Range = [0, 71] indicating thread count of [1,72] Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.
	24	**Reserved** Project: All Format: MBZ
	23:19	**URB Entry Allocation Size** Project: All Format: U5 count (of 512-bit units) – 1 FormatDesc Range [0,31] = [1,32] 512-bit units = [2,64] 256-bit URB rows Specifies the length of each URB entry owned by this FF unit. **Programming Notes** • This field is always used (even if **VS Function Enable** is DISABLED). • Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE.
	18	**Reserved** Project: All Format: MBZ

3DSTATE_VS		
	17:11	**Number of URB Entries** Project: All Format: **[DevILK]** FormatDesc Format = U9 shift right by 2, see valid settings below **[DevCTG-B]**: Format = U7, see valid settings below **[Pre DevCTG-B]**: Format = U6, see valid settings below **DevBW-A,B Restriction**: Format = U6, see valid settings below Range **[DevILK]** Range = [2=8 entries, 3=12 entries, 4=16 entries, 8=32 entries, 16=64 entries, 24=96 entries, 32=128 entries, 42=168 entries, 48=192 entires, 56=224 entries, 64=256 entries] (see restriction above) **[DevCTG-B]**: Range = [8,12, 16, 32, 64] (see restriction above) **[Pre DevCTG-B]**: Range = [8,12, 16, 32] (see restriction above) **DevBW-A,B Restriction**: Range = [8,12, 16] Specifies the number of URB entries that are used by this FF unit. **Programming Notes** • This field is always used (even if **VS Function Enable** is DISABLED). • Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. • This field must be programmed to 12 or greater in order to process TRISTRIP_ADJ primitives, otherwise operation is UNDEFINED (possible hang).

<table>
<tr><th colspan="3">3DSTATE_VS</th></tr>
<tr><td></td><td>10</td><td>Statistics Enable
Project: All
Format: Enable FormatDesc
Address: GraphicsAddress[31:0]
Surface Type: U32
Range 0..2^32-1
If ENABLED, this FF unit will engage in statistics gathering. See the Statistics Gathering section later in this chapter. If DISABLED, statistics information associated with this FF stage will be left unchanged.

Programming Notes
This field is effectively if VS Function Enable is DISABLED.</td></tr>
<tr><td></td><td>9:0</td><td>Reserved Project: All Format: MBZ</td></tr>
<tr><td>5</td><td>31:5</td><td>Sampler State Offset
Project: All
Format: GeneralStateOffset[31:5] FormatDesc
This field, together with the General State Base Address, specifies the starting location of the Sampler State Table used by threads spawned by this FF unit. It is specified as a 32-byte-granular offset from the General State Base Address. The Sampler will apply the offset to the General State Base Address when accessing Sampler State data.

Programming Notes
This field is ignored if VS Function Enable is DISABLED.

<table><tr><th>Errata De</th><th>scription</th><th>Project</th></tr><tr><td>Errata BWT007</td><td>Sampler state pointed at by offsets from General State must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)</td><td>[DevBW-A,B]</td></tr></table></td></tr>
<tr><td></td><td>4:3</td><td>Reserved Project: All Format: MBZ</td></tr>
</table>

3DSTATE_VS		
	2:0	**Sampler Count** Project: All but ILK Format: U3 FormatDesc Range [0,4] Specifies how many samplers (in multiples of 4) the vertex shader 0 kernel uses. Used only for prefetching the associated sampler state entries. 0: no samplers used 1: between 1 and 4 samplers used 2: between 5 and 8 samplers used 3: between 9 and 12 samplers used 4: between 13 and 16 samplers used **Programming Notes** This field is ignored if **VS Function Enable** is DISABLED.
	31:0	**Reserved** Project: [DevILK:A-B] Format: MBZ
6	31:2	**Reserved** Project: All Format: MBZ

3DSTATE_VS

1 Vertex Cache Di

Project: All

Format: Disable FormatDesc

This bit controls the operation of the Vertex Cache. This field is always used.

Value Na	me	Description	Project
0	Vertex Cache is DISABLED and the VS Function is ENABLED	the Vertex Cache is not used and all incoming vertices will be passed to VS threads.	All
1	Vertex Cache is ENABLED and the VS Function is ENABLED	incoming vertices that do not hit in the Vertex Cache will be passed to VS threads.	All
2	Vertex Cache is ENABLED and the VS Function is DISABLED	input vertices that miss in the Vertex Cache will be assembled and written to the URB, though pass thru the VS stage unmodified (not shaded).	

Programming Notes

- The Vertex Cache is invalidated whenever the Vertex Cache becomes DISABLED , whenever the VS Function Enable toggles, between 3DPRIMITIVE commands and between instances within a 3DPRIMITIVE command.
- See the Vertex Caching section (above) for details on implicit vertex cache disabling and the chicken bit available to turn of any implicit disable.

0 VS Function Enable

Project: All

Format: Enable FormatDesc

This field is always used.

Value Na	me	Description	Project
0	Disable	VF-generated vertices will pass thru the VS function and sent down the pipeline unmodified. The Vertex Cache is still available in this mode, if enabled.	All
1	Enable	VF-generated vertices will pass thru the VS function and sent down the pipeline unmodified. The Vertex Cache is still available in this mode, if enabled.	All

4.2.2 Input Vertices

Refer to *3D Overview* for a description of the vertex information input to the VS stage.

4.3 VS Thread Request Generation

The following discussion assumes the VS Function is ENABLED.

When the Vertex Cache is disabled, the VS unit will pass each pair of incoming vertices to a VS thread. Under certain circumstances (e.g., prior to a state change or pipeline flush) the VS unit will spawn a VS thread to process a single vertex. Note that, in this case, the "unused" vertex slot will be "disabled" via the Execution Mask provided by the VS unit to the GEN4 subsystem as part of the thread dispatch (See ISA doc). The VS thread will in itself be unaware of the single-vertex case, and therefore a single VS kernel can be used to process one or two vertices. (The performance of single-vertex processing will roughly equal the two-vertex case).

When the Vertex Cache is enabled, the VF unit will detect vertices that hit in the cache and mark these vertices so that they will bypass VS thread processing and be output via a reference to the cached VUE. The VS unit will keep track of these cache-hit vertices as it proceeds to process cache-miss vertices. The VS unit guarantees that vertices will exit the unit in the order they are received. This may require the VS unit to issue single-vertex VS threads to process a cache-miss vertex that has yet to be paired up with another cache-miss vertex (if this condition is preventing the VS unit from producing any output).

4.3.1 Thread Payload

The following table describes the payload delivered to VS threads.

Table 7. VS Thread Payload

DWord	Bit	Description
R0.7	31	Snapshot Flag If set, this thread has matched some debug criteria. (See *Debug* for further description).
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID:** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. (See *Debug* for further description). Format: Reserved for HW Implementation Use.

DWord Bit		Description
R0.5	31:10	**Scratch Space Offset:** Specifies the of the scratch space allocated to the thread, specified as a 1KB-granular offset from the **General State Base Address**. See **Scratch Space Base Offset** description in VS_STATE. (See *3D Pipeline* for further description on scratch space allocation). Format = GeneralStateOffset[31:10]
	9:8	Reserved
	7:0	**FFTID:** This ID is assigned by the FF unit and used to identify the thread within the set of outstanding threads spawned by the FF unit. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer.** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **General State Base Address** or **Dynamic State Base Address**. Format = GeneralStateOffset[31:5] **[Pre-DevGT]** Format = DynamicStateOffset[31:5] **[DevGT+]**
	4	Reserved
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space). (See *3D Pipeline* for further description). Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:0	Reserved : delivered as zeros (reserved for message header fields)
R0.1	31:27	Reserved
	26:16	**Handle ID 1:** This ID is assigned by the FF unit and used to identify the URB Return Handle 1 to the FF unit (as FF-specific index value, not a URB address). If only one vertex is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread's Channel Mask). (See *Generic FF Unit* for further description). Format = Reserved for HW Implementation Use.
	15:13	Reserved

DWord	Bit	Description
	12:0	**URB Return Handle 1:** This is the URB handle where the EU's upper channels (DWords 7:4) results are to be stored. If only one vertex is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread's Channel Mask). (See *Generic FF Unit* for further description). Format: U9 opaque handle **[Pre-DevILK]** Format: U10 opaque handle **[DevILK]** Format: U11 handle [DevSmallGT] Format: U12 handle **[DevGT+]** Format: U13 handle **[DevHSW+]**
R0.0	31:27	Reserved
	26:16	**Handle ID 0:** This ID is assigned by the FF unit and used to identify the URB Return Handle 0 to the FF unit (as FF-specific index value, not a URB address). (See *Generic FF Unit* for further description). Format = Reserved for HW Implementation Use.
	15:13	Reserved
	12:0	**URB Return Handle 0:** This is the URB handle where the EU's lower channels (DWords 3:0) results are to be stored. (See *Generic FF Unit* for further description). Format: U9 opaque handle **[Pre-DevILK]** Format: U10 opaque handle **[DevILK]** Format: U11 handle [DevSmallGT] Format: U12 handle **[DevGT+]** Format: U13 handle **[DevHSW+]**
[Varies] optional	255:0	Constant Data (optional) : **[Pre-DevGT]:** Some amount of constant data (possible none) can be extracted from the URB and passed to the thread following the R0 Header. The data is read from the Constant URB Entry at some offset (**Constant URB Entry Read Offset** state) from the handle. The amount of data provided is defined by the **Constant URB Entry Read Length** state. **[DevGT+]:** Some amount of constant data (possible none) can be extracted from the push constant buffer (PCB) and passed to the thread following the R0 Header. The amount of data provided is defined by the sum of the read lengths in the last 3DSTATE_CONSTANT_VS command (taking the buffer enables into account). The Constant Data arrives in a non-interleaved format.

DWord	Bit	Description
Varies	255:0	**Vertex Data :** Data from (possibly) one or (more typically) two Vertex URB Entries is passed to the thread in the thread payload. The **Vertex URB Entry Read Offset** and **Vertex URB Entry Read Length** state variables define the regions of the URB entries that are read from the URB and passed in the thread payload. These SVs can be used to provide a subset of the URB data as required by SW. The vertex data is laid out in the thread header in an interleaved format. The lower DWords (0-3) of these GRF registers always contain data from a Vertex URB Entry. The upper DWords (4-7) may contain data from another Vertex URB Entry. This allows two vertices to be processed (shaded) in parallel SIMD8 fashion. The VS kernel is not aware of the validity of the upper vertex.

4.4 VS Thread Execution

A VS kernel (with one exception mentioned below) assumes it is to operate on two vertices in parallel. Input data is either passed directly in the thread payload (including the input vertex data) or indirectly via pointers passed in the payload.

Refer to *ISA* chapters for specifics on writing kernels that operate in SIMD4x2 fashion.

Refer to 3D Pipeline Stage Overview (*3D Overview*) for information on FF-unit/Thread interactions.

In the (unlikely) event that the VS kernel needs to determine whether it is processing one or two vertices, the kernel can compare the **URB Return Handle 0** and **URB Return Handle 1** fields of the thread payload. These fields will be different if two vertices are being processed, and identical if one vertex is being processed. An example of when this test may be required is if the kernel outputs some vertex-dependent results into a memory buffer – without the test the single vertex case might incorrectly output two sets of results. Note that this is not the case for writing the URB destinations, as the Execution Mask will prevent the write of an undefined output.

4.4.1 Vertex Output

VS threads must always write the destination URB handles passed in the payload. VS threads are not permitted to request additional destination handles. Refer to 3D Pipeline Stage Overview (*3D Overview*) for details on how destination vertices are written and any required contents/formats.

4.4.2 Thread Termination

VS threads must signal thread termination, in all likelihood on the last message output to the URB shared function. Refer to the *ISA* doc for details on End-Of-Thread indication.

4.5 Primitive Output

The VS unit will produce an output vertex reference for every input vertex reference received from the VF unit, in the order received. The VS unit simply copies the `PrimitiveType`, `StartPrim`, and `EndPrim` information associated with input vertices to the output vertices, and does not use this information in any way. Neither does the VS unit perform any readback of URB data.

4.6 Other VS Functions

4.6.1 Statistics Gathering

The VS stage tracks a single pipeline statistic, the number of times a vertex shader is executed. A vertex shader is executed for each vertex that is fetched on behalf of a 3DPRIMITIVE command, unless the shaded results for that vertex are already available in the vertex cache. If the **Statistics Enable** bit in VS_STATE is set, the VS_INVOCATION_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) will be incremented for *each vertex* that is dispatched to a VS thread. This counter will often need to be incremented by 2 for each thread invoked since 2 vertices are dispatched to one VS thread in the general case.

5. Geometry Shader (GS) Stage

5.1 GS Stage Overview

The GS stage of the GENx 3D Pipeline is used to convert objects within incoming primitives into new primitives through use of a spawned GENx thread. When enabled, the GS unit buffers incoming vertices, assembles the vertices of each individual object within the primitives, and passes these object vertices (along with other data) to the GENx subsystem for processing by a GS thread.

When the GS stage is disabled, vertices flow through the unit unmodified, with the exception that the Vertex Header of each vertex is read back from the URB and passed along with the vertex to the next (CLIP) stage.

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Pipeline* chapter for a general description of a 3D Pipeline stage, as much of the GS stage operation and control falls under these "common" functions. I.e., most stage state variables and GS thread payload parameters are described in *3D Pipeline*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the GS stage, and any exceptions the GS stage exhibits with respect to common FF unit functions.

5.2 GS Stage Input

As a stage of the GENx 3D pipeline, the GS stage receives inputs from the previous VS stage. Refer to *3D Pipeline* for an overview of the various types of input to a 3D Pipeline stage. The remainder of this subsection describes the inputs specific to the GS stage.

5.2.1 State

5.2.1.1 3DST ATE_GS_SVB_INDEX [DevCTG+]

The 3DSTATE_GS_SVB_INDEX instruction is used to program geometry shader streamed vertex buffer indexes or the Internal Vertex Count state register.

Four independent index values are supported. Each instance of this instruction programs one of the indexes, selected by the **Index Number** field. All four indexes are delivered to the geometry shader thread, and kernel code is responsible for using the correct index for each data port message.

This instruction is treated like non-pipelined state, thus a pipeline flush is executed before the indexes are changed.

3DSTATE_GS_SVB_INDEX

Project:	[DevILK, DevCTG,]	Length Bias:	2

The 3DSTATE_GS_SVB_INDEX instruction is used to program geometry shader streamed vertex buffer indexes.

DWord	Bit	Description
0	31:29	**Command Type** **Default Value: 3h GFXPIPE Format: OpCode**
	28:27	**Command SubType** **Default Value: 3h GFXPIPE_3D Format: OpCode**
	26:24	**3D Command Opcode** **Default Value: 1h 3DST ATE_NONPIPELINED Format: OpCode**
	24:16	**3D Command Sub Opcode** **Default Value: 0Bh 3DST ATE_GS_SVB_INDEX Format: OpCode**
	15:8	**Reserved Pro ject: All Format: MBZ**
	7:0	**DWord Length** **Default Value: 1h Excludes DWord (0,1)** **Format: =n Total Length - 2** **Project: [Pre-De vILK]**
	7:0	**DWord Length** **Default Value: 2h Excludes DWord (0,1)** **Format: =n Total Length - 2** **Project: [De vILK**
2	31:0	**Streamed Vertex Buffer Index (SVBI)** **Project: A ll** **Format: U32 Index into an SVB** **Range 0..2^ 27-1** **This field contains the value to be loaded into the SVBI register selected by Index Number.** Programming Notes: [DevILK+]: If a buffer is not enabled then the SVBI must be set to 0x0 in order to not cause overflow in that SVBI.

3DSTATE_GS_SVB_INDEX		
3	31:0	**Maximum Index** **Project: De v ILK** **Format: U32 Index into an SVB** **Range 0..2^ 27** **This field specifies the maximum value of the selected SVBI, enforced by the device. Software should set this field to one past the last valid vertex index, so that the clamped value can be used as a vertex count (see Internal Vertex Count in 3DPRIMITIVE.** **Programming Notes** • Once and SVBI reaches the Maximum Index then all SVBI values will discontinue to increment. • If a buffer is not enabled then the MaxSVBI must be set to 0xFFFFFFFF in order to not cause overflow in that SVBI.

5.2.1.2 GS_STAT E [Pre-DevSNB]

The following table describes the format and contents of the GS_STATE structure referenced by the **Pointer to GS State** field of the 3DSTATE_PIPELINED_POINTERS command.

[DevILK] Note: Any change in GS_STATE will require a URB_FENCE state to be sent before the next 3D_PRIMITIVE command. This is required since the hardware does not reset the dispatch ID on a PSP command alone.

GS_STATE

Project: [Pre-DevSNB]

Controls the GS stage hardware.

DWord	Bit	Description
0	31:6	**Kernel Start Pointer** Project: [Pre-ILK Address: GeneralStateOffset[31:6] Surface Type: Kernel This field specifies the starting location (1st GENx core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Base Address.** **Errata Description** / **Project**: BWT007 — Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) — [DevBW-A,B]
	31:6	**Kernel Start Pointer** Project: [DevILK] Address: InstructionBaseOffset[31:6] Surface Type: Kernel This field specifies the starting location (1st GENx core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **Instruction Base Address.**
	5:4	**Reserved Project: A II Format: MBZ**
	3:1	**GRF Register Count** **Project: All Format: U3 register block count - 1** **Defines the number of GRF Register Blocks used by the kernel. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16.**
	0	**Reserved Project: A II Format: MBZ**

GS_STATE		
1	31	**Single Program Flow (SPF)** **Project: A II** **Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1).** **Value Name / Description / Project** **0h Reserved / All** **1h Enable / Single Program Flow enabled / All**
	30:26	**Reserved Project: A II Format: MBZ**
	25:18	**Binding Table Entry Count Project: A II Format: U8** **Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state.** **Note: For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache.** **[DevILK] MBZ**
	17	**Thread Priority** **Project: A II** **Specifies the priority of the thread for dispatch** **Value Name / Description / Project** **0h Normal Priority / Normal Priority / All** **1h High Priority / High Priority / DevILK**
	16	**Floating Point Mode** **Project: A II** **Specifies the initial floating point mode used by the dispatched thread.** **Value Name / Description / Project** **0h IEEE-754 / Use IEEE-754 Rules / All** **1h Alternate / Use alternate rules / All**
	15:14	**Reserved Project: A II Format: MBZ**
	13	**Illegal Opcode Exception Enable Project: A II Format: Enable** **This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.**
	12	**Reserved Project: A II Format: MBZ**

GS_STATE		
	11	**Mask Stack Exception Enable** Project: A II Format: Enable **This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*.**
	10:8	**Reserved** Project: A II Format: MBZ
	7	**Software Exception Enable** Project: A II Format: Enable **This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.**
	6:0	**Reserved** Project: A II Format: MBZ
2	31:10	**Scratch Space Base Pointer** Project: A II Format: GeneralStateOffset[31:10] **Specifies the location of the scratch space area allocated to this FF unit, specified as a 1KB-granular offset from the General State Base Address. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by Per-Thread Scratch Space.**
	9:4	**Reserved** Project: A II Format: MBZ
	3:0	**Per-Thread Scratch Space** **Project: A II** **Format: U4 power of 2 Bytes over 1K Bytes FormatDesc** **Range [0,11] indicating [1K Bytes, 2M Bytes]** **Specifies the amount of scratch space to be allocated to each thread spawned by this FF unit.** **The driver must allocate enough contiguous scratch space, starting at the Scratch Space Base Pointer, to ensure that the Maximum Number of Threads can each get Per-Thread Scratch Space size without exceeding the driver-allocated scratch space.**
3	31	**Reserved** Project: A II Format: MBZ
	30:25	**Constant URB Entry Read Length** **Project: A II** **Format: U6 FormatDesc** **Range [0 ,63]** **Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments.**
	24	**Reserved** Project: A II Format: MBZ
	23:18	**Constant URB Entry Read Offset** **Project: A II** **Format: U6 FormatDesc** **Range [0 ,63]** **Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload.**
	17	**Reserved** Project: A II Format: MBZ

GS_STATE		
	16:11	**Vertex URB Entry Read Length** **Project: A II** **Format: U6 FormatDesc** **Range [1 ,63]** **Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments.** **It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread.**
	10	**Reserved Project: A II Format: MBZ**
	9:4	**Vertex URB Entry Read Offset** **Project: A II** **Format: U6 FormatDesc** **Range [0 ,63]** **Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread.**
	3:0	**Dispatch GRF Start Register for URB Data** **Project: A II** **Format: U4 FormatDesc** **Range [0,15] indicating GRF [R0,R15]** **Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload.**
4	31:30	**Reserved Project: A II Format: MBZ**
	29:25	**Maximum Number of Threads** **Project: Pre-De vILK** **Format: U5 thread count – 1** **Range [0,1]** **Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock.** **Programming Notes:** • **When running in dual-thread mode, the Number of URB Entries field must contain an even number. Each thread will be allocated one half the total number of entries.** • **A URB_FENCE command must be issued subsequent to any change to the value in this field (via PIPELINE_STATE_POINTERS) and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER). See *Graphics Processing Engine* (Command Ordering Rules)** **Format = U5 representing thread count – 1** **Range = [0,1] indicating thread count of [1,2]**

GS_STATE		
	29:25	**Maximum Number of Threads** **Project: De vILK** **Format: U5 thread count – 1** **Range [DevILK] [0] indicating thread count of [1]** **[0,31] indicating thread count of [1,32]** **Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock.** **Programming Notes:** • **When the Maximum Number of Threads > 1, the Number of URB Entries field must contain an <u>even</u> number.** • **A URB_FENCE command must be issued subsequent to any change to the value in this field (via PIPELINE_STATE_POINTERS) and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER). See *Graphics Processing Engine* (Command Ordering Rules)** • **[DevILK] If rendering is disabled, then the Maximum Number of Threads must be a value that is > 1.** • **[DevILK] If rendering is enabled and stream out is enabled then the Maximum Number of Threads must be == 1.**
	24	**Reserved** Project: All Format: MBZ
	23:19	**URB Entry Allocation Size** **Project: A II** **Format: U5 count (of 512-bit units) – 1** **Range [0,31] = [1,32] 512-bit units = [2,64] 256-bit URB rows** **Specifies the length of each URB entry owned by this FF unit.** **Programming Notes** **Project** **Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE.** **All**
	18	**Reserved Project: A II Format: MBZ**

GS_STATE

	17:11	**Number of URB Entries** **Project: A ll** **Format: U7 Count of URB entries** **Range [1,32] if GS enabled, otherwise ignored.** **Specifies the number of URB entries that are used by this FF unit.** **Programming Notes — Project** **When running in dual-thread mode, the Number of URB Entries field must contain an even number. Each thread will be allocated one half the total number of entries. — All** **If ENABLED, the GS stage must be allocated at least one URB entry — All** **Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. — All**
	10	**GS Statistics Enable** **Project: A ll** **Format: Enable** **This bit controls whether GS-unit-specific statistics register(s) can be incremented.** **Value Na me — Description — Project** **0h Disable — GS_INVOCATIONS_COUNT cannot increment — All** **1h Enable — GS_INVOCATIONS_COUNT can increment — All** **Programming Notes — Project** **When this bit is set, the CLIP stage must not be in pass-through mode and it must have its GS Output Object Statistic Enable bit set. — All**
	9	**SO Statistics Enable** **Project: De vILK** **Format: Enable** **This bit controls whether certain StreamOutput statistics register(s) can be incremented.** **Value Na me — Description — Project** **0h Disable — SO_NUM_PRIMS_WRITTEN and SO_PRIM_STORAGE_NEEDED cannot increment — DevILK** **1h En able — SO_NUM_PRIMS_WRITTEN and SO_PRIM_STORAGE_NEEDED can increment — DevILK**
	9	**Reserved** Project: Pre-DevILK Format: MBZ
	8	**Reserved Pro ject Pre-DevILK Forma t: MBZ** **:**

GS_STATE

	8	**Rendering Enabled** **Project: De vILK** **Format: U1** **This state bit is used to indicate whether or not the GS thread will be allocating and outputting VUE handles for rendering. This bit must be set if the thread will attempt to allocate a handle. If clear, the GS thread must not allocate handles (e.g., when only performing stream output without concurrent rendering).**
	7:0	**Reserved Project: Pre-DevILK Format: MBZ**
5	31:5	**Sampler State Pointer** **Project: A ll** **Format: GeneralStateOffset[31: 5]** **This field specifies the starting location of the Sampler State Table used by threads spawned by this FF unit. It is specified as a 32-byte-granular offset from the General State Pointer.** **Errata De scription — Project** **Sampler state pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) — BW-A,B**
	4:3	**Reserved Project: A ll Format: MBZ**
	2:0	**Sampler Count** **Project: All** **Format: U3** **Specifies how many samplers (in multiples of 4) the geometry shader kernel uses. Used only for prefetching the associated sampler state entries.** **[DevILK] MBZ** **Value Na me — Description — Project** **0h — No Samplers — no samplers used — All** **1h — 1-4 Samplers — between 1 and 4 samplers used — All** **2h — 5-8 Samplers — between 5 and 8 samplers used — All** **3h 9-12 Samplers — between 9 and 12 samplers used — All** **4h 13-1 6 Samplers — between 13 and 16 samplers used — All** **5h-7h Res erved — Reserved — All**
6	31	**Reserved Project: A ll Format: Enable**

GS_STATE		
	30	**Reorder Enable Project: A ll Format: Ena ble** **This bit controls whether the GS unit reorders TRISTRIP/TRISTRIP_REV vertices passed in the GS thread payload.** **If ENABLED, the GS unit will reorder the vertices for "odd-numbered" triangles originating from TRISTRIP topologies and "even-numbered" triangles originating from TRISTRIP_REV topologies. (Note that the first triangle is considered "triangle 0", which is even-numbered).** **With respect to the PrimType passed in the GS thread payload, the GS unit passes TRISTRIP when the vertices are not reordered, and TRISTRIP_REV when the vertices are reordered (regardless of whether a TRISTRIP or TRISTRIP_REV topology was being processed)** **If DISABLED, TRISTRIP/TRISTRIP_REV vertices are not reordered, and always passed in the order they are received from the pipeline. The GS unit will still toggle PrimType on alternating (as described above) so that the GS thread can perform the reordering internally (or do whatever is necessary to account for the non-reordering of its input).**
	29	**Discard Adjaceny Project: Pre-DevILK Format: Enable** **When set, adjacent vertices will not be passed in the GS payload when objects with adjacency are processed. Instead, only the non-adjacent vertices will be passed in the same fashion as the without-adjacency form of the primitive. Software should set this bit whenever a GS kernel is used that does not expect adjacent vertices. This allows both with-adjacency/without-adjacency variants of the primitive to be submitted to the pipeline (via 3DPRIMITIVE) – the GS unit will silently discard any adjacent vertices and present the GS thread with only the internal object.** **When clear, adjacent vertices will be passed to the GS thread, as dictated by the incoming primitive type. Software should only clear this bit when a GS kernel is used that does expect adjacent vertices. E.g., if the GS kernel is compiled to expect a TRIANGLE_ADJ object, software must clear this bit.** **Software should also clear this bit if the GS kernel expects a POINT object (which doesn't have a with-adjacency variant).** **This bit is used to provide limited compatibility between submitted primitive types and the object type expected by the GS kernel. The only hardware assistance is to allow the submission of a with-adjacency variant of a primitive when operating with a GS kernel that expects the without-adjacency variant of the object. (E.g., when the GS kernel is compiled to expect a TRIANGLE object, software should set this bit just in case a TRILIST_ADJ is submitted to the pipeline.) Note that the GS unit is otherwise not aware of the object type that is expected by the GS kernel. It is up to software to ensure that the submitted primitive type (in 3DPRIMITIVE) is otherwise compatible with the object type expected by the GS kernel. (E.g., if the GS kernel expects a LINE_ADJ object, only LINELIST_ADJ or LINESTRIP_ADJ should be submitted, otherwise the GS kernel will produce unpredictable results.)** **Also note that it is possible to craft a GS kernel which can accept any object type that's thrown at it by first examining the PrimType passed in the payload and then using this info to correctly interpret the number of vertices passed in the payload.**
	29 Res erved	Project: [Dev-ILK] Format: MBZ
	28	**SVBI Payload Enable Project: CTG+ Format: Enable** **This field controls whether the optional R1 header phase containing the Streamed Vertex Buffer Indices is delivered.**

GS_STATE		
	27	**SVBI Post-Increment Enable** **Project: DevILK Format: Enable** **This bit should be set whenever the GS thread is performing only the SO function (no GS, no Render). Setting this bit allows the GS FF unit to post-increment the SVBI values after GS threads are dispatched. This allows the threads to complete without the need for further synchronization. The increment value is specified by SVBI Post-Increment Value.** **If this bit is clear, the GS thread must use the FF_SYNC message to report the amount of data it will output and receive the appropriate (synchronized) SVBI values in the writeback. This is required whenever the GS function is required and software cannot guarantee that the GS shader will output a constant amount of output (vertices).** **Programming Note: Since the GS threads are provided with overflow-clamped SVBI inputs, they are always responsible for overflow detection given those inputs.**
	26	**Reserved Project: DevILK Format: MBZ**
	25:16	**SVBI Post-Increment Value** **Project: DevILK Format: U10** **If SVBI Post-Increment Enable is set, all the SVBI state registers will be incremented by this value after the dispatch of every GS thread. If SVBI Post-Increment Enable is clear, this field is ignored.**
	15:7	**Reserved Project: Dev-ILK-B Format: MBZ**

<table>
<tr><td></td><td>6</td><td>
GS Pass Through Enable Project: De vILK-B Format: Enable
[DevILK
“1” indicates GS Threads are not launched.
“0” indicates GS Threads are launched.
Note: This bit is looked at by GS only when GS is enabled.
It is used to enable GS readbacks for user clip distances when User clip planes are enabled.
<table>
<tr><td colspan="2">Desired Functionality</td><td colspan="3">Programming</td><td>Affect</td></tr>
<tr><td>GShader functionally ON
(GS thread launch desired)</td><td>User Clip Planes Enabled</td><td>GS enable bit</td><td>GS Pass Thru Enable bit</td><td>Readback User Clip bits0-3/4-7</td><td>Affect</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>N/A (bit not looked at unless GS enable bit is set)</td><td>0</td><td>GS threads NOT launched
Clip distances NOT readback
Incoming Vertices passed thru to CL FF</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>GS threads NOT launched
Clip distances readback
Incoming Vertices passed thru to CL FF</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>GS threads launched
Clip distances NOT readback</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>GS threads launched
Clip distances readback</td></tr>
</table>
</td></tr>
<tr><td></td><td>31:0</td><td>Reserved Project: [DevILK] Format: MBZ</td></tr>
<tr><td></td><td>5</td><td>
User Clip Planes 4-7 Enabled Project: De vILK Format: Enable
“1” indicates User Clip Planes 4-7 are enabled. Hence GS needs to perform an extra Readback Phase and send this data to CL fixed function.
“0” indicates User Clip Planes 4-7 are NOT enabled. Hence No extra Readback phase Required.
</td></tr>
</table>

GS_STATE		
	4	**User Clip Planes 0-3 Enabled** **Project: De vILKD** **Format: Enable** **"1" indicates User Clip Planes 0-3 are enabled. Hence GS needs to send the data corresponding to these planes to CL fixed function.** **"0" indicates User Clip Planes 0-3 are NOT enabled. Hence GS need NOT to send the data corresponding to these planes to CL fixed function**
	27:4	**Reserved Project: De vILK Format: MBZ**
	3:0	**Maximum VPIndex Project: A II Format: U4 index value (# of viewports -1)** **This field specifies the maximum valid VPIndex value, corresponding to the number of active viewports. If the source of the VPIndex exceeds this maximum value, a VPIndex value of 0 is passed down the pipeline. Note that this clamping does not affect a VPIndex value stored in the URB.**

5.3 Object Staging

The GS unit's Object Staging Buffer (OSB) accepts primitive topologies as a stream of incoming vertices, and spawns a thread for each individual object within the topology.

5.4 GS Thread Request Generation

5.4.1 Object Vertex Ordering

The following table defines the number and order of object vertices passed in the Vertex Data portion of the GS thread payload, assuming an input topology with *N* vertices. The ObjectType passed to the thread is, by default, the incoming PrimTopologyType. Exceptions to this rule (for the TRISTRIP variants) are called out.

PrimTopologyType	Order of Vertices in Payload	GS Notes
<PRIMITIVE_TOPOLOGY> (N = # of vertices)	[<object#>] = (<vert#>,…); [{modified PrimType passed to thread}]	
POINTLIST	[0] = (**0**); [1] = (**1**); …; [N-2] = (**N-2**);	
POINTLIST_BF	N/A	
LINELIST (N is multiple of 2)	[0] = (0,**1**); [1] = (2,**3**); …; [(N/2)-1] = (N-2,**N-1**)	

PrimTopologyType	Order of Vertices in Payload	GS Notes
LINELIST_ADJ (N is multiple of 4)	[0] = (0,1,**2**,3); [1] = (4,5,**6**,7); …; [(N/4)-1)] = (N-4,N-3,**N-2**,N-1)	
LINESTRIP (N >= 2)	[0] = (0,**1**); [1] = (1,**2**); …; [N-2] = (N-2,**N-1**)	
LINESTRIP_ADJ (N >= 4)	[0] = (0,1,**2**,3); [1] = (1,2,**3**,4); …; [N-4] = (N-4,N-3,**N-2**,N-1)	
LINESTRIP_BF	N/A	
LINESTRIP_CONT	Same as LINESTRIP	**Handled same as LINESTRIP**
LINESTRIP_CONT_BF	Same as LINESTRIP	**Handled same as LINESTRIP**
LINELOOP (N >= 2)	[0] = (0,**1**); [1] = (1,**2**); [N] = (N-1,**0**);	**Not supported after GS.**
TRILIST (N is multiple of 3)	[0] = (0,1,**2**); [1] = (3,4,**5**); …; [(N/3)-1] = (N-3,N-2,**N-1**)	
RECTLIST	Same as TRILIST	**Handled same as TRILIST**
TRILIST_ADJ (N is multiple of 6)	[0] = (0,1,2,3,**4**,5); [1] = (6,7,8,9,**10**,11); …; [(N/6)-1] = (N-6,N-5,N-4,N-3,**N-2**,N-1)	
TRISTRIP (Reorder ENABLED) (N >= 3)	[0] = (0,1,**2**); {TRISTRIP} [1] = (1,**3**,2); {TRISTRIP_REV} [k even] = (k,k+1,**k+2**) {TRISTRIP} [k odd] = (k,**k+2**,k+1) {TRISTRIP_REV} [N-3] = (see above)	"Odd" triangles have vertices reordered, though identified as TRISTRIP_REV so the thread knows this
TRISTRIP (Reorder DISABLED) (N >= 3)	[0] = (0,1,**2**) {TRISTRIP} [1] = (1,2,**3**) {TRISTRIP_REV}; … [N-3] = (N-3,N-2,**N-1**) {TRISTRIP or TRISTRIP_REV}	"Odd" triangles do not have vertices reordered, though identified as TRISTRIP_REV so the thread knows this
TRISTRIP_REV (Reorder ENABLED) (N >= 3)	[0] = (0,**2**,1) {TRISTRIP_REV}; [1] = (1,2,**3**) {TRISTRIP}; …; [k even] = (k,**k+2**,k+1) {TRISTRIP_REV} [k odd] = (k,k+1,**k+2**) {TRISTRIP} [N-3] = (see above)	"Odd" triangles have vertices reordered, though identified as TRISTRIP so the thread knows this
TRISTRIP_REV (Reorder DISABLED) (N >= 3)	[0] = (0,1,**2**) {TRISTRIP_REV} [1] = (1,2,**3**) {TRISTRIP}; …; [N-3] = (N-3,N-2,**N-1**) {TRISTRIP or TRISTRIP_REV}	"Odd" triangles do not have vertices reordered, though identified as TRISTRIP so the thread knows this

PrimTopologyType	Order of Vertices in Payload	GS Notes
TRISTRIP_ADJ (N even, N >= 6)	N = 6 or 7: [0] = (0,1,2,5,**4**,3) N = 8 or 9: [0] = (0,1,2,6,**4**,3); [1] = (2,5,**6**,7,4,0); …; N > 10: [0] = (0,1,2,6,**4**,3); [1] = (2,5,**6**,8,4,0); …; [k>1, even] = (2k,2k-2, 2k+2, 2k+6,**2k+4**, 2k+3); [k>2, odd] = (2k, 2k+3, **2k+4**, 2k+6, 2k+2, 2k-2);…; Trailing object: [(N/2)-3, even] = (N-6,N-8,N-4,N-1,**N-2**,N-3); [(N/2)-3, odd] = (N-6,N-3,**N-2**,N-1,N-4,N-8);	"Odd" objects have vertices reordered.
TRIFAN (N > 2)	[0] = (0,1,**2**); [1] = (0,2,**3**); …; [N-3] = (0, N-2, **N-1**);	**Only used by OGL**
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON	Same as TRIFAN	
QUADLIST (N is multiple of 4)	[0] = (0,1,2,**3**); [1] = (4,5,6,**7**); …; [(N/4)-1] = (N-4,N-3,N-2,**N-1**);	**Not supported after GS.**
QUADSTRIP (N is multiple of 2, N >=4)	**[0] = (0,1,3,2);** **[1] = (2,3,5,4); … ;** **[(N/2)-2] = (N-4,N-3,N-1,N-2);**	**Not supported after GS.**

5.4.2 GS Thread

Table 5-1 shows the layout of the payload delivered to GS threads. Refer to 3D Pipeline Stage Overview (*3D Pipeline*) for details on those fields that are common amongst the various pipeline stages.

Table 5-1. GS Thread Payload

GRF DWord	Bit De	scription
R0.7	31	Reserved
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the scratch space allocated to this thread, specified as a 1KB-aligned offset from the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Reserved
	7:0	**FFTID.** This ID is assigned by the fixed function unit and is relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **General State Base Address** or **Dynamic State Base Address**. Format = GeneralStateOffset[31:5] **[Pre-DevSNB]**
	4	Reserved
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space). (See *Generic Pipeline Stage* for further description). **Programming Notes:** • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it. Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:10	Reserved : delivered as zeros (reserved for message header fields)

GRF DWord	Bit De	scription
	9	**Edge Indicator [1].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex2 to Vertex0 is an exterior edge of the polygon (i.e., this is the last or only triangle of the polygon). If clear, that edge is an interior edge. The kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved
	8	**Edge Indicator [0].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex0 to Vertex1 is an exterior edge of the polygon (i.e., this is the first or only triangle of the polygon). If clear, that edge is an interior edge. The kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved
	7	[**Pre-Dev-ILK**]: Reserved: MBZ **[Dev-ILK+]: Rendering Enabled Hint**: This is a copy of the corresponding GS_STATE/3DSTATE_GS bit . This bit can be used to inform the GS kernel whether or not it needs to output VUEs down the pipeline for possible rendering (as the state which controls whether rendering is enabled can change after the kernel is compiled). Of course, SW is free to use this bit for other purposes. Format: U1
	6:5	Reserved
	4:0	**Primitive Topology Type.** This field identifies the Primitive Topology Type associated with the primitive containing this object. It indirectly specifies the number of input vertices included in the thread payload. Note that the GS unit may toggle this value between TRISTRIP and TRISTRIP_REV, as described in 5.4.1. Format: See *3D Pipeline*
R0.0	31:23	Reserved
	22:16	**[Pre-DevILK]: Handle ID.** This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. Format: Reserved for Implementation Use [**DevILK+**]: Reserved
	15:9	Reserved
	8:0	**[Pre-DevILK: URB Return Handle.** This is the initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the first destination URB entry. [**DevILK+**]: Reserved
Streamed Vertex Buffer Index Values (passed in R1) (only included for [DevCTG+] and SVBI Payload Enable is set.		
R1.7		**[DevILK+]:** **Maximum Streamed Vertex Buffer Index 3** This is a copy of the **Maximum Index** field sent in the last 3DSTATE_GS_SVB_INDEX command targetting SVBI[3]. The thread will need to use the maximum indices of all bounds SOBs. **[Pre-DevILK]:** Reserved

GRF DWord	Bit De	scription
R1.6		**[DevILK+]:** **Maximum Streamed Vertex Buffer Index 2** **[Pre-DevILK]:** Reserved
R1.5		**[DevILK+]:** **Maximum Streamed Vertex Buffer Index 1** **[Pre-DevILK]:** Reserved
R1.4		**[DevILK+]:** **Maximum Streamed Vertex Buffer Index 0** **[Pre-DevILK]:** Reserved
R1.3	31:0	**Streamed Vertex Buffer Index 3** This field represents the initial value of the index #3. Format = U32 Range = [0,2^{27}-1]
R1.2	31:0	**Streamed Vertex Buffer Index 2**
R1.1	31:0	**Streamed Vertex Buffer Index 1**
R1.0	31:0	**Streamed Vertex Buffer Index 0**
[Varies] optional	31:0	**Constant Data (optional) :** **[Pre-DevSNB]:** Some amount of constant data (possible none) can be extracted from the URB and passed to the thread following the R0 Header. The data is read from the Constant URB Entry at some offset (**Constant URB Entry Read Offset** state) from the handle. The amount of data provided is defined by the **Constant URB Entry Read Length** state. The Constant Data arrives in a non-interleaved format.
Varies	31:0	**Vertex Data.** There can be up to 6 vertices supplied, each with a size defined by the **Vertex URB Entry Read Length** state. The amount of data provided for each vertex is defined by the **Vertex URB Entry Read Length** state Vertex 0 DWord 0 is located at Rn.0, Vertex 0 DWord 1 is located at Rn.1, etc. Vertex 1 DWord 0 immediately follows the last DWord of Vertex 0, and so on.

Figure 5 GS Dispatch Layouts

Table 5-2. GS Thread Payload shows the detailed layout of the payload delivered to GS threads.

Refer to 3D Pipeline Stage Overview (*3D Pipeline*) for details on those fields that are common amongst the various pipeline stages.

Table 5-2. GS Thread Payload

GRF DWord	Bit De	scription
	30:0	Reserved

GRF DWord	Bit De	scription
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the scratch space allocated to this thread, specified as a 1KB-aligned offset from the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Reserved
	7:0	**FFTID.** This ID is assigned by the fixed function unit and is relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer.** Specifies the location of the Sampler State Table to be used by this thread, specified as a 32-byte granular offset from the **Dynamic State Base Address**.
	4	Reserved
	3:0	**Per Thread Scratch Space.** Specifies the amount of scratch space allowed to be used by this thread. The value specifies the power that two will be raised to (over determine the amount of scratch space). (See *Generic Pipeline Stage* for further description). **Programming Notes:** • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it. Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:24	**Semaphore Index**. This is a Dword index to be used in URB_ATOMIC commands if the thread. This information is only required for pull-model vertex inputs and InstanceCount>1. Format = U8
	23	Reserved
	22	**Hint**: This is a copy of the corresponding 3DSTATE_GS bit . Format: U1
	21:16	**Primitive Topology Type.** This field identifies the Primitive Topology Type associated with the primitive containing this object. It indirectly specifies the number of input vertices included in the thread payload. Note that the GS unit may toggle this value between TRISTRIP and TRISTRIP_REV. If the **Discard Adjaceny** bit is set, the topology type passed in the payload is UNDEFINED. Format: See *3D Pipeline*

GRF DWord	Bit De	scription
	15:12	Reserved
	11:0	**Semaphore Handle:** This is the URB handle pointing to the first GS semaphore DWord in the URB. Software is responsible for statically allocating the 96 HS semaphore Dwords in the URB. E.g., room for this allocation can be made in any FF URB range by using allocation size and handle count parameters that, when multiplied together, leave at least 96 Dwords of unused space at the end of the URB region., Format: U12 handle
R0.1	31:27	**GS Instance ID 1**. For each input object, the GS unit can spawn multiple threads (instances). This field starts at zero for the first instance of an object and increments for subsequent instances. Format: U5
	26:25	Reserved
	24:16	**Handle ID 1:** This ID is assigned by the FF unit and used to identify the URB Return Handle 1 to the FF unit (as FF-specific index value, not a URB address). If only one object/instance is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread's Channel Mask). Format = Reserved for HW Implementation Use.
	15:12	Reserved
	11:0	**URB Return Handle 1:** This is the URB handle where the EU's upper channels (DWords 7:4) results are to be stored. If only one object/instance is to be processed (shaded) by the thread, this field will effectively be ignored (no results are stored for these channels, as controlled by the thread's Channel Mask). Format: U12 handle
R0.0	31:27	**GS Instance ID 0**. For each input object, the GS unit can spawn multiple threads (instances). This field starts at zero for the first instance of an object and increments for subsequent instances. Format: U5
	26:25	Reserved
	24:16	**Handle ID 0:** This ID is assigned by the FF unit and used to identify the URB Return Handle 0 to the FF unit (as FF-specific index value, not a URB address). Format = Reserved for HW Implementation Use.
	15:12	Reserved
	11:0	**URB Return Handle 0:** This is the URB handle where the EU's lower channels (DWords 3:0) results are to be stored. Format: U12 handle
The following register is included only if Include PrimitiveID is enabled.		
R1.7-R1.5	31:0	**Reserved: MBZ**

GRF DWord	Bit De	scription
R1.4	31:0	**Primitive ID 1** . This field contains the Primitive ID associated with (all instances) of input object 1. Only valid in DUAL_OBJECT mode. Format: U32
R1.3-R1.1	31:0	**Reserved: MBZ**
R1.0	31:0	**Primitive ID 0** . This field contains the Primitive ID associated with (all instances) of input object 0 Format: U32
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled.		
Rn.7	31:16	**ICP 7 Handle ID**
	15:12	Reserved
	11:0	**ICP 7 Handle**
Rn.6	31:16	**ICP 6 Handle ID**
	15:12	Reserved
	11:0	**ICP 6 Handle**
Rn.5	31:16	**ICP 5 Handle ID**
	15:12	Reserved
	11:0	**ICP 5 Handle**
Rn.4	31:16	**ICP 4 Handle ID**
	15:12	Reserved
	11:0	**ICP 4 Handle**
Rn.3	31:16	**ICP 3 Handle ID**
	15:12	Reserved
	11:0	**ICP 3 Handle**
Rn.2	31:16	**ICP 2 Handle ID**
	15:12	Reserved
	11:0	**ICP 2 Handle**
Rn.1	31:16	**ICP 1 Handle ID**
	15:12	Reserved
	11:0	**ICP 1 Handle**
Rn.0	31:16	**ICP 0 Handle ID**
	15:12	Reserved
	11:0	**ICP 0 Handle**
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled and when ICP Count >7.		

GRF DWord	Bit De	scription
Rn+1.7	31:16	**ICP 15 Handle ID**
	15:12	Reserved
	11:0	**ICP 15 Handle**
Rn+1.6	31:16	**ICP 14 Handle ID**
	15:12	Reserved
	11:0	**ICP 14 Handle**
Rn+1.5	31:16	**ICP 13 Handle ID**
	15:12	Reserved
	11:0	**ICP 13 Handle**
Rn+1.4	31:16	**ICP 12 Handle ID**
	15:12	Reserved
	11:0	**ICP 12 Handle**
Rn+1.3	31:16	**ICP 11 Handle ID**
	15:12	Reserved
	11:0	**ICP 11 Handle**
Rn+1.2	31:16	**ICP 10 Handle ID**
	15:12	Reserved
	11:0	**ICP 10 Handle**
Rn+1.1	31:16	**ICP 9 Handle ID**
	15:12	Reserved
	11:0	**ICP 9 Handle**
Rn+1.0	31:16	**ICP 8 Handle ID**
	15:12	Reserved
	11:0	**ICP 8 Handle**
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled and when ICP Count >15.		
Rn+2.7	31:16	**ICP 23 Handle ID**
	15:12	Reserved
	11:0	**ICP 23 Handle**
Rn+2.6	31:16	**ICP 22 Handle ID**
	15:12	Reserved
	11:0	**ICP 22 Handle**
Rn+2.5	31:16	**ICP 21 Handle ID**
	15:12	Reserved

GRF DWord	Bit De	scription
	11:0	**ICP 21 Handle**
Rn+2.4	31:16	**ICP 20 Handle ID**
	15:12	Reserved
	11:0	**ICP 20 Handle**
Rn+2.3	31:16	**ICP 19 Handle ID**
	15:12	Reserved
	11:0	**ICP 19 Handle**
Rn+2.2	31:16	**ICP 18 Handle ID**
	15:12	Reserved
	11:0	**ICP 18 Handle**
Rn+2.1	31:16	**ICP 17 Handle ID**
	15:12	Reserved
	11:0	**ICP 17 Handle**
Rn+2.0	31:16	**ICP 16 Handle ID**
	15:12	Reserved
	11:0	**ICP 16 Handle**
The following register is included only if SINGLE or DUAL_INSTANCE mode and Include Vertex Handles is enabled and when ICP Count >23.		
Rn+3.7	31:16	**ICP 31 Handle ID**
	15:12	Reserved
	11:0	**ICP 31 Handle**
Rn+3.6	31:16	**ICP 30 Handle ID**
	15:12	Reserved
	11:0	**ICP 30 Handle**
Rn+3.5	31:16	**ICP 29 Handle ID**
	15:12	Reserved
	11:0	**ICP 29 Handle**
Rn+3.4	31:16	**ICP 28 Handle ID**
	15:12	Reserved
	11:0	**ICP 28 Handle**
Rn+3.3	31:16	**ICP 27 Handle ID**
	15:12	Reserved
	11:0	**ICP 27 Handle**
Rn+3.2	31:16	**ICP 26 Handle ID**

GRF DWord	Bit De	scription
	15:12	Reserved
	11:0	**ICP 26 Handle**
Rn+3.1	31:16	**ICP 25 Handle ID**
	15:12	Reserved
	11:0	**ICP 25 Handle**
Rn+3.0	31:16	**ICP 24 Handle ID**
	15:12	Reserved
	11:0	**ICP 24 Handle**
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled.		
Rn.7	31:16	**Object 1 ICP 3 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 3 Handle**
Rn.6	31:16	**Object 1 ICP 2 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 2 Handle**
Rn.5	31:16	**Object 1 ICP 1 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 1 Handle**
Rn.4	31:16	**Object 1 ICP 0 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 0 Handle**
Rn.3	31:16	**Object 0 ICP 3 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 3 Handle**
Rn.2	31:16	**Object 0 ICP 2 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 2 Handle**
Rn.1	31:16	**Object 0 ICP 1 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 1 Handle**
Rn.0	31:16	**Object 0 ICP 0 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 0 Handle**

GRF DWord	Bit De	scription
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled and ICP Count>4		
Rn+1.7	31:16	**Object 1 ICP 7 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 7 Handle**
Rn+1.6	31:16	**Object 1 ICP 6 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 6 Handle**
Rn+1.5	31:16	**Object 1 ICP 5 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 5 Handle**
Rn+1.4	31:16	**Object 1 ICP 4 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 4 Handle**
Rn+1.3	31:16	**Object 0 ICP 7 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 7 Handle**
Rn+1.2	31:16	**Object 0 ICP 6 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 6 Handle**
Rn+1.1	31:16	**Object 0 ICP 5 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 5 Handle**
Rn+1.0	31:16	**Object 0 ICP 4 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 4 Handle**
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled and ICP Count>7		
Rn+2.7	31:16	**Object 1 ICP 11 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 11 Handle**
Rn+2.6	31:16	**Object 1 ICP 10 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 10 Handle**
Rn+2.5	31:16	**Object 1 ICP 9 Handle ID**

GRF DWord	Bit De	scription
	15:12	Reserved
	11:0	**Object 1 ICP 9 Handle**
Rn+2.4	31:16	**Object 1 ICP 8 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 8 Handle**
Rn+2.3	31:16	**Object 0 ICP 11 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 11 Handle**
Rn+2.2	31:16	**Object 0 ICP 10 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 10 Handle**
Rn+2.1	31:16	**Object 0 ICP 9 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 9 Handle**
Rn+2.0	31:16	**Object 0 ICP 8 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 8 Handle**
The following register is included only if DUAL_OBJECT mode and Include Vertex Handles is enabled and ICP Count>12		
Rn+3.7	31:16	**Object 1 ICP 15 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 15 Handle**
Rn+3.6	31:16	**Object 1 ICP 14 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 14 Handle**
Rn+3.5	31:16	**Object 1 ICP 13 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 13 Handle**
Rn+3.4	31:16	**Object 1 ICP 12 Handle ID**
	15:12	Reserved
	11:0	**Object 1 ICP 12 Handle**
Rn+3.3	31:16	**Object 0 ICP 15 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 15 Handle**

GRF DWord	Bit De	scription
Rn+3.2	31:16	**Object 0 ICP 14 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 14 Handle**
Rn+3.1	31:16	**Object 0 ICP 13 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 13 Handle**
Rn+3.0	31:16	**Object 0 ICP 12 Handle ID**
	15:12	Reserved
	11:0	**Object 0 ICP 12 Handle**
[Varies] optional	31:0	**Constant Data (optional) :** Some amount of constant data (possible none) can be extracted from the push constant buffer (PCB) and passed to the thread following the R0 Header. The amount of data provided is defined by the sum of the read lengths in the last 3DSTATE_CONSTANT_GS command (taking the buffer enables into account). The Constant Data arrives in a non-interleaved format.
Varies	31:0	**Pushed Vertex Data.** There can be up to 32 vertices supplied, each with a size defined by the **Vertex URB Entry Read Length** state. The amount of data provided for each vertex is defined by the **Vertex URB Entry Read Length** state For SINGLE or DUAL_INSTANCE dispatch modes, the pushed data for Vertex 0 immediately follows any pushed constant data. The pushed data for Vertex 1 immediately follows Vertex 0, and so on. There is no upper/lower swizzling of data. For DUAL_OBJECT dispatch mode, the pushed vertex data is split into upper and lower halves with Object 0 input vertices in the lower half, and Object 1 input vertices in the upper half.

5.5 GS Thread Execution

A GS thread is capable of performing arbitrary algorithms given the thread payload (especially vertex) data and associated data structures (binding tables, sampler state, etc.) as input. Output can take the form of vertices output to the FF pipeline (at the GS unit) and/or data written to memory buffers via the DataPort.

The primary usage models for GS threads include (possible combinations of):

- Compiled application-provided “GS shader” programs, specifying an algorithm to convert the vertices of an input object into some output primitives. For example, a GS shader may convert lines of a line strip into polygons representing a corresponding segment of a blade of grass centered on the line. Or it could use adjacency information to detect silhouette edges of triangles and output polygons extruding out from the those edges. Or it could output absolutely nothing, effectively terminating the pipeline at the GS stage.
- Driver-generated instructions used to write pre-clipped vertices into memory buffers (see Stream Output below). This may be required whether or not an app-provided GS shader is enabled.

- Driver-generated instructions used to emulate API functions not supported by specialized hardware. These functions might include (but are not limited to):
    - Conversion of API-defined topologies into topologies that can be rendered (e.g., LINELOOP→LINESTRIP, POLYGON→TRIFAN, QUADs→TRIFAN, etc.)
    - Emulation of "Polygon Fill Mode", where incoming polygons can be converted to points, lines (wireframe), or solid objects.
    - Emulation of wide/sprite points.
- Things best left to the imagination.

[**DevILK+**]: When rendering is required, concurrent GS threads must use the FF_SYNC message (URB shared function) to request an initial VUE handle and synchronize output of VUEs to the pipeline (see *URB* in *Shared Functions*). Only one GS thread can be outputting VUEs to the pipeline at a time. In order to achieve parallelism, GS threads should perform the GS shader algorithm (along with any other required functions) and buffer results (either in the GRF or scratch memory) before issuing the FF_SYNC message. The issuing GS thread will be stalled on the FF_SYNC writeback until it is that thread's turn to output VUEs. As only one GS thread at a time can output VUEs, the post-FF_SYNC output portion of the kernel should be optimized as much as possible to maximize parallelism.

5.5.1 GS Shader Programming Notes [DevILK-A]

At least one handle needs to be allocated per thread

UEM path is used in readback so,
1. EOT with FFSYNC **cannot** be used – Handleid is required for registering EOT in uem.
2. EOT with DAP WR **cannot** be used – Handleid is required for registering EOT in uem.
3. EOT with NULL SFID **cannot** be used – Handleid is required for registering EOT in uem.
4. EOT **can only** be sent through URBWRITE with complete bit set – required in uem.

5.5.2 GS Shader Programming Notes [DevILK-B]

When rendering is enabled:
1. At least one handle needs to be requested by the kernal.
2. Enabled Hint should be **set** in GS_STATE.
3. EOT with FFSYNC **cannot** be used.
4. EOT with DAP WR **cannot** be used.
5. EOT with NULL SFID **cannot** be used
6. EOT **can only** be sent through URBWRITE with complete bit set.

When rendering is disabled:
1. **NO** handles should be allocated.
2. Rendering Enabled Hint should be **reset** in GS_STATE**.**
3. FFSYNC with EOT **can** be used.
4. DAP WR with EOT **can** be used.
5. EOT with URBWRITE **can** be used.
6. EOT with SFID Null messages **can** be used if required.

5.5.3 Vertex Output

The GS kernel will typically use the URB_WRITE message to output vertices and request additional handles. (Refer to the *3D Pipeline* chapter for a general discussion of how FF units output vertices, and the *URB* chapter for details on the use of the URB_WRITE message.)

The following table lists which primitive topology types are valid for output by a GS thread.

LINELIST	**Yes**
LINELIST_ADJ	**No**
LINESTRIP	**Yes**
LINESTRIP_ADJ	**No**
LINESTRIP_BF	**Yes**
LINESTRIP_CONT	**Yes**
LINESTRIP_CONT_BF	**Yes**
LINELOOP	**No**
POINTLIST	**Yes**
POINTLIST_BF	**Yes**
POLYGON	**Yes**
QUADLIST	**No**
QUADSTRIP	**No**
RECTLIST	**Yes**
TRIFAN	**Yes**
TRIFAN_NOSTIPPLE	**Yes**
TRILIST	**Yes**
TRILIST_ADJ	**No**
TRISTRIP	**Yes**
TRISTRIP_ADJ	**No**
TRISTRIP_REV	**Yes**

The GS thread is responsible for providing correct PrimType, PrimStart and PrimEnd information for each vertex output, in the same fashion as the Vertex Fetch unit. Given that the GS thread is likely performing an algorithm as specified by an application “geometry shader” program, where the algorithm dictates when and if a vertex is to be output, the GS thread is allowed to output incomplete primitives (too few or too many vertices). The downstream FF units will correctly handle any dangling vertices.

However, the PrimStart and PrimEnd indicators must be correct for all vertices, e.g., the last vertex of a topology must have PrimEnd set. This may require the GS thread to postpone completion of a vertex output operation until either the next vertex is encountered or the algorithm (not the thread) completes.

Note that, through use (clearing) of the **Complete** bit in the URB_WRITE message, is it possible to write a vertex to the URB yet delay the "complete" indication until later. The PrimType, PrimStart, and PrimEnd indications are not sampled by the FF pipeline until **Complete** is set. This relieves the GS thread from actually having to buffer the pending vertex.

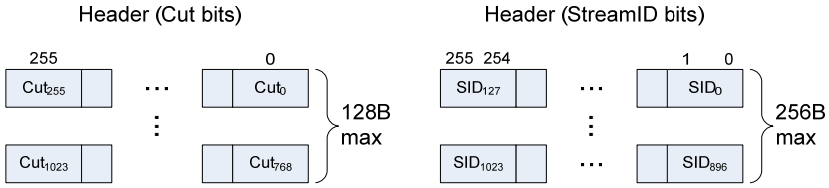
A GS or CLIP thread is restricted as to the number of URB handles it can retain. Here a "retained" handle refers to a URB handle that (a) has been pre-allocated or allocated and returned to the thread via the **Allocate** bit in the URB_WRITE message, and (b) has yet to be returned to the pipeline via the **Complete** bit in the URB_WRITE message.

- [**Pre-DevILK**]: When operating in single-thread mode (**Maximum Number of Threads** == 1), the number of retained handles must not exceed min(16, **Number of URB Entries**).
- [**Pre-DevILK**]: When operating in dual-thread mode (**Maximum Number of Threads** == 2), the number of retained handles must not exceed (**Number of URB Entries**/2).
- [**DevILK+**]: The number of retained handles must not exceed min(32, **Number of URB Entries**).

This restriction is not expected to be significant in that most/all GS/CLIP threads are expected to retain only a few (<=4) handles.

5.5.3.1 GS URB Entry

All outputs of a GS thread will be stored in the single GS thread output URB entry. Cut (1 bit/vertex) or StreamID (2 bits/vertex) bits are packed into an optional 1-8 32B header. The **Control Data Format** and **Control Data Header Size** states are used to specify the size and contents of the header data (if any).



Following the optional header is a variable number of 16B or 32B-aligned/granular vertices:

- When rendering is DISABLED, typically output vertices are 32B-aligned, with the exception of 16B-alignment for vertices <= 16B in length.
    - The absolute worst case size comes from three DW scalars output per vertex. If these are, say, three ".x" outputs, you need to store each DW in a 128b (16B) element, plus another pad 16B to keep the 32B alignment. So you require 4*16B = 64B/vertex. You have to have room for 1024 scalars / 3 scalar/vtx = 341 vertices. 341*64B = 21,824B. Then add 96B to hold 2b/vtx streamID and you get 21,920B entries.
- When rendering is ENABLED, each output vertex is 32B-aligned. Here the vertex header and vertex 'position' is required and therefore the minimum size vertex is 32B.
    - Here the worst case size isn't as bad as render-disabled, as you have to have a 4DW position output, plus any additional output. So, say you output 5 DW per vertex. You need 64B/vertex (16B vtx header, 16B position, 16B for the 2nd element, and 16B of pad). You have to have room for 1024 scalars / 5 = 204 vertices. 204*64 = 13,056B. Then add 64B to hold 2b/vtx streamID and you get 13,120B entries.

The size of the URB entry should be based on the declared maximum # of output vertices and the declared output vertex size (the union of per-stream vertex structures, if required).

5.5.3.2 GS Output Topologies

The following table lists which primitive topology types are valid for output by a GS thread.

PrimTopologyType	Supported for GS Thread Output?
LINELIST	Yes
LINELIST_ADJ	No
LINESTRIP	Yes
LINESTRIP_ADJ	No
LINESTRIP_BF	Yes
LINESTRIP_CONT	Yes
LINESTRIP_CONT_BF	Yes
LINELOOP	No
POINTLIST	Yes
POINTLIST_BF	Yes
POLYGON	Yes
QUADLIST	No
QUADSTRIP	No
RECTLIST	Yes
TRIFAN	Yes
TRIFAN_NOSTIPPLE	Yes
TRILIST	Yes
TRILIST_ADJ	No
TRISTRIP	Yes
TRISTRIP_ADJ	No
TRISTRIP_REV	Yes
PATCHLIST_xxx	Yes

5.5.3.3 GS Output StreamID

When the **GS Enable** is DISABLED, output vertices will be assigned a StreamID = 0;

When the **GS Enable** is ENABLED, output vertices will be assigned a StreamID = **Default StreamID** under the following conditions:

- **Control Data Format** = 0, or
- **Control Data Format** > 0 and **Control Data Format** = GSCTL_CUT

When the GS is enabled, **Control Data Format** > 0 and **Control Data Format** = GSCTL_SID, output vertices will be assigned a StreamID as programmed in the Control Data output by the thread.

5.5.4 Stream Output

With a “Stream Output” function, vertex data can be written to one or more memory buffers for subsequent readback by the CPU or use in subsequent Draw operations. The Stream Output function is defined such that the pipeline is tapped immediately following the GS stage (just prior to clipping) and in such a way that permits the GS kernel to perform the writes after the GS shader function.

The final contents of Stream Output buffers must follow the strict pipeline ordering of vertices. Given this ordering requirement, it will be necessary to run the GS stage in a single-threaded fashion (**Maximum Number of Threads** == 1). Otherwise concurrent GS threads might append vertices to the output buffer out of order.

Hardware support for the Stream Output is limited to a special “Streamed Vertex Buffer Write” DataPort message. (Refer to *DataPort* chapter). Through use of this message type, the GS thread can write from 1 to 4 DWords to specified ‘element’ (indexed entry) in a BUFFER surface. The DataPort will inhibit writes past the end of the buffer.

Stream Output is allowed to either a set (<= 4) of “single element buffers” (SEBs) or a single “multiple element buffer” (MEB). The SEB is a simple 1D array of 1-4 DWord elements, while the MEB is a 1D array of structures, with a maximum structure pitch of 2K bytes. Up to 16 1-4 DWord elements within the MEB structure can be written, with arbitrary, multiple-DWord “gaps” that must be left unmodified in memory.

Software will likely need to define separate surface states for each SEB, and separate surface states for each element within the MEB structure. The surfaces are selected via the normal binding table mechanisms.

The need for separate SEB surface states is obvious, as the SEBs are separate buffers in memory. The MEB surface-per-element allows the GS kernel to address the MEB using an structure index. Here each surface would be specified as having the same structure pitch, but with different starting addresses corresponding to the different element offsets within the structure – in effect, defining a set of interleaved surfaces. The GS kernel would output one write message per element.

(Note that software could, if it wished, treat the MEB as a single 1D array of DWords, though it would then have to write the buffer one DWord at a time, performing the address calculations within the GS kernel. This should not be necessary, and is certainly not recommended due to obvious performance and complexity reasons.)

Programming Note: If the GS stage is enabled, software must always allocate at least one GS URB Entry. This is true even if the GS thread never needs to output vertices to the pipeline, e.g., when only performing stream output. This is an artifact of the need to pass the GS thread an initial destination URB handle.

5.5.4.1 Streamed Vertex Buffer Indexing [DevBW,DevCL]

To perform the Stream Output function, the GS kernel will need to manage a write offset associated with the output buffer(s). As this offset must persist between GS kernel invocations, it will need to be reside in memory. Software will likely need to define a separate surface to maintain this offset, and any other variables required to support the Stream Output function. There is no special hardware support for this functionality, and therefore software will need to rely on existing, generic memory read/write functions provided by the GEN4 subsystem.

5.5.4.2 Streamed Vertex Buffer Indexing [DevCTG+]

The GS unit supports four Streamed Vertex Buffer Indicies (SVBIs) in hardware. Only when the **Streamed Vertex Buffer Enable** bit (GS_STATE) is set will the current SVBI values be passed to GS threads via R1 of the thread payload. The GS thread is then responsible for (a) using/incrementing these initial values when generating the **Destination Index** field of DataPort Streamed Vertex Buffer Write messages – as the DataPort will this field and not the SVBIs directly to write out vertex data, and (b) correctly programming the Increment SVBIs bit of the DataPort Streamed Vertex Buffer Write message in order to cause the GS's SVBI values to increment as required. The incremented SVBI values will be passed to the next GS thread unless they are reloaded from the command stream.

The SVBIs can be loaded (either directly or indirectly from memory) via the new 3DSTATE_GS_SVB_INDEX command. Software would use this command to specify initial values when an SVB was bound to the pipeline.

5.5.5 Thread Termination

GS threads must terminate by sending a URB_WRITE message with the **EOT** and **Complete** bits set. The Used bit can be set (if outputting a VUE) or clear (if freeing an used VUE).

5.6 Vertex Header Readback

The GS unit performs a readback of the Vertex Header of each vertex exiting the GS stage (either passed through or generated by a GS thread) as this information is required by the next FF stage (CLIP). Software is responsible for ensuring that any required Vertex Header fields are valid at this point in the pipeline. See *Vertex Data Overview* for a description of the Vertex Header fields and how they are read-back and used by the GS unit.

5.7 Primitive Output

(This section refers to output from the GS unit to the pipeline, not output from the GS thread)

The GS unit will output primitives (either passed-through or generated by a GS thread) in the proper order. This includes the buffering of a concurrent GS thread's output until the preceding GS thread terminates. Note that the requirement to buffer subsequent GS thread output until the preceding GS thread terminates has ramifications on determining the number of VUEs allocated to the GS unit and the number of concurrent GS threads allowed.

5.8 Other Functionality

5.8.1 Statistics Gathering

There are a number of GS/StreamOutput pipeline statistics counters associated with the GS stage and GS threads. This subsection describes these counters and controls depending on device, even in the cases where functions outside of the GS stage (e.g., DataPort) are involved in the statistics gathering.

Refer to the *Statistics Gathering* summary provided earlier in this specification. Refer to the *Memory Interface Registers* chapter for details on these MMIO pipeline statistics counter registers, as well as the chapters corresponding to the other functions involved (e.g., DataPort, URB shared functions).

5.8.1.1 GS Invocations

The GS unit controls the GS_INVOCATIONS counter, which the number of times a GS thread is executed. A GS thread is executed for each object (triangle, line or point) that is derived from the stream of incoming primitive topologies. If the **Statistics Enable** bit in GS_STATE is set, the GS unit will increment the GS_INVOCATIONS_COUNT register (see Memory Interface Registers in Volume Ia, *GPU*) for each object that is dispatched to a GS thread.

5.8.1.2 GS Primitives Output [Pre-DevILK]

The GS_PRIMITIVES_COUNT pipeline statistics register counts objects (triangles/lines/points) output by GS threads.

5.8.1.2.1 Pre-Dev ILK

The GS_PRIMITIVES_COUNT is actually tracked by the CLIP stage on behalf of the GS stage. This statistic has an enable bit (**GS Output Object Statistic Enable**) in CLIP_STATE separate from the CLIP stage's **Statistics Enable**. In order to provide consistent statistics reporting at the API level, **GS Output Object Statistic Enable** in CLIP_STATE should always be set and cleared in lock-step with **Statistics Enable** in GS_STATE. Likewise, the CLIP stage should not be put in pass-through mode when the GS stage is enabled with its **Statistics Enable** set. See the *Clipper* chapter for more details.

5.8.1.2.2 Dev ILK+

As a effect of GS threads issuing FF_SYNC messages to the URB shared function, the GS_PRIMITIVES_COUNT register is incremented by the **NumGSPrimsGenerated** field of that message.

5.8.1.3 Stream Output Primitives Written [DevILK+]

5.8.1.3.1 Dev CTG

Whenever a GS thread outputs a DataPort Streamed Vertex Buffer Write (SVBWrite) message with the **Increment Num Prims Written** bit set, the SO_NUM_PRIMS_WRITTEN register will be incremented. The **Statistics Enable** bit in GS_STATE does not affect the increment of this register.

Programming Note: The GS thread is solely responsible for limiting the increment of SO_NUM_PRIMS_WRITTEN in the face of SVB buffer overflow. There is no hardware performing this function.

5.8.1.3.2 Dev ILK+

GS threads must terminate by issuing a URBWrite message with EOT set. The URBWrite header contains an SONumPrimsWritten Increment Count

Whenever a GS thread outputs a DataPort Streamed Vertex Buffer Write (SVBWrite) message with the **Increment Num Prims Written** bit set, the SO_NUM_PRIMS_WRITTEN register will be incremented. The **Statistics Enable** bit in GS_STATE does not affect the increment of this register. **Programming Note:** The GS thread is solely responsible for limiting the increment of SO_NUM_PRIMS_WRITTEN in the face of SVB buffer overflow. There is no hardware performing this function.

5.8.1.3.3 Dev ILK+

GS threads must terminate by issuing a URBWrite message with EOT set. The URBWrite header contains an SONumPrimsWritten Increment Count

Whenever a GS thread outputs a DataPort Streamed Vertex Buffer Write (SVBWrite) message with the **Increment Num Prims Written** bit set, the SO_NUM_PRIMS_WRITTEN register will be incremented. The **Statistics Enable** bit in GS_STATE does not affect the increment of this register.

Programming Note: The GS thread is solely responsible for limiting the increment of SO_NUM_PRIMS_WRITTEN in the face of SVB buffer overflow. There is no hardware performing this function.

5.8.1.4 Stream Output Primitive Storage Needed [DevCTG+]

Whenever a GS thread outputs a DataPort Streamed Vertex Buffer Write (SVBWrite) message with the **Increment Prim Storage Needed** bit set, the SO_NUM_PRIM_STORAGE_NEEDED register will be incremented. The **Statistics Enable** bit in GS_STATE does not affect the increment of this register.

Programming Note: There should be no need for GS threads to limit the increment of SO_PRIM_STORAGE_NEEDED, as this value should reflect the minimum buffer size required to avoid overflow.

6. Clip Stage

6.1 CLIP Stage Overview

The CLIP stage of the GENx 3D Pipeline is similar to the GS stage in that it can be used to perform general processing on incoming 3D objects via spawned GENx threads. However, the CLIP stage also includes specialized logic to perform a *ClipTest* function on incoming objects. These two usage models of the CLIP stage are outlined below.

Refer to the *Common 3D FF Unit Functions* subsection in the *3D Overview* chapter for a general description of a 3D Pipeline stage, as much of the CLIP stage operation and control falls under these "common" functions. I.e., many of the CLIP stage state variables and CLIP thread payload parameters are described in *3D Overview*, and although they are listed here for completeness, that chapter provides the detailed description of the associated functions.

Refer to this chapter for an overall description of the CLIP stage, details on the ClipTest function, and any exceptions the CLIP stage exhibits with respect to common FF unit functions.

6.1.1 Clip Stage – General-Purpose Processing

Numerous state variable controls are provided to tailor the ClipTest function as required by the API or primitive characteristics. These controls allow a mode where all objects are passed to CLIP threads, and in this regard the CLIP stage can be used as a second GS stage. However, unlike the GS stage, primitives output by CLIP threads will not be subject to 3D Clipping, and therefore any clip-testing/clipping of these primitives (if required) would need to be performed by the CLIP thread itself.

6.1.2 Clip Stage – 3D Clipping

The ClipTest fixed function is provided to optimize the CLIP stage for support of generalized *3D Clipping*. The CLIP FF unit examines the position of incoming vertices, performs a fixed function *VertexClipTest* on these positions, and then examines the results for the vertices of each independent object in *ClipDetermination*.

The results of ClipDetermination indicate whether an object is to be processed by a thread (MustClip), discarded (TrivialReject) or passed down the pipeline unmodified (TrivialAccept). In the MustClip case, the spawned thread is responsible for performing the actual 3D Clipping algorithm. The CLIP thread is passed the source object vertex data and is able to output a new, arbitrary 3D primitive (e.g., the clipped primitive), or no output at all. Note that the output primitive is independent in that it is comprised of newly-generated VUEs, and does not share vertices with the source primitive or other CLIP-generated primitives.

New vertices produced by the CLIP threads are stored in the URB. Their Vertex Headers are then read from the VUEs in order to insert the relevant information into the 3D pipeline. The CLIP unit maintains the proper ordering of CLIP-generated primitives and any surrounding trivially-accepted primitives. The CLIP unit also supports multiple concurrent CLIP threads and maintains the proper ordering of the thread outputs as dictated by the order of the source objects.

The outgoing primitive stream is sent down the pipeline to the Strip/Fan (SF) FF stage (now including the read-back VUE Vertex Header data such as Vertex Rosition (NDC or screen space), RTAIndex, VPIndex, PointWidth) and control information (PrimType, PrimStart, PrimEnd) while the remainder of the vertex data remains in the VUE in the URB.

6.1.3 [Dev ILK] Fixed Function Clipper

[**DevILK+**] Havendale/Auburndale onwards the device supports Fixed Function Clippling. Prior to this fixed function pipeline had Clipping done in the EU. However the clipper thread latency was high and caused a bottleneck in the pipeline. Hence the motivation for a fixed function clipper.

[**DevILK**] Most cases the fixed function clipper will do the clipping. However in the following cases the clip kernel must be invoked by programming the "force kernel clip mode" bit in the CL State.

- **Point and Wireframe Fill Modes:** The clip kernel is currently enabled to emulate point and wireframe fill modes, converting input triangle geometry (list, strip, fan, or polygon) to a line list or point list. This is required for both HWVP and SWVP.

- **Non-Perspective Barycentric:** If there is a need for Non-Perspective Barycentric parameter interpolation, the state bit to launch a clip thread is set.

6.2 Concepts

This section provides an overview of 3D clip-testing and clipping concepts, as defined by the OpenGL APIs. It is provided as background material: some of the concepts impact HW functionality while others impact CLIP kernel functionality.

6.2.1 The Clip Volume

3D objects are optionally clipped to the *clip volume*. The clip volume is defined as the intersection of a set of *clip half-spaces*. Six of these half-spaces define the view volume, while additional, user-defined half-spaces can be employed to perform clipping (or at least culling) within the view volume.

The CLIP stage design will permit the enable/disable of certain subsets of these clip half-spaces. This capability can be used, for example, to disable viewport, guardband, and near and far clipping as required by the API and other conditions.

6.2.1.1 View Volume

The intersection of the six view half-spaces defines the *view volume*. The view volume is defined in 4D clip space coordinates as:

View Clip Plane	'Outside' Condition	
	4D Clip Space	NDC space, positive w
XMIN (NDC Left)	**clip.x < -clip.w**	**ndc.x < -1**
XMAX (NDC Right)	**clip.w < clip.x**	**ndc.x > 1**
YMIN (NDC Bottom)	**clip.y < -clip.w**	**ndc.y < -1**
YMAX (NDC top)	**clip.w < clip.y**	**ndc.y > 1**
ZMIN (NDC Near)	**OGL: clip.z < -clip.w**	**OGL: ndc.z < -1.0**
ZMAX (NDC Far)	**clip.w < clip.z**	**ndc.z > 1.0**

Note that, since the 2D (X,Y) extent of the projected view volume is subsequently mapped to the 2D pixel space viewport, the terms "viewport" and "view volume" are used somewhat interchangeably in this discussion.

The CLIP unit will perform view volume clip test using NDC coordinates (the results of the speculative PerspectiveDivide). The treatment of negative ndc.w and invalid (NaN, +/-INF) coordinates is clarified below.

Negative W Coordinates

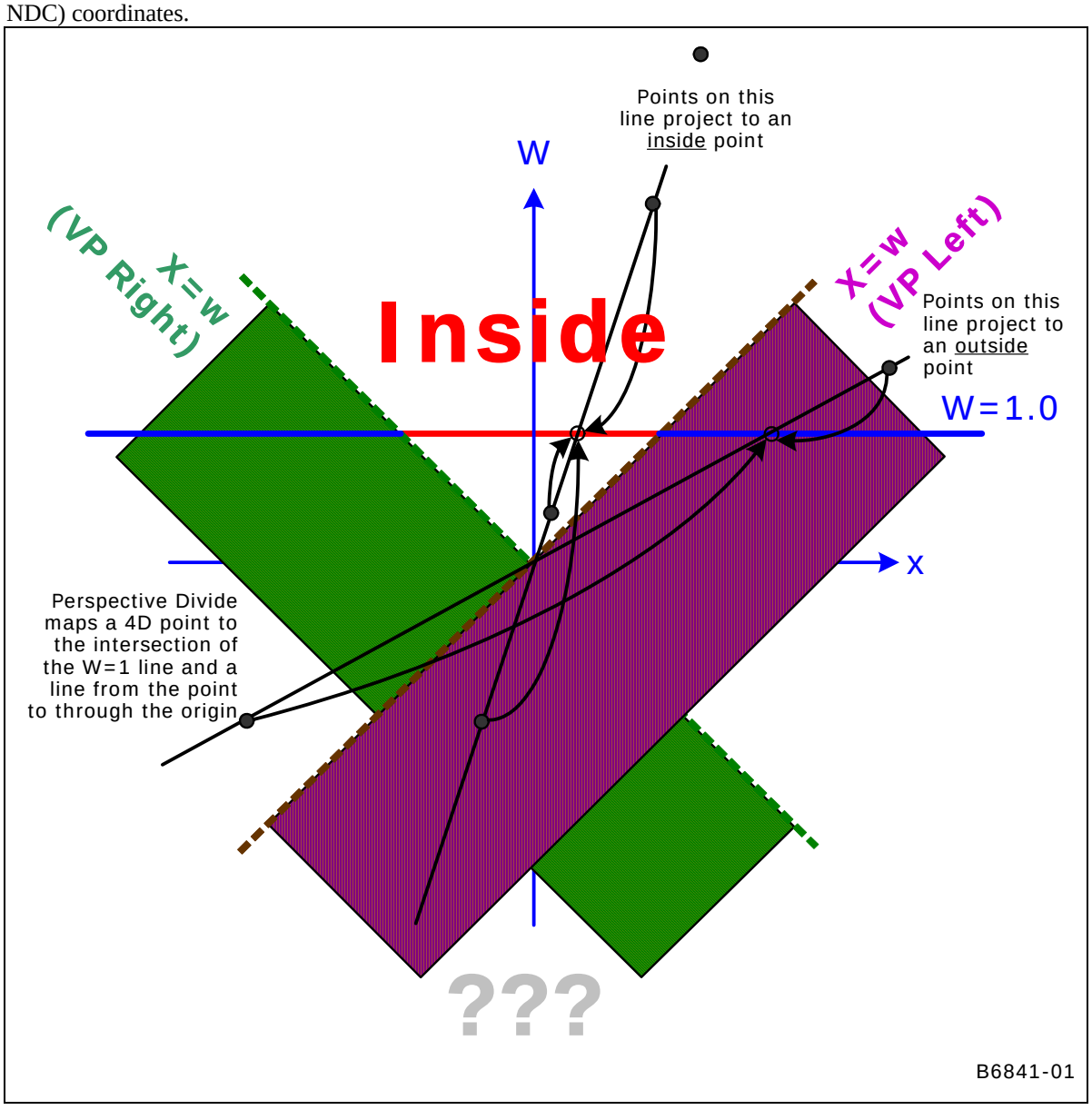
Consider for a moment vertices with a negative clip.w coordinate. Examination of the API definitions for "outside" shows that it is impossible for that vertex to be considered inside both the XMIN (NDC Left) and XMAX (NDC Right) planes. The clip.x coordinate would need to be greater than or equal to some positive value (-clip.w) to be considered inside the XMIN plane, while also being less than or equal to the negative (clip.w) value to be considered inside the XMAX plane. Obviously both these conditions cannot be met simultaneously, so a vertex with a negative clip.w coordinate will always appear outside.

Surprisingly, it is possible for a vertex to be outside both the XMIN and XMAX planes (and likewise for the Y axis). This arises when clip.w is negative and clip.x falls between clip.w and -clip.w. Note, however, that in NDC space (post perspective-divide), this same vertex would be considered inside. This disparity arises from the loss of information from the perspective divide operation, specifically the signs of the input operands. The CLIP stage will avoid this artifact by supporting an additional clip.w=0 clip plane – a negative ndc.rhw value indicates the point is outside of the clip.w=0 plane. (See sections below for related errata in DevBW and DevCL devices)

The assumption made in the Clip stage is that only the w>0 portion of clip space is considered visible. The VertexClipTest function tests each incoming 1/w value and, if negative, the vertex is tagged as being outside the w=0 plane. These vertex outcodes are combined in ClipDetermination to determine TA/TR/MC status.

A negative w coordinate poses an additional issue due to the fact that VertexClipTest is performed using post-perspection-projection coordinates (NDC or screen space). This disparity arises from the loss of information from the perspective divide operation, specifically the signs of the input operands. For example, to test for (x>w) using NDC coordinates, (x/w>1) must be used when w>0, and (x/w<1) must be used when w<0. The VertexClipTest function therefore uses the sign of the incoming 1/w coordinate to select the appropriate comparison function for each of the VP and GB clip planes.

As the CLIP thread performs clipping in 4D clip space, only the truly visible portions of objects (i.e, meeting the 4D clip space visibility criteria) will be considered. The CLIP thread should not output negative w (clip or NDC) coordinates.



6.2.2 User-Specified Clipping

The various APIs define mechanisms by which objects can be clipped or culled according to some user-specified parameter(s) in addition to the implied viewport clipping. In GENx, the HW support of these mechanisms is restricted to use of the 8 UserClipFlags (UCFs) of the VUE Vertex Header. Software is required to provide the remaining support (e.g., the JITTER including GENx instructions to cause a distance value to be computed, tested for visibility, and generation of the appropriate UCF bit.)

6.2.2.1 User Clip Planes (OGL)

In OpenGL APIs, up to 6 *user clip planes* can be defined and enabled. These planes define half-spaces that are intersected with the view volume (and each other) to form a final clip volume. Each user clip plane is specified by four coefficients of a plane equation in clip space coordinates (`UserClipPlane[n].xyzw`). A point is not visible if it has a negative distance to the plane. Therefore, points P that satisfy the following equation are considered to lie in the half-space and therefore may be visible:

$$(P.xyzw \text{ dot } UCP[n].xyzw) >= 0, \qquad 0<=n<=5$$

There is no direct HW support for this distance computation. The driver/JITTER is required to cause the distances to be correctly computed/compared in a shader, with the comparison result (boolean) placed in the proper location in the Vertex Header.

6.2.3 Negative-W Clipping Errata

In DevBW and DevCL-A devices there is a bug in the definition of the handling of negative RHW (1/w) coordinates in the Clip unit's trivial reject logic. The fault may cause line and triangle objects to be erroneously trivially rejected and therefore be manifested as occasional missing geometry.

This section defines a correction of the problem in DevCTG+. This section also describes a partial fix (ECO) that is incorporated into DevCL-B, and an additional ECO HW change for DevBW-E0.

DevCL-B ECO (partial fix)

The DevCL-B ECO parallels the PreDevBW-E0,DevCL-A SW workaround in that it uses UC7 logic to provide full trivial-accept (TA), trivial-reject (TR) and mustclip (MC) support for the w=0 clip plane . The pre-clipper shader kernel will have to be modified to set NDC x/w, y/w, z/w to 0.0 if w<0. However, this ECO allows all 8 UserClipFlags to be supported (with limitations).

Note that this ECO is suboptimal due to constraints on the location and extent of the ECO.

DevBW-E0 ECO (partial fix)

This ECO extends the DevCL-B ECO described above. In VertexClipTest, if the vertex has a negative W coordinate, the VP & GB outcodes are inverted (if enabled). (In addition, a bug related to mis-handling of z = -0 is resolved, but that is unrelated to neg-w handling). Note that the inversion of the outcodes is not entirely correct in that it mis-handles the '=' condition. As a result, the clip boundaries will be treated as "outside" in the negative-w regions. (Unfortunately correct handling the '=' case made the ECO untenable).

On the bright side, this ECO:

- Removes the need for any VS/software workaround. The HW will detect a negative w and compute the (almost-correct) VP & GB outcodes.

- Removes the need to set UserClipFlagsMustClipEnable. There is no reason to force a clip thread specifically for UC7 (which is set if w<0). As the outcodes are set correctly even when w<0, clip threads will be spawned as required. In addition, objects completely in w<0 space will be correctly TR'd against UC7 (assuming that UC is enabled).

However, UC7 will still be routed to the BAD outcode and subsequently will cause a clip thread to be spawned – therefore spawning clip threads for objects with any vertex having UC7 set.

DevCTG+ Fix:

A new "NEGW" vertex outcode is added. It is set for a vertex if the RHW component is negative. Also invert the computed VP,GB vertex outcodes if NEGW is seen. In ClipDetermination, NEGW is treated like a separate clip plane in determination of trivial accept, trivial reject and mustclip cases.

The following table summarizes the software workarounds required for the various devices

Figure 6-1 SW Workaround Summary

Device	VS/Kernel	Clip State	Clip Kernel	Notes
Pre-DevBW-E **DevCL-A**	**If (w<0) {** **npc.xyzw=0** **UC7=1** **}**	**Enable UC7** **Set UCFMustClipEnable to force clips for mixed NEGW cases.**	**If UC7 set, other outcodes are undefined and must be recomputed.**	**UC7 unavailable for normal use**
DevCL-B+	**If (w<0) {** **npc.xyz=0** **}**	**Enable UC7** **Set UCFMustClipEnable to force clips for mixed NEGW cases.**	**If UC7 set, other outcodes are undefined and must be recomputed.** **Will see "BAD" objects due to BAD←UC7 hack.**	**UC7 is supported (routed to BAD before being used for NEGW).**
DevBW-E0+	**No WA required**	**Enable UC7 in order to allow TR against w<0.** **No need to set UCFMustClipEnable.**	**Will see "BAD" objects due to BAD←UC7 hack.**	**UC7 is supported (routed to BAD before being used for NEGW).**
DevCTG+ No	**WA required**	**None, other than enabling NEGW clip**	**None.**	

6.2.3.1 W Clipping Errata (DevBW,DevCL-A)

The DevBW and DevCL-A devices contain a definitional error in that a separate clip.w=0 clip plane was not implemented, and instead a negative ndc.rhw value caused all clip outcodes (except for BAD and UCs) to get set in VertexClipTest. This behavior can lead to false trivial rejects for line and triangle objects. The Trivial Reject function is therefore UNDEFINED under the following conditions:

1. Line or triangle object
2. At least one vertex has a negative RHW component
3. At least one vertex has a non-negative RHW component
4. All vertices straddle one or more common VP/GB clip planes
5. All vertices are not outside of a common enabled clip plane (including UCFs) – i.e., the object should not be trivially rejected

Software must prevent these conditions from occuring whenever it uses a Clip Mode which uses the trivial reject function (NORMAL or CLIP_NON_REJECTED). A suggested workaround is to have the previous shader detect negative w coordinates, and if seen, set all NDC coordinates (x/w, y/w, z/w, 1/w) in the Vertex Header with 0.0, and set/utilize a UserClipFlag to cliptest against w=0.

6.2.3.2 W Clipping Errata (DevCL-B)

The DevCL-B device includes a partial fix for the errata (previous section). The fix parallels the suggested Dev-BW,Dev-CL-A SW workaround in that it uses UC7 logic to provide full trivial-accept (TA), trivial-reject (TR) and mustclip (MC) support for the w=0 clip plane (thus correctly handling negative RHW components). The pre-clipper shader kernel will have to be modified to set NDC x/w, y/w, z/w to 0.0 if w<0. However, this ECO allows all 8 UserClipFlags to be supported (with limitations).

The fix consists of 4 parts:

(1) Reroute UserClipFlag[7] into BAD

In VertexClipTest, instead of

```
outcode[BAD] = ISNAN(rhw)
```

the fix adds

```
outcode[BAD] = ISNAN(rhw) || UserClipFlag[7]
```

In concert with (4) (BAD Forces SPAWN, below), this change will force SPAWN whenever a vertex has rhw==NaN or has UserClipFlag[7] set, assuming REJECT_ALL mode is not in effect. Previously, only rhw==NAN lead to a BAD object, and all BAD objects were discarded except in CLIP_ALL mode.

(2) Prevent Setting of All Outcodes upon Negative RHW

In VertexClipTest, the fix removes the following logic (which was the source of the original problem):

```
if (0 rhw_neg)
{
    outCode[VP_XMIN] = 1
    outCode[VP_XMAX] = 1
    outCode[VP_YMIN] = 1
    outCode[VP_YMAX] = 1
    outCode[VP_ZMIN] = 1
    outCode[VP_ZMAX] = 1
    outCode[GB_XMIN] = 1
    outCode[GB_XMAX] = 1
    outCode[GB_YMIN] = 1
    outCode[GB_YMAX] = 1
    goto UserClipFlags
}
```

This change prevents some false trivial rejects. However, it is not a complete fix in that the computed VP,GB outcodes are still not correct when w<0. In order to completely remove false TRs, the pre-clipper kernel must set x/w, y/w and z/w (the NDC coordinates in the vertex header) to 0.0 whenever w<0. Note that 1/w must be passed normally (not forced to 0.0 as in the DevBW,DevCL-A workaround) – as the sign of 1/w is used to set UC7 (see below).

(3) Reroute rhw_neg into UCF7

In VertexClipTest, instead of

```
outcode[UC7] = UserClipFlag[7] && UserClipFlagClipTestEnable[7]
```

the fix adds

```
outcode[UC7] = rhw_neg && UserClipFlagClipTestEnable[7]
```

This change routes UserClipFlag[7] into BAD, thus using UC7 logic to perform TA/TR/MC determination for the w=0 clip plane. Note that UCFClipTestEnableMask[7] still applies to UC7, though UC7 is now sourced from rhw_neg instead of UserClipFlag[7].

(4) BAD Forces SPAWN except in REJECT_ALL Mode

In the application of ClipMode, a BAD object (any vertex has rhw=NaN or UserClipFlag[7] set) will force a SPAWN unless ClipMode is REJECT_ALL. This is what provides support for cliptest/clipping against UserClipFlag[7]. Performance-wise, neither of these BAD cases are expected to occur very often (at least compared to negative W).

6.2.3.2.1 Support for Clip-Testing Against W=0

Software must set UserClipFlagsClipTestEnable[7] to enable clip-testing against the w=0 plane. If set, the rhw_neg bit will be routed to UC7, therefore permitting trivial reject, trivial accept and must clip determination like the other seven UCFs.

As the rhw_neg bit is handled as a UCF, it is subject to the UserClipFlagsMustClipEnable state bit. If this state bit is set, UCFs (by themselves) can lead to a mustclip determination (for the assumed purpose of 3D clipping against that plane). This is the expected setting for OGL use of the UCFs. If clear, the UCFs (by themselves) will not lead to a mustclip determination.

6.2.4 Tristrip Clipping Errata [Pre-DevBW, DevCL, DevCTG-A]

The HW clip unit has an implementation bug in the ClipDetermination logic related to the processing of tristrip primitives (TRISTRIP, TRISTRIP_REV and TRISTRIP_ADJ). If an object in the tristrip is determined to be a trivial reject case (TR), and the next object in the strip is determined to be a trivial accept (TA) case, a primitive topology can be emitted (for that TA object and possibly subsequent objects) with an incorrect primitive topology type. More specifically, instead of emitting a TRISTRIP_REV primtype, a TRISTRIP primtype may be omitted, and vice versa. This will lead to incorrect face culling (if enabled) downstream in the SF unit and be manifested by missing/extra triangles rendered.

TR to TA transitions can occur with when the tristrip crosses a viewport XY or UCF clip boundary. Note that this is not an issue with the VPZ or GBXY boundaries, as crossing one of those boundaries would cause at least one MustClip (MC) object between TR and TA objects and therefore the fault is not encountered. The same applies to VPXY when the GBXY cliptest is disabled (as there objects will get clipped against the VPXY boundaries).

There is a way for software to work around this problem, assuming that avoiding the use of trsitrips in the first place is not practical. Software can disable cliptest against the VPXY (assuming GBXY is disabled) and UCF flags prior to submitting tristrip primitives. This will likely incur a performance penalty as objects that could be trivially rejected against these boundaries will be sent down the pipe. Note that objects that would have been TR-ed against VPXY will likely be discarded in the SF unit's 2D clipping logic, so only partial SF processing will be incurred.

The same is not true for the UCF flags. When used for "ClipDistance", the could-have-been-TRed objects will be completely set-up and rasterized, with the PS kernel eventually killing all pixels. When used for 'CullDistance", the feature will appear to be non-functionaly as no culling will occur. One way to avoid some of this performance penalty would be for software to to leave the **UCF ClipTest Enable Bitmask** bits set, but also set the **UserClipFlags MustClip Enable** bit. This would (a) permit trivial reject against the UCFs, and (b) avoid the fault condition by forcing a MustClip case between TR and TA objects. The clip kernel would simply need to pass through any UCF-clipped only objects (which should be the default operation of the clip kernel).

6.2.5 Guard Band

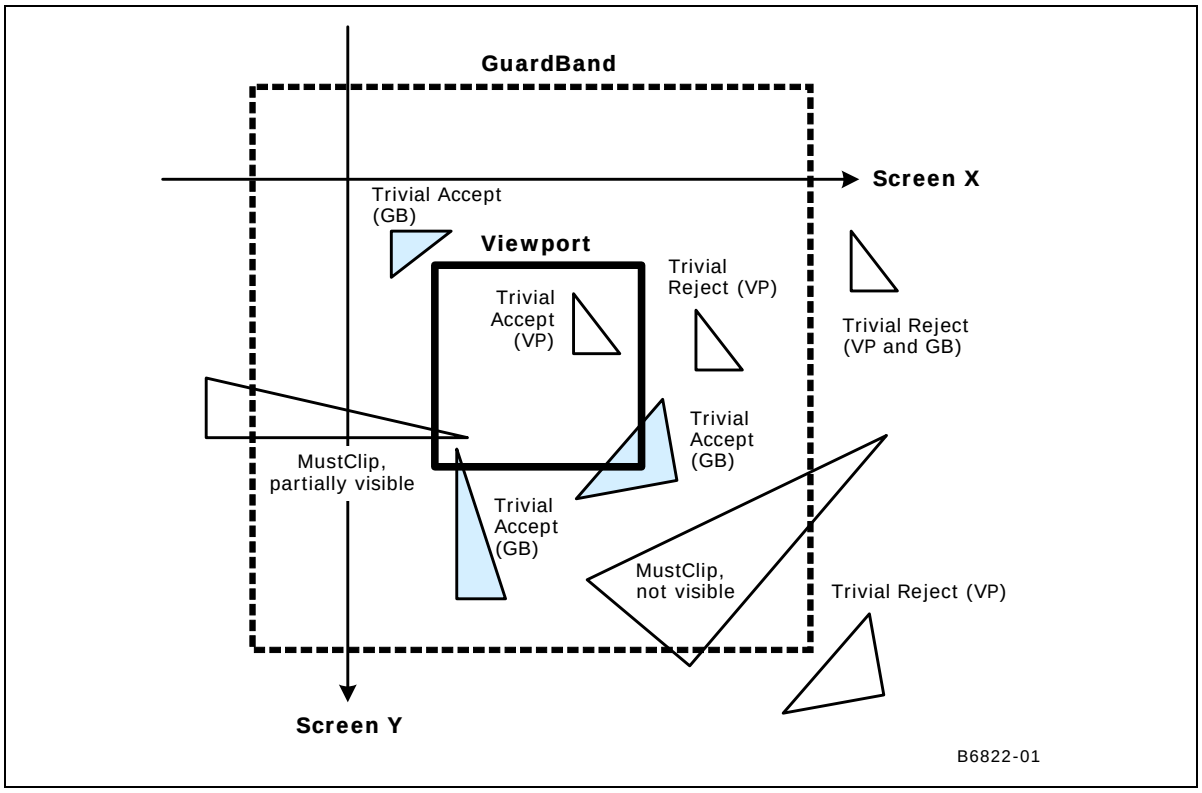
3DClipping is time consuming. For cases where 2DClipping is sufficient, we are willing to forgo 3DClipping and instead apply 2DClipping during rendering. In the general case, this is possible only when an object is totally within the ZMin and ZMax planes, and only clipping to the view volume X/Y MIN/MAX clip planes is required, as 2DClipping is restricted to a screen-aligned 2D rectangle.

However, we must ensure that the 2D extent of these objects do not exceed the limitations of the renderer's coordinate space (see Vertex X,Y Clamping and Quantization in the SF section). Therefore we define a 2D *guardband* region corresponding to (though likely somewhat smaller than) the maximum 2D extent supported by the renderer. During VertexClipTest, vertices are (optionally) subjected to an additional visibility test based on the 2D guardband region.

During ClipDetermination, if an object is not trivially-rejected from the 2D viewport, the XMIN_GB, XMAX_GB, YMIN_GB and YMAX_GB guardband outcodes are used instead of the XMIN, XMAX, YMIN, YMAX view volume outcodes to determine trivial-accept. This will allow objects that fall within the guardband and possibly intersect the viewport to be trivially-accepted and passed down the pipeline.

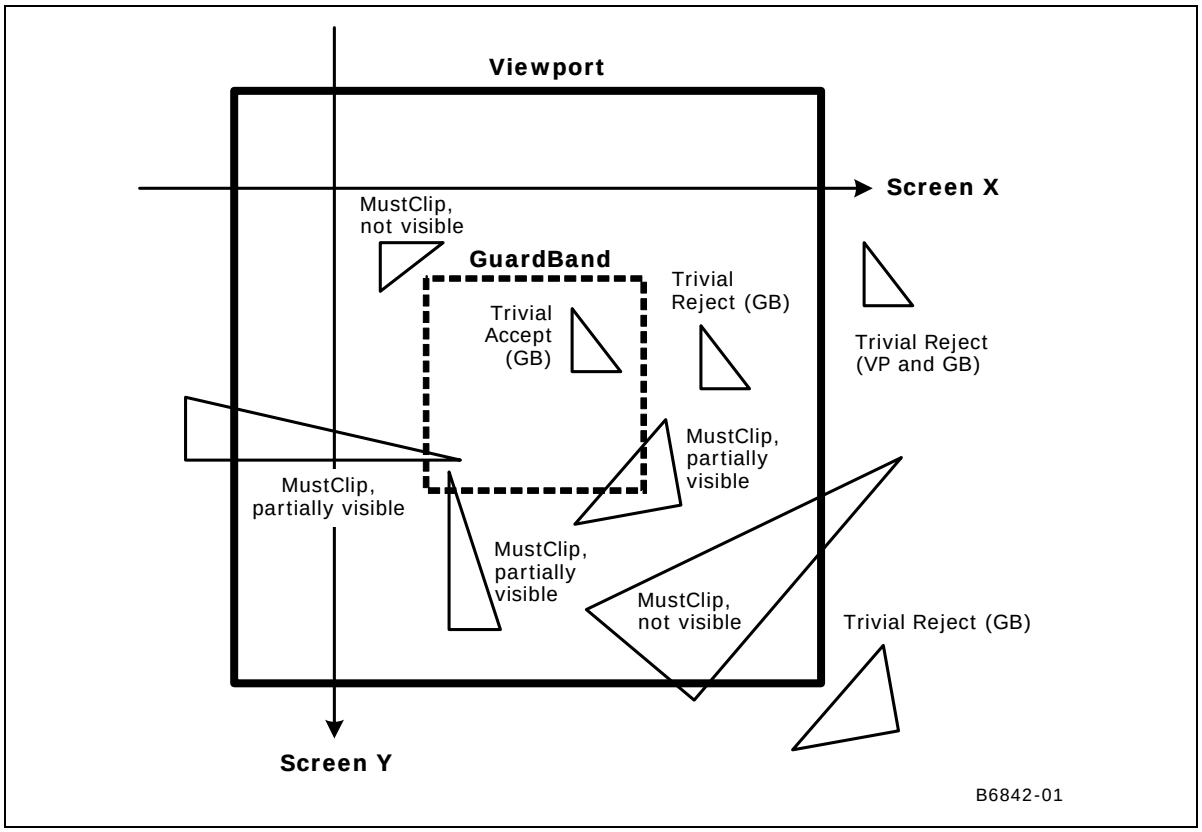
The diagram below shows some examples of objects (triangles) in relation to the viewport and guardband. The shaded triangles are examples of triangles that are not trivially accepted to the viewport but trivially accepted to the guardband and therefore passed to down the pipeline. Without the guardband, these triangles would have to be submitted to a CLIP thread.

Figure 6-2. Normal Guardband Operation



The CLIP stage needs to handle the case where the viewport XY is larger than the screen space coordinate range supported by the SF and WM units. This condition may arise when the API defines an implicit 2D clip between the viewport XY extent and the rendertarget. In the GENx 3D pipeline, the guardband must used to force explicit clipping in order to ensure legal coordinates are passed out of the CLIP stage. Therefore the CLIP unit supports a guardband that can be larger or smaller than the viewport (in any particular direction). The following diagram illustrates a case with a very large viewport, extending well beyond the guardband. Note that the only trivial accept case is where objects are completely within the guardband.

Figure 6-3. Very Large Viewport Case

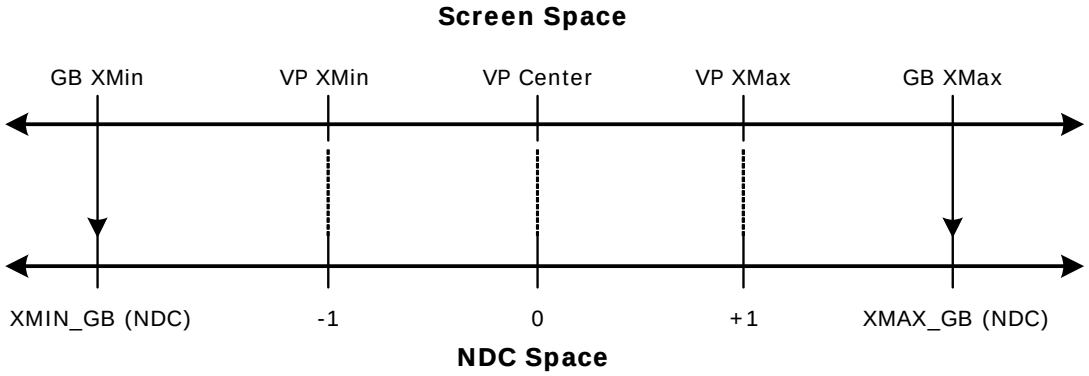


[Pre-DevILK]: Programming Restriction: Varying ViewportIndex within Strip-based Primitives

The forementioned case, where objects must be clip-tested and clipped against the guardband, leads to an somewhat obscure CLIP unit programming restriction. The fundamental issue is that the CLIP unit does not natively support clip-testing of strip topologies where the ViewportIndex can vary from vertex to vertex. The proper handling of this (i.e., applying the ViewportIndex from the leading vertex of the object to all object vertices) would require clip-testing on a per-object, not a per-vertex, basis. As the CLIP unit only uses the ViewportIndex to access the corresponding viewport-normalized guardband parameters, this exceptional condition could be ignored by turning off the guardband and thereby ignoring the incorrect results provided by the guardband cliptest. (Note that the viewport cliptest is performed against fixed values and therefore not dependent on the ViewportIndex). This leads to the conflict where the guardband needs to be enabled (to handle a very large guardband) but disabled (to ignore the incorrect cliptest results for strips with varying ViewportIndex). In this case, software will likely have to resort to use of the CLIP_ALL Clip Mode. This will pass all objects to a CLIP thread, where the correct clip-testing and clipping can be performed.

6.2.5.1 NDC Guardband Parameters

When the CLIP unit performs VertexClipTest in NDC space, the guardband limits must be provided as NDC coordinates. The diagram below shows how the guardband NDC coordinates are derived. Specifically, the XMIN_GB NDC coordinate is simply the ratio of the (screen space) distance from the screen space VP center to the screen space GB XMin boundary over the distance from the VP center to the VP XMin (left) boundary. A similar computation yields the XMAX_GB (right), YMIN_GB (bottom) and YMAX_GB (top) guardband NDC coordinates.



B6843-01

As these guardband parameters are defined relative to the viewport, each of the up-to-16 sets of viewport specifications supported in the 3D pipeline will require a corresponding set of guardband parameters. These guardband parameters are provided as a separate memory-resident state structure (CLIP_VIEWPORT), and referenced via the **Clipper Viewport State Pointer** contained in the CLIP_STATE structure. Note that the CLIP_VIEWPORT structure has a different definition than the SF_VIEWPORT structure used by the SF unit.

6.2.5.2 Screen Space Guardband Parameters

When the CLIP unit performs VertexClipTest in screen space, the guardband limits must be provided as screen space coordinates. Note that YMIN_GB will correspond to the screen space GB top, and YMAX_GB will correspond to the screen space GB bottom, which is opposite from the NDC case.

6.2.6 Vertex-Based Clip Testing & Considerations

The CLIP unit performs clip test and determines whether objects need to be clipped based solely on information (position, UserClipFlags) provided at the vertices of the object as they arrive at the clip stage. Issues arise if and when the corresponding rendered object is not constrained to the convex hull of the object. Different APIs impose different treatment of these conditions.

In addition and in the more general case, a CLIP thread could be used to convert the object (as defined by its vertices) into some arbitrary output primitive. In this case, the CLIP unit's ClipTest/ClipDetermination logic may not be suitable for determination of when to reject/accept/clip objects. In this case the ClipMode can be used to route all (or all non-rejected) objects to CLIP threads, where the proper clip-test and clipping can occur in the CLIP kernel.

One issue that arises is whether a trivial-reject to the VPXY is suitable. If this were allowed, an object might be discarded even if it would have been partially visible in the viewport. A second issue is whether a TA against the GB is suitable. If this were allowed, portions of the rendered object might be visible in the VP even if the object should have been clipped out of the VP.

6.2.6.1 Triangle Objects

In the normal processing of triangle-based primitives (tristrip/trilist/polygon/etc.), the footprint of each triangle is constrained to the 2D convex hull. I.e., the rendering of these triangles will not produce pixels outside of the triangle. Therefore the normal operation of the CLIP unit functions will support the proper clip testing and clip determination for triangle objects:

- Both the VPXY and GB clip boundaries can be utilized (as described above). If the triangle is TR against the VP, it can be discarded. Otherwise, if the triangle is TA against the GB, it can be passed down the pipeline (assuming it is TA against VPZ, UCFs, etc.) and properly handled by 2DClipping.
- The GB parameters can be programmed to coincide with the maximum allowable screen space extent (though making the GB marginally smaller than this max extent is highly recommended).

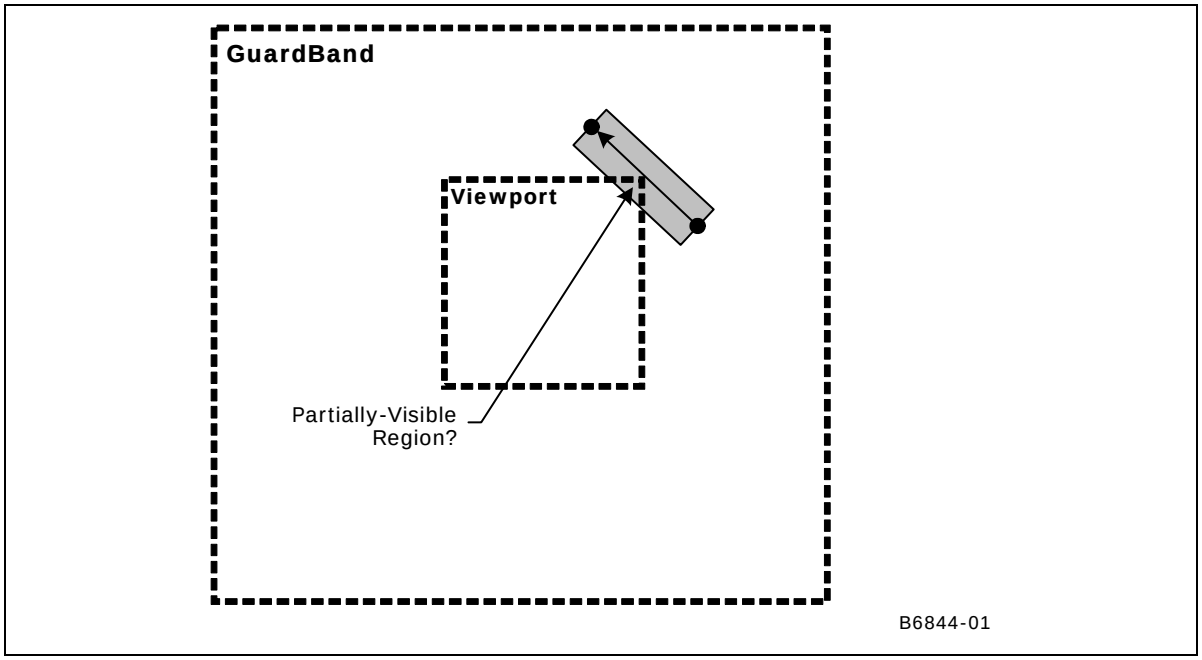
6.2.6.2 Non-Wide Line Objects

In the normal processing of non-wide, line-based primitives (linestrip/linelist/etc.), the footprint of each line is constrained to the 2D convex hull. I.e., the rendering of these lines will not produce pixels off of the line. Therefore the normal operation of the CLIP unit functions will support the proper clip testing and clip determination for non-wide line objects. (See Triangle Objects above).

6.2.6.3 Wide Line Objects

The GENx rendering hardware supports wide lines (solid lines with a line width or anti-aliased lines). When rendered, pixels outside of the convex hull will be generated.

The following diagram shows an example of a wide line that normally would be TA against the GB. If the TA is allowed, the partially-visible region of the line would be rendered.

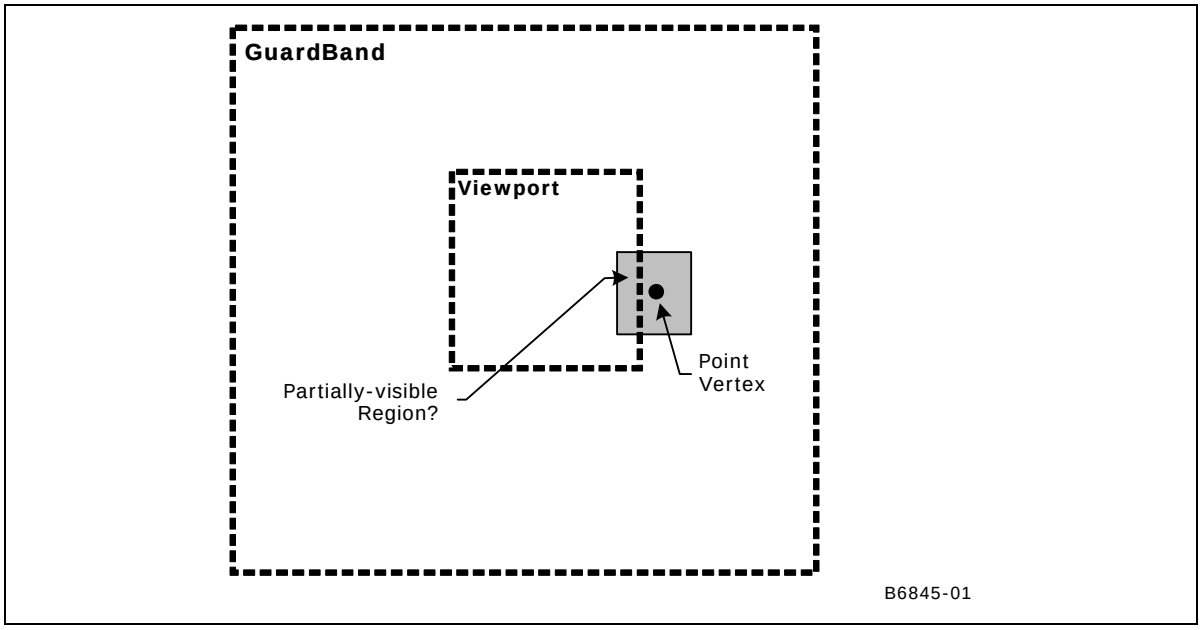


In general, OpenGL dictates that the partially-visible region must not be rendered. In this case the line must be clipped-out against the VPXY (not TA against the GB). To accomplish this, SW could disable the GB when drawing wide lines.

6.2.6.4 Wide Points

The GENx rendering hardware supports a width parameter for native line objects. When rendered, pixels surrounding the point (center) vertex will be generated.

The following diagram shows an example wide point that normally would be TR against the VPXY. If the TR is allowed, the partially-visible region of the point would not be rendered.



In general, OpenGL dictates that the partially-visible region must not be rendered. In this case the point must be TR against the VPXY (not TA against the GB). To accomplish this, SW could disable the GB when drawing wide points.

6.2.6.5 RECT LIST

The CLIP unit treats RECTLIST exactly like TRILIST. No special consideration is made for the implied 4th vertex of each rectangle (although ViewportXY and Guardband VertexClipTest theoretically should be sufficient to drive ClipDetermination). Given this, and the fact that RECTLIST is primarily intended for driver-generated "BLT" functions, there are number of restrictions on the use of RECTLIST, especially regarding the CLIP unit. Refer to the RECTLIST definition in 3D Pipeline.

6.2.7 3D Clipping

If an object needs to be clipped, it will be passed to the CLIP thread. The CLIP thread will perform some (arbitrary) algorithm to clip the primitive, and subsequently output "new" vertices as a primitive defining the visible region of the input object (assuming there is a visible region). In the process of spawning the CLIP thread, the input vertices may be considered "consumed" and therefore dereferenced. Therefore the CLIP thread will need to copy (if required) any input VUE data to a new output VUE – there is no mechanism to "output" input vertices other than copying.

Note : Thread based Clipping is supported only on [Pre-DevSNB].

6.3 CLIP Stage Input

As a stage of the GENx 3D pipeline, the CLIP stage receives inputs from the previous (GS) stage. Refer to *3D Overview* for an overview of the various types of input to a 3D Pipeline stage. The remainder of this subsection describes the inputs specific to the CLIP stage.

6.3.1 State

6.3.1.1 CLIP_ST ATE [Pre-DevSNB]

The following table describes the format and contents of the CLIP_STATE structure referenced by the **Pointer to CLIP State** field of the 3DSTATE_PIPELINED_POINTERS command.

CLIP_STATE

Project: [Pre-DevSNB]

Controls the CLIP stage hardware.

DWord	Bit	Description
0	31:6	Kernel Start Pointer Project: [Pre-DevILK] Address: GeneralStateOffset[31:6] Surface Type: Kernel This field specifies the starting location (1st GENx core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the General State Base Address. Errata / Description / Project: BWT007 — Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) — [DevBW]
	31:6	Kernel Start Pointer Project: [DevILK] Address: InstructionBaseOffset[31:6] Surface Type: Kernel This field specifies the starting location (1st GENx core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the Instruction Base Address.
	5:4	**Reserved Project: All Format: MBZ**

CLIP_STATE		
	0	**Reserved Project: All Format: MBZ**
1	31	**Single Program Flow (SPF)** **Project: A II** **Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1).** **Value Name Description Project** **0h Reserved All** **1h Enable Single Program Flow enabled All**
	30:26	**Reserved Project: All Format: MBZ**
	25:18	**Binding Table Entry Count Project: A II Format: U8** **Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state.** **Note: For kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache.** **[DevILK] MBZ**
	17	**Thread Priority** **Project: A II** **Specifies the priority of the thread for dispatch** **Value Name Description Project** **0h Normal Priority Normal Priority All** **1h High Priority High Priority [DevILK+]**
	16	**Floating Point Mode** **Project: A II** **Specifies the initial floating point mode used by the dispatched thread.** **Value Name Description Project** **0h IEEE-754 Use IEEE-754 Rules All** **1h Alternate Use alternate rules All**
	15:14	**Reserved Project: All Format: MBZ**
	13	**Illegal Opcode Exception Enable Project: A II Format: Enable** **This bit gets loaded into EU CR0.1[12] (note the bit # difference). See Exceptions and ISA Execution Environment.**
	12	**Reserved Project: All Format: MBZ**

CLIP_STATE		
	11	**Mask Stack Exception Enable** **Project: A II Format: Enable** **This bit gets loaded into EU CR0.1[11]. See Exceptions and ISA Execution Environment.**
	10:8	**Reserved Project: All Format: MBZ**
	7	**Software Exception Enable** **Project: A II Format: Enable** **This bit gets loaded into EU CR0.1[13] (note the bit # difference). See Exceptions and ISA Execution Environment.**
	6:0	**Reserved Project: All Format: MBZ**
2	31:10	Scratch Space Base Pointer Project: All Address: GeneralStateOffset[31:10] Surface Type: ScratchSpace Specifies the location of the scratch space area allocated to this FF unit, specified as a 1KB-granular offset from the General State Base Address. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by Per-Thread Scratch Space.
	9:4	**Reserved Project: A II Format: MBZ**
	3:0	**Per-Thread Scratch Space** **Project: A II** **Format: U4 power of 2 Bytes over 1K Bytes FormatDesc** **Range [0,11] indicating [1K Bytes, 2M Bytes]** **Specifies the amount of scratch space to be allocated to each thread spawned by this FF unit.** **The driver must allocate enough contiguous scratch space, starting at the Scratch Space Base Pointer, to ensure that the Maximum Number of Threads can each get Per-Thread Scratch Space size without exceeding the driver-allocated scratch space.**
3	31	**Reserved Project: All Format: MBZ**
	30:25	**Constant URB Entry Read Length** **Project: A II** **Format: U6 FormatDesc** **Range [0 ,63]** **Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments.**
	24	**Reserved Project: A II Format: MBZ**

CLIP_STATE		
	23:18	**Constant URB Entry Read Offset** **Project: A II** **Format: U6 FormatDesc** **Range [0 ,63]** **Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload.**
	23:17	**Reserved**
	16:11	Vertex URB Entry Read Length Project: All Format: U6 FormatDesc Range [1,63] Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments. Programming Notes It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread.
	10	**Reserved Project: All Format: MBZ**
	9:4	**Vertex URB Entry Read Offset** **Project: A II** **Format: U6 FormatDesc** **Range [0 ,63]** **Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread.**
	3:0	**Dispatch GRF Start Register for URB Data** **Project: A II** **Format: U4 FormatDesc** **Range [0,15] indicating GRF [R0,R15]** **Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload.**
4	31:30	**Reserved Project: All Format: MBZ**

CLIP_STATE

	29:25	**Maximum Number of Threads** **Project: Pre-De vILK** **Format: U5 thread count – 1** **Range [0,1] indicating thread count of [1,2]** **Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock.** **Programming Notes — Project** **When running in dual-thread mode, the Number of URB Entries field must contain an even number. Each thread will be allocated one half the total number of entries. — All** **A URB_FENCE command must be issued subsequent to any change to the value in this field (via PIPELINE_STATE_POINTERS) and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER). See Graphics Processing Engine (Command Ordering Rules) — All**
	29:25	**Maximum Number of Threads** Project: DevILK Format: U5 thread count – 1 Range [0,15] indicating thread count of [1,16] Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. **Programming Notes** A URB_FENCE command must be issued subsequent to any change to the value in this field and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER). See *Graphics Processing Engine* (Command Ordering Rules)
	24	**Reserved Project: All Format: MBZ**
	23:19	**URB Entry Allocation Size** **Project: A ll** **Format: U5 count (of 512-bit units) – 1** **Range [0,31] = [1,32] 512-bit units = [2,64] 256-bit URB rows** **Specifies the length of each URB entry owned by this FF unit.** **Programming Notes — Project** **Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. — All**
	18	**Reserved Project: All Format: MBZ**

CLIP_STATE

	17:11	**Number of URB Entries** **Project: A** **II** **Format:** **U7** **Count of URB entries** **Range** **[1,32] if GS enabled, otherwise ignored.** **Specifies the number of URB entries that are used by this FF unit.** **Programming Notes** **Project** **When running in dual-thread mode, the Number of URB Entries field must contain an even number. Each thread will be allocated one half the total number of entries.** **All** **If ENABLED, the GS stage must be allocated at least one URB entry** **All** **Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE.** **All**
	10	**Clipper Statistics Enable** **Project: A** **II** **Format: Enable** **This bit controls whether Clip-unit-specific statistics register(s) can be incremented.** **Value Name** **Description** **Project** **0h** **Disable** **CL_INVOCATIONS_COUNT cannot increment** **All** **1h** **Enable** **CL_INVOCATIONS_COUNT can increment** **All**
	9	**GS Output Object Statistics Enable** Project: Pre-DevILK Format: Enable FormatDesc If ENABLED, the CLIP stage will increment GS_PRIMITIVES_COUNT on behalf of the GS stage as appropriate; see the Statistics Gathering section of this chapter. If DISABLED, GS_PRIMITIVES_COUNT will be left unchanged. **Programming Notes** SW should clear this bit whenever **Statistics Enable** in GS_STATE is clear *or* the GS stage is disabled If the GS fixed function is enabled and GS statistics gathering is desired, the CLIP stage *cannot be disabled* (put in pass-through mode) and this bit *must* be set. **Clip Mode** may be set to CLIPMODE_ACCEPT_ALL to effectively disable clipping, however.
	9	**Reserved Project: DevILK Format: MBZ**
	8:0	**Reserved Project: All Format: MBZ**
5	31	**Reserved Project: All Format: MBZ**

CLIP_STATE		
	30	**API Mode** **Project: A II** **Controls the definition of the NEAR clipping plane** **Value Name / Description / Project** **0h API_OGL / NEAR VP boundary == 0.0 (NDC) / All** **1h Reserved / NEAR VP boundary == -1.0 (NDC) / All**
	29	**Vertex Position Space** **Project: [Pre-De vSNB]** **This field specifies the coordinate system within which the incoming Vertex Position X,Y,Z values are defined. The setting affects VertexClipTest.** **Value Name / Description / Project** **0h VPOS_NDCSPACE / Vertex Position is in NDC space / All** **1h VPOS_SCREENSPACE / Vertex Position is in Screen space / All** **[DevILK] The VPOS_SCREENSPACE cannot be programmed when the Fixed Function Clipper is enabled, i.e. when CLIP MODE is set to CLIPMODE_NORMAL**
	28	**Viewport XY ClipTest Enable** **Project: A II** **Format: Enable** **This field is used to control whether the Viewport X,Y extents are considered in VertexClipTest. See Tristrip Clipping Errata subsection.**
	27	**Viewport Z ClipTest Enable** **Project: A II** **Format: Enable** **This field is used to control whether the Viewport Z extents (near, far) are considered in VertexClipTest.**
	26	**Guardband ClipTest Enable** **Project: A II** **Format: Enable** **This field is used to control whether the Guardband X,Y extents are considered in VertexClipTest for non-point objects.** **If the Guardband ClipTest is DISABLED but the Viewport XY ClipTest is ENABLED, ClipDetermination operates as if the Guardband were coincident with the Viewport.** **If both the Guardband and Viewport XY ClipTest are DISABLED, all vertices are considered "visible" with respect to the XY directions.**
	25	**Negative W ClipTest Enable** **Project: CTG+** **Format: Enable** **This field is used to control whether the w=0 plane (NEGW) is considered in VertexClipTest.**

CLIP_STATE		
	24	**UserClipFlags MustClip Enable** **Project: A II** **Format: Enable** **This field is used to include the UserClipFlags in MustClip determination, in order to support clipping to User Clip Planes. If ENABLED, the setting of enabled UserClipFlag bits can cause a CLIP thread to be spawned. If the enabled UCF values at the object vertices do not indicate a trivial accept or reject with relation to the UCFs, then a CLIP thread will be spawned (unless the object is trivially rejected for other reasons).** **If DISABLED, the UserClipFlags are only used for trivial accept or reject determination, and will not lead to a CLIP thread being spawned unless indicated by other cliptest results (or SV bits).**
	23:16	**UserClipFlags ClipTest Enable Bitmask** **Project: A II** **Format: Enable** **This field is used to include the UserClipFlags in MustClip determination, in order to support clipping to User Clip Planes. If ENABLED, the setting of enabled UserClipFlag bits can cause a CLIP thread to be spawned. If the enabled UCF values at the object vertices do not indicate a trivial accept or reject with relation to the UCFs, then a CLIP thread will be spawned (unless the object is trivially rejected for other reasons).** **If DISABLED, the UserClipFlags are only used for trivial accept or reject determination, and will not lead to a CLIP thread being spawned unless indicated by other cliptest results (or SV bits).**

CLIP_STATE

	15:13	**Clip Mode** Project: All This field specifies a general mode of the CLIP unit, when the CLIP unit is ENABLED.

Value Name		Description	Project
0h	CLIPMODE_NORMAL	[Pre-DevILK TrivialAccept objects are passed down the pipeline, MustClip objects are passed to CLIP threads, TrivialReject and BAD objects are discarded [DevILK] TrivialAccept objects are passed down the pipeline, MustClip objects Clipped in the Fixed Function Clipper HW, TrivialReject and BAD objects are discarded	All
1h	CLIPMODE_ALL	All objects (including BAD objects & TrivReject) are passed to CLIP threads, regardless of classification	All
2h	CLIPMODE_CLIP_NON_REJECTED	TrivialAccept and MustClip objects are passed to CLIP threads, TrivReject and BAD objects are discarded	All
3h	CLIPMODE_REJECT_ALL	All objects are discarded	All
4h	CLIPMODE_ACCEPT_ALL	All objects (except BAD objects) are trivially accepted. This effectively disables the clip-test/clip-determination function. Note that the CLIP unit will still filter out adacency information, which may be required since the SF unit does not accept primitives with adjacency.	All
5h	CLIPMODE_KERNELCLIP	[Pre-DevILK] Reserved [DevILK]: force kernel clip mode. If this bit is set, when CLunit detects a MC case a clip thread is launched.	All
6h-7h		Reserved	All

Errata	Description	Project
#	See previous sections (W Clipping Errata) for the description of errata regarding negative W and trivial reject. These errata impact the programming of Clip Mode.	DevBW, DevCL, DevCL

12:0	**Reserved Project: Pre-DevILK Format: MBZ**
12:6	**Reserved Project: DevILK Format: MBZ**

CLIP_STATE		
	5:4	**Triangle Strip/List Provoking Vertex Select** **Project: De vILK+ Format: U2** **[DevILK]: Triangle Strip/List Provoking Vertex Select: Selects which vertex of a triangle (in a triangle strip or list primitive) is considered the "provoking vertex". Used for flat shading of primitives.** **Format = 0-based vertex index** **0h = Vertex 0** **1h = Vertex 1** **2h = Vertex 2** **3h = Reserved**
	3:2	**Line Strip/List Provoking Vertex Select** **Project: De vILK+ Format: U2** **[DevILK] Line Strip/List Provoking Vertex Select: Selects which vertex of a line (in a line strip or list primitive) is considered the "provoking vertex".** **Format = 0-based vertex index** **0h – Vertex 0** **1h – Vertex 1** **2h – Reserved** **3h – Reserved**
	1:0	**Triangle Fan Provoking Vertex Select** **Project: De vILK+ Format: U2** **[DevILK]: Triangle Fan Provoking Vertex Select: Selects which vertex of a triangle (in a triangle fan primitive) is considered the "provoking vertex".** **Format = 0-based vertex index** **0h = Vertex 0** **1h = Vertex 1** **2h = Vertex 2** **3h = Reserved**
6	31:5	Clipper Viewport State Pointer Project: Pre-GT Address: GeneralStateOffset[31:5] Surface Type: CLIP_VIEWPORT Specifies the location of the current CLIP_VIEWPORT data structure, as a 32-byte aligned offset from General State Base Pointer). The CLIP unit accesses the viewport state throught it's Instruction/State Cache (ISC).
	4:0	**Reserved Project: A II Format: MBZ**
7	31:0	**Screen Space Viewport X Min** **Project: A II Format: FLO AT32** **This field contains the XMin (left) extent of the screen-space viewport. This field is only used when Vertex Position Space = VPOS_SCREENSPACE.**

CLIP_STATE		
8	31:0	**Screen Space Viewport X Max** **Project: A II** **Format: FLOAT32** **This field contains the XMax (right) extent of the screen-space viewport. This field is only used when Vertex Position Space = VPOS_SCREENSPACE.**
9	31:0	**Screen Space Viewport Y Min** **Project: A II** **Format: FLOAT32** **This field contains the YMin (top) extent of the screen-space viewport. This field is only used when Vertex Position Space = VPOS_SCREENSPACE.**
10	31:0	**Screen Space Viewport Y Max** **Project: A II** **Format: FLOAT32** **This field contains the YMax (bottom) extent of the screen-space viewport. This field is only used when Vertex Position Space = VPOS_SCREENSPACE.**

6.3.1.2 CLIP_VIEWPORT

The viewport-related state is stored as an array of up to 16 elements, each of which contains the DWords described here. The start of each element is spaced 4 DWords apart. The first element of the viewport state array is aligned to a 32-byte boundary, and is located at (**General State Base Pointer** + **Clipper Viewport State Pointer)**. Note that the definition of the CLIP_VIEWPORT structure differs from the SF_VIEWPORT structure used by the SF unit.

CLIP_VIEWPORT			
Project:	DevILK **+**	**Length Bias:**	2
Viewport data used by the Clip unit.			
DWord	**Bit**	**Description**	
0	31:0	XMin Clip Guardband Project: All Format: FLOAT32 FormatDesc For VPOS_NDCSPACE: This 32-bit float represents the XMin guardband boundary (normalized to Viewport.XMin == -1.0f). This corresponds to the left boundary of the NDC guardband. For: VPOS_SCREENSPACE This 32-bit float represents the XMin guardband boundary in screen space coordinates. This corresponds to the left boundary of the screen space guardband.	

CLIP_VIEWPORT		
1	31:0	**XMax Clip Guardband** Project: All Format: FLOAT32 FormatDesc For VPOS_NDCSPACE: This 32-bit float represents the XMax guardband boundary (normalized to Viewport.XMax == 1.0f). This corresponds to the right boundary of the NDC guardband. For: VPOS_SCREENSPACE This 32-bit float represents the XMax guardband boundary in screen space coordinates. This corresponds to the right boundary of the screen space guardband.
2	31:0	**YMin Clip Guardband** Project: All Format: FLOAT32 FormatDesc For VPOS_NDCSPACE: This 32-bit float represents the YMin guardband boundary (normalized to Viewport.YMin == -1.0f). This corresponds to the bottom boundary of the NDC guardband. For: VPOS_SCREENSPACE This 32-bit float represents the YMin guardband boundary in screen space coordinates. This corresponds to the top boundary of the screen space guardband.
3	31:0	**YMax Clip Guardband** Project: All Format: FLOAT32 FormatDesc **For VPOS_NDCSPACE:** **This 32-bit float represents the YMax guardband boundary (normalized to Viewport.YMax == 1.0f). This corresponds to the top boundary of the NDC guardband.** **For: VPOS_SCREENSPACE** **This 32-bit float represents the YMax guardband boundary in screen space coordinates. This corresponds to the bottom boundary of the screen space guardband.**

6.3.1.3 CLIP_VIEWPORT

The viewport-related state is stored as an array of up to 16 elements, each of which contains the DWords described here. The start of each element is spaced 4 DWords apart. The first element of the viewport state array is aligned to a 32-byte boundary, and is located at (**General State Base Pointer** + **Clipper Viewport State Pointer)**.

Note that the definition of the CLIP_VIEWPORT structure differs from the SF_VIEWPORT structure used by the SF unit.

CLIP_VIEWPORT		
Project:	All	
Viewport data used by the Clip unit.		
DWord	**Bit**	**Description**
0	31:0	**XMin Clip Guardband** **Project: A II** **Format: FLO AT32** **For VPOS_NDCSPACE:** **This 32-bit float represents the XMin guardband boundary (normalized to Viewport.XMin == -1.0f). This corresponds to the left boundary of the NDC guardband.** **For: VPOS_SCREENSPACE** **This 32-bit float represents the XMin guardband boundary in screen space coordinates. This corresponds to the left boundary of the screen space guardband.**
1	31:0	**XMax Clip Guardband** **Project: A II** **Format: FLO AT32** **For VPOS_NDCSPACE:** **This 32-bit float represents the XMax guardband boundary (normalized to Viewport.XMax == 1.0f). This corresponds to the right boundary of the NDC guardband.** **For: VPOS_SCREENSPACE** **This 32-bit float represents the XMax guardband boundary in screen space coordinates. This corresponds to the right boundary of the screen space guardband.**
2	31:0	**YMin Clip Guardband** **Project: A II** **Format: FLO AT32** **For VPOS_NDCSPACE:** **This 32-bit float represents the YMin guardband boundary (normalized to Viewport.YMin == -1.0f). This corresponds to the bottom boundary of the NDC guardband.** **For: VPOS_SCREENSPACE** **This 32-bit float represents the YMin guardband boundary in screen space coordinates. This corresponds to the top boundary of the screen space guardband.**

CLIP_VIEWPORT		
3	31:0	**YMax Clip Guardband** **Project: A II** **Format: FLO AT32** **For VPOS_NDCSPACE:** **This 32-bit float represents the YMax guardband boundary (normalized to Viewport.YMax == 1.0f). This corresponds to the top boundary of the NDC guardband.** **For: VPOS_SCREENSPACE** **This 32-bit float represents the YMax guardband boundary in screen space coordinates. This corresponds to the bottom boundary of the screen space guardband.**

6.4 VertexClipTest Function

The VertexClipTest function compares each incoming vertex position (x,y,z,w) with various viewport and guardband parameters (either hard-coded values or specified by state variables).

The RHW component of the incoming vertex position is tested for NaN value, and if a NaN is detected, the vertex is marked as "BAD" by setting the outcode[BAD]. [**DevILK+**]: If a NaN is deteced in any vertex homogeneous x,y,z,w component or an enabled ClipDistance value, the vertex is marked as "BAD" by setting the outcode[BAD].

In general, any object containing a BAD vertex will be discarded, as how to clip/render such objects is undefined. However, in the case of CLIP_ALL mode, a CLIP thread will be spawned even for objects with "BAD" vertices. The CLIP kernel is required to handle this case, and can examine the **Object Outcode [BAD]** payload bit to detect the condition. (Note that the VP and GB Object Outcodes are UNDEFINED when BAD is set).

If the incoming RHW coordinate is negative (including negative 0) the NEGW outcode is set. Also, this condition is used to select the proper comparison functions for the VP and GB outcode tests (below).

Next, the VPXY and GB outcodes are computed, depending on the corresponding enable SV bits. If one of VPXY or GB is disabled, the enabled set of outcodes are copied to the disabled set of outcodes. This effectively defines the disabled boundaries to coincide with the enabled boundaries (i.e., disabling the GB is just like setting it to the VPXY values, and vice versa.).

The VPZ outcode is computed as required by the API mode SV.

Finally, the incoming UserClipFlags are masked and copied to corresponding outcodes.

The following algorithm is used by VertexClipTest:

```
//
// Vertex ClipTest
//
//  On input:
//  if (CLIP.PreMapped)
//    x,y are viewport mapped
```

```
//    z is NDC ([0,1] is visible)
//  else
//    x,y,z are NDC (post-perspective divide)
//  w is always 1/w

//
// Initialize outCodes to “inside”
//
outCode[*] = 0

//
// Check if w is NaN
// Any object containing one of these “bad” vertices will likely be discarded
 //
 #ifdef (DevBW-E0 || DevCL-B)
 if (ISNAN(w)|| UserClipFlag[7])
 #elseif (DevCTG)
 if (ISNAN(w))
 #elseif (DevILK+)
 if (ISNAN(homogeneous x,y,z,w or enabled ClipDistance value)
 #endif
{
   outCode[BAD] = 1
}

//
//  If 1/w is negative, w is negative and therefore outside of the w=0 plane
//
//
rhw_neg = ISNEG(rhw)
if (rhw_neg)
{
#ifdef (PreDevBW-E0 || DevCL-A)
   outCode[VP_XMIN] = 1
   outCode[VP_XMAX] = 1
   outCode[VP_YMIN] = 1
   outCode[VP_YMAX] = 1
   outCode[VP_ZMIN] = 1
   outCode[VP_ZMAX] = 1
   outCode[GB_XMIN] = 1
   outCode[GB_XMAX] = 1
   outCode[GB_YMIN] = 1
   outCode[GB_YMAX] = 1
   goto UserClipFlags
 #elseifdef (DevCTG+)
   outCode[NEGW] = 1
 #endif
}

//
// View Volume Clip Test
//   If Premapped, the 2D viewport is defined in screen space
//     otherwise the canonical NDC viewvolume applies ([-1,1])
//
if (CLIP_STATE.PreMapped)
{
   vp_XMIN = CLIP_STATE.VP_XMIN
   vp_XMAX = CLIP_STATE.VP_XMAX
   vp_YMIN = CLIP_STATE.VP_YMIN
```

```
      vp_YMAX = CLIP_STATE.VP_YMAX
   } else {
      vp_XMIN = -1.0f
      vp_XMAX = +1.0f
      vp_YMIN = -1.0f
      vp_YMAX = +1.0f
   }
    if (CLIP_STATE.ViewportXYClipTestEnable) {
      outCode[VP_XMIN] = (x < vp_XMIN)
      outCode[VP_XMAX] = (x > vp_XMAX)
      outCode[VP_YMIN] = (y < vp_YMIN)
      outCode[VP_YMAX] = (y > vp_YMAX)
     #ifdef (DevBW-E0)
       if (rhw_neg) {
           outCode[VP_XMIN] = (x >= vp_XMIN)
           outCode[VP_XMAX] = (x <= vp_XMAX)
           outCode[VP_YMIN] = (y >= vp_XMIN)
           outCode[VP_YMAX] = (y <= vp_XMAX)
       }
    #endif
     #ifdef (DevCTG+)
       if (rhw_neg) {
           outCode[VP_XMIN] = (x > vp_XMIN)
           outCode[VP_XMAX] = (x < vp_XMAX)
           outCode[VP_YMIN] = (y > vp_XMIN)
           outCode[VP_YMAX] = (y < vp_XMAX)
       }
    #endif
    }
    if (CLIP_STATE.ViewportZClipTestEnable) {
      if (CLIP_STATE.APIMode == APIMODE_NOT_OGL) {
           vp_ZMIN = 0.0f
           vp_ZMAX = 1.0f
      } else {  // OGL
           vp_ZMIN = -1.0f
           vp_ZMAX = 1.0f
      }
      outCode[VP_ZMIN] = (z < vp_ZMIN)
      outCode[VP_ZMAX] = (z > vp_ZMAX)
     #ifdef (DevBW-E0)
       if (rhw_neg) {
           outCode[VP_ZMIN] = (z >= vp_ZMIN)
           outCode[VP_ZMAX] = (z <= vp_ZMAX)
       }
    #endif
     #ifdef (DevCTG+)
       if (rhw_neg) {
           outCode[VP_ZMIN] = (z > vp_ZMIN)
           outCode[VP_ZMAX] = (z < vp_ZMAX)
       }
    #endif
   }

   //
```

```
   // Guardband Clip Test
   //
   if {CLIP_STATE.GuardbandClipTestEnable) {
       gb_XMIN = CLIP_STATE.Viewport[vpindex].GB_XMIN
       gb_XMAX = CLIP_STATE.Viewport[vpindex].GB_XMAX
       gb_YMIN = CLIP_STATE.Viewport[vpindex].GB_YMIN
       gb_YMAX = CLIP_STATE.Viewport[vpindex].GB_YMAX
       outCode[GB_XMIN] = (x < gb_XMIN)
       outCode[GB_XMAX] = (x > gb_XMAX)
       outCode[GB_YMIN] = (y < gb_YMIN)
       outCode[GB_YMAX] = (y > gb_YMAX)
     #ifdef (DevBW-E0)
       if (rhw_neg) {
           outCode[GB_XMIN] = (x >= gb_XMIN)
           outCode[GB_XMAX] = (x <= gb_XMAX)
           outCode[GB_YMIN] = (y >= gb_YMIN)
           outCode[GB_YMAX] = (y <= gb_YMAX)
       }
    #endif
     #ifdef (DevCTG+)
       if (rhw_neg) {
           outCode[GB_XMIN] = (x > gb_XMIN)
           outCode[GB_XMAX] = (x < gb_XMAX)
           outCode[GB_YMIN] = (y > gb_YMIN)
           outCode[GB_YMAX] = (y < gb_YMAX)
       }
    #endif
   }

   //
   // Handle case where either VP or GB disabled (but not both)
   // In this case, the disabled set take on the outcodes of the enabled set
   //
   if (CLIP_STATE.ViewportXYClipTestEnable && !CLIP_STATE.GuardbandClipTestEnable)
{
       outCode[GB_XMIN] = outCode[VP_XMIN]
       outCode[GB_XMAX] = outCode[VP_XMAX]
       outCode[GB_YMIN] = outCode[VP_YMIN]
       outCode[GB_YMAX] = outCode[VP_YMAX]
   } else if (!CLIP_STATE.ViewportXYClipTestEnable &&
CLIP_STATE.GuardbandClipTestEnable) {
       outCode[VP_XMIN] = outCode[GB_XMIN]
       outCode[VP_XMAX] = outCode[GB_XMAX]
       outCode[VP_YMIN] = outCode[GB_YMIN]
       outCode[VP_YMAX] = outCode[GB_YMAX]
   }

   //
   // X/Y/Z NaN Handling
   //
   xyorgben = (CLIP_STATE.ViewportXYClipTestEnable ||
CLIP_STATE.GuardbandClipTestEnable)
   if (isNAN(x)) {
        outCode[GB_XMIN] = xyorgben
        outCode[GB_XMAX] = xyorgben
       outCode[VP_XMIN] = xyorgben
       outCode[VP_XMAX] = xyorgben
    }
```

```
   if (isNAN(y)) {
        outCode[GB_YMIN] = xyorgben
        outCode[GB_YMAX] = xyorgben
       outCode[VP_YMIN] = xyorgben
       outCode[VP_YMAX] = xyorgben
   }

   if (isNaN) {
        outCode[VP_ZMIN] = CLIP_STATE.ViewportZClipTestEnable
        outCode[VP_ZMAX] = CLIP_STATE.ViewportZClipTestEnable
   }

   //
   // UserClipFlags
   //
   ExamineUCFs
   for (i=0; i<7; i++)
   {
      outCode[UC0+i] = userClipFlag[i] &
CLIP_STATE.UserClipFlagsClipTestEnableBitmask[i]
    }
    #ifdef (DevBW-E0 || DevCL-B)
      outCode[UC7] = rhw_neg & CLIP_STATE.UserClipFlagsClipTestEnableBitmask[7]
    #else
      outCode[UC7] = userClipFlag[i] &
CLIP_STATE.UserClipFlagsClipTestEnableBitmask[7]
    #endif
```

6.5 Object Staging

The CLIP unit's Object Staging Buffer (OSB) accepts streams of input vertex information packets, along with each vertex's VertexClipTest result (outCode). This information is buffered until a complete object or the last vertex of the primitive topology is received. The OSB then performs the ClipDetermination function on the object vertices, and takes the actions required by the results of that function.

6.5.1 Partial Object Removal

The OSB is responsible for removing incomplete LINESTRIP and TRISTRIP objects that it may receive from the preceding stage (GS). Partial object removal is not supported for other primitive types due to either (a) the GS is not permitted to output those primitive types (e.g., primitives with adjacency info), and the VF unit will have removed the partial objects as part of 3DPRIMITIVE processing, or (b) although the GS thread is allowed to output the primitive type (e.g., LINELIST), it is assumed that the GS kernel will be correctly implemented to avoid outputting partial objects (or pipeline behavior is UNDEFINED).

An object is considered 'partial' if the last vertex of the primitive topology is encountered (i.e., PrimEnd is set) before a complete set of vertices for that object have been received. Given that only LINESTRIP and TRISTRIP primitive types are subject to CLIP unit partial object removal, the only supported cases of partial objects are 1-vertex LINESTRIPs and 1 or 2-vertex TRISTRIPs.

Partial Object Removal is performed only when the CLIP stage is ENABLED.

6.5.2 ClipDetermination Function

In ClipDetermination, the vertex outcodes of the primitive are combined in order to determine the clip status of the object (TR: trivially reject; TA: trivial accept; MC: must clip; BAD: invalid coordinate). Only those vertices included in the object are examined (3 vertices for a triangle, 2 for a line, and 1 for a point). The outcode bit arrays for the vertices are separately ANDed (intersection) and ORed (union) together (across vertices, not within the array) to yield objANDCode and objORCode bit arrays.

TR/TA against interesting boundary subsets are then computed. The TR status is computed as the logical OR of the appropriate objANDCode bits, as the vertices need only be outside of one common boundary to be trivially rejected. The TA status is computed as the logical NOR of the appropriate objORCode bits, as any vertex being outside of any of the boundaries prevents the object from being trivially accepted.

If any vertex contains a BAD coordinate, the object is considered BAD and any computed TR/TA results will effectively be ignored in the final action determination.

Next, the boundary subset TR/TA results are combined to determine an overall status of the object. If the object is TR against any viewport or enabled UC plane, the object is considered TR. Note that, by definition, being TR against a VPXY boundary implies that the vertices will be TR agains the corresponding GB boundary, so computing `TR_GB` is unnecessary.

The treatment of the UCF outcodes is conditional on the UserClipFlags MustClip Enable state. If DISABLED, an object that is not TR against the UCFs is considered TA against them. Put another way, objects will only be culled (not clipped) with respect to the UCFs. If ENABLED, the UCF outcodes are treated like the other outcodes, in that they are used to determine TR, TA or MC status, and an object can be passed to a CLIP thread simply based on it straddling a UCF.

Finally, the object is considered MC if it is neither TR or TA.

The following logic is used to compute the final TR, TA, and MC status.

```
//
// ClipDetermination
//

//
// Compute objANDCode and objORCode
//
switch (object type) {
case POINT:
{
   objANDCode[…] = v0.outCode[…]
   objORCode[…]  = v0.outCode[…]
} break
case LINE:
{
   objANDCode[…] = v0.outCode[…] & v1.outCode[…]
   objORCode[…]  = v0.outCode[…] | v1.outCode[…]
} break
case TRIANGLE:
{
   objANDCode[…] = v0.outCode[…] & v1.outCode[…] & v2.outCode[…]
   objORCode[…]  = v0.outCode[…] | v1.outCode[…] | v2.outCode[…]
} break

//
```

```
    // Determine TR/TA against interesting boundary subsets
    //
    TR_VPXY = (objANDCode[VP_L] | objANDCode[VP_R] | objANDCode[VP_T] |
objANDCode[VP_B])
    TR_GB   = (objANDCode[GB_L] | objANDCode[GB_R] | objANDCode[GB_T] |
objANDCode[GB_B])
    TA_GB   = !(objORCode[GB_L] | objORCode[GB_R] | objORCode[GB_T] |
objORCode[GB_B])
    TA_VPZ  = !(objORCode[VP_N] | objORCode[VP_Z])
    TR_VPZ  = (objANDCode[VP_N] | objANDCode[VP_Z])
    TA_UC   = !(objORCode[UC0] | objORCode[UC1] | … | objORCode[UC7])
    TR_UC   = (objANDCode[UC0] | objANDCode[UC1] | … | objANDCode[UC7])
    BAD     = objORCode[BAD]
    #ifdef (DevCTG+)
    TA_NEGW  = !objORCode[NEGW]
    TR_NEGW  = objANDCode[NEGW]
    #endif

    //
    // Trivial Reject
    //
    //    An object is considered TR if all vertices are TR against any common
boundary
    //    Note that this allows the case of the VPXY being outside the GB
    //
    #ifdef (DevCTG+)
    TR = TR_GB || TR_VPXY || TR_VPZ || TR_UC || TR_NEGW
    #else
    TR = TR_GB || TR_VPXY || TR_VPZ || TR_UC
    #endif

    //
    // Trivial Accept
    //
    // For an object to be TA, it must be TA against the VPZ and GB, not TR,
    //   and considered TA against the UC planes and (DevCTG+) NEGW
    // If the UCMC mode is disabled, an object is considered TA against the UC
    //   as long as it isn't TR against the UC.
    // If the UCMC mode is enabled, then the object really has to be TA against the
UC
    //   to be considered TA
    // In this way, enabling the UCMC mode will force clipping if the object is
neither
    //   TA or TR against the UC
    //
    #ifdef (DevCTG+)
    TA = !TR && TA_GB && TA_VPZ && TA_NEGW
    #else
    TA = !TR && TA_GB && TA_VPZ
    #endif
    UCMC = CLIP_STATE.UserClipFlagsMustClipEnable
    TA = TA && ( (UCMC && TA_UC) || (!UCMC && !TR_UC) )

    //
    // MustClip
    // This is simply defined as not TA or TR
    // Note that exactly one of TA, TR and MC will be set
    //
    MC = !(TA || TR)
```

6.5.3 ClipMode

The ClipMode state determines what action the CLIP unit takes given the results of ClipDetermination. The possible actions are:

- PASSTHRU: Pass the object directly down the pipeline. A CLIP thread is not spawned.
- DISCARD: Remove the object from the pipeline and dereference object vertices as required (i.e., dereferencing will not occur if the vertices are shared with other objects).
- SPAWN: Pass the object to a CLIP thread. In the process of initiating the thread, the object vertices may be dereferenced.

The following logic is used to determine what to do with the object (PASSTHRU or DISCARD or SPAWN).

DevBW-E0,DevCL-B Errata: SPAWN is forced if the object is BAD and ClipMode is not REJECT_ALL

```
//
// Use the ClipMode to determine the action to take
//
switch (CLIP_STATE.ClipMode) {
  case NORMAL: {
    PASSTHRU = TA && !BAD
    DISCARD  = TR || BAD
    SPAWN    = MC && !BAD
  }
  case CLIP_ALL: {
    PASSTHRU = 0
    DISCARD  = 0
    SPAWN    = 1
  }
  case CLIP_NOT_REJECT: {
    PASSTHRU = 0
    DISCARD  = TR || BAD
    SPAWN    = !(TR || BAD)
  }
  case REJECT_ALL: {
    PASSTHRU = 0
    DISCARD  = 1
    SPAWN    = 0
  }
  case ACCEPT_ALL: {
    PASSTHRU = !BAD
    DISCARD  = BAD
    SPAWN    = 0
  }
} endswitch

#ifdef (DevBW-E0 || DevCL-B)
if (BAD && CLIP_STATE.ClipMode != REJECT_ALL) {
    DISCARD = 0
    SPAWN = 1
}
#endif
```

6.5.3.1 NORMAL ClipMode

In NORMAL mode, objects will be discarded if TR or BAD, passed through if TA, and passed to a CLIP thread if MC. Those mode is typically used when the CLIP kernel is only used to perform 3D Clipping (the expected usage model).

6.5.3.2 CLIP_AL L ClipMode

In CLIP_ALL mode, all objects (regardless of classification) will be passed to CLIP threads. Note that this includes BAD objects. This mode can be used to perform arbitrary processing in the CLIP thread, or as a backup if for some reason the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are not sufficient for controlling 3D Clipping.

6.5.3.3 CLIP_NON_REJECT ClipMode

This mode is similar to CLIP_ALL mode, but TR and BAD objects are discarded an all other (TA, MC) objects are passed to CLIP threads. Usage of this mode assumes that the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are sufficient at least in respect to determining trivial reject.

6.5.3.4 REJECT_ALL ClipMode

In REJECT_ALL mode, all objects (regardless of classification) are discarded. This mode effectively clips out all objects.

6.5.3.5 ACCEPT_ALL ClipMode

In ACCEPT_ALL mode, all non-BAD objects are passed directly down the pipeline. This mode partially disables the CLIP stage. BAD objects will still be discarded, and incomplete primitives (generated by a GS thread) will be discarded.

Primitive topologies with adjacency are also handled, in that the adjacent-only vertices are dereferenced and only non-adjacent objects are passed down the pipeline. This condition can arise when primitive topologies with adjacency are generated but the GS stage is disabled. If this condition is allowed, the CLIP stage must not be completely disabled – as this would allow adjacent vertices to pass through the CLIP stage and lead to UNPREDICATBLE results as the rest of the pipeline does not comprehend adjacency.

6.6 Object Pass-Through

Depending on ClipMode, objects may be passed directly down the pipeline. The PrimTopologyType associated with the output objects may differ from the input PrimTopologyType, as shown in the table below.

Programming Note: The CLIP unit does not tolerate primitives with adjacency that have “dangling vertices”. This should not be an issue under normal conditions, as the VF unit will not generate these sorts of primitives and the GS thread is restricted (though by specification only) to not output these sorts of primitives.

Input PrimTopologyType	Pass-Through Output PrimTopologyType	Notes
POINTLIST	POINTLIST	
POINTLIST_BF	POINTLIST_BF	
LINELIST	LINELIST	
LINELIST_ADJ	LINELIST	**Adjacent vertices removed.**
LINESTRIP	LINESTRIP	
LINESTRIP_ADJ	LINESTRIP	**Adjacent vertices removed.**
LINESTRIP_BF	LINESTRIP_BF	
LINESTRIP_CONT	LINESTRIP_CONT	
LINESTRIP_CONT_BF	LINESTRIP_CONT_BF	
LINELOOP	N/A	**Not supported after GS.**
TRILIST	TRILIST	
RECTLIST	RECTLIST	
TRILIST_ADJ	TRILIST	**Adjacent vertices removed.**
TRISTRIP	TRISTRIP or TRISTRIP_REV	**Depends on where the incoming strip is broken (if at all) by discarded or clipped objects** **See Tristrip Clipping Errata subsection.**
TRISTRIP_REV	TRISTRIP or TRISTRIP_REV	**Depends on where the incoming strip is broken (if at all) by discarded or clipped objects** **See Tristrip Clipping Errata subsection.**
TRISTRIP_ADJ	TRISTRIP or TRISTRIP_REV	**Depends on where the incoming strip is broken (if at all) by discarded or clipped objects** **Adjacent vertices removed.** **See Tristrip Clipping Errata subsection.**
TRIFAN	TRIFAN	
TRIFAN_NOSTIPPLE	TRIFAN_NOSTIPPLE	
POLYGON	POLYGON	
QUADLIST	N/A	**Not supported after GS.**
QUADSTRIP	N/A	**Not supported after GS.**

6.7 CLIP Thread Request Generation [Pre-DevSNB]

6.7.1 Object Vertex Ordering

The following table defines the number and order of object vertices passed in the Vertex Data portion of the CLIP thread payload, assuming an input topology with *N* vertices. The ObjectType passed to the thread is, by default, the incoming PrimTopologyType. Exceptions to this rule (for the TRISTRIP variants) are called out.

PrimTopologyType	Order of Vertices in Payload	Notes
<PRIMITIVE_TOPOLOGY>	[<object#>] = (<vert#>,…);	
POINTLIST	[0] = (0); [1] = (1); …; [N-2] = (N-2);	
POINTLIST_BF	Same as POINTLIST	**Handled same as POINTLIST**
LINELIST (N is multiple of 2)	[0] = (0,1); [1] = (2,3); …; [(N/2)-1] = (N-2,N-1)	
LINELIST_ADJ (N is multiple of 4)	[0] = (1,2); [1] = (5,6); …; [(N/4)-1)] = (,N-3,N-2)	**Adjacent vertices immediately dereferenced**
LINESTRIP (N >= 2)	[0] = (0,1); [1] = (1,2); …; [N-2] = (N-2,N-1)	
LINESTRIP_ADJ (N >= 4)	[0] = (1,2); [1] = (2,3); …; [N-4] = (N-3,N-2)	**Adjacent-only vertices immediately dereferenced**
LINESTRIP_BF	Same as LINESTRIP	**Handled same as LINESTRIP**
LINESTRIP_CONT	Same as LINESTRIP	**Handled same as LINESTRIP**
LINESTRIP_CONT_BF	Same as LINESTRIP	**Handled same as LINESTRIP**
LINELOOP	N/A	**Not supported after GS.**
TRILIST (N is multiple of 3)	[0] = (0,1,2); [1] = (3,4,5); …; [(N/3)-1] = (N-3,N-2,N-1)	
RECTLIST	Same as TRILIST	**Handled same as TRILIST**

PrimTopologyType	Order of Vertices in Payload	Notes
TRILIST_ADJ (N is multiple of 6)	[0] = (0,2,4); [1] = (6,8,10); …; [(N/6)-1] = (N-6,N-4,N-2)	**Adjacent vertices immediately dereferenced**
TRISTRIP (N >= 3)	[0] = (0,1,2) {TRISTRIP} [1] = (1,2,3) {TRISTRIP_REVERSE}; … [N-3] = (N-3,N-2,N-1) {TRISTRIP or TRISTRIP_REVERSE}	**"Odd" triangles do not have vertices reordered, though identified as TRISTRIP_REVERSE so the thread knows this**
TRISTRIP_REV (N >= 3)	[0] = (0,1,2) {TRISTRIP_REVERSE} [1] = (1,2,3) {TRISTRIP}; …; [N-3] = (N-3,N-2,N-1) {TRISTRIP or TRISTRIP_REVERSE}	**"Odd" triangles do not have vertices reordered, though identified as TRISTRIP so the thread knows this**
TRISTRIP_ADJ (N even, N >= 6)	[0] = (0,2,4) {TRISTRIP}; [1] = (2,4,6) {TRISTRIP_REVERSE}; …; [k even] = (2k,2k+2,2k+4) {TRISTRIP}; [k odd] = (2k,2k+2,2k+4) {TRISTRIP_REVERSE};…; [(N/2)-3, even] = (N-6,N-4,N-2) {TRISTRIP}; [(N/2)-3, odd] = (N-6,N-4,N-2) {TRISTRIP_REVERSE};	**"Odd" triangles do not have vertices reordered, though identified as TRISTRIP_REVERSE so the thread knows this.** **Adjacent vertices immediately dereferenced**
TRIFAN (N > 2)	[0] = (0,1,2); [1] = (0,2,3); …; [N-3] = (0, N-2, N-1);	**Only used by OGL**
TRIFAN_NOSTIPPLE	Same as TRIFAN	
POLYGON	Same as TRIFAN	
QUADLIST	N/A	**Not supported after GS.**
QUADSTRIP	**N/A**	**Not supported after GS.**

6.7.2 CLIP Thread Payload

Table 6-1 shows the layout of the payload delivered to CLIP threads.

Refer to 3D Pipeline Stage Overview (*3D Overview*) for details on those fields that are common amongst the various pipeline stages.

Table 6-1. CLIP Thread Payload

DWord	Bit	Description
R0.7	31	**Reserved**
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID.** This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer.** Specifies the location of the Scratch Space allocated to this thread, as an 1KB-aligned offset from the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	Reserved
	7:0	**FFTID.** This ID is assigned by the fixed function unit and is a relative identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for Implementation Use
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:4	Reserved
	3:0	**Per Thread Scratch Space.** Specifies the amount of Scratch Space allocated to this thread, as a power of 2 bytes in excess of 1KB. Format = U4 Range = [0,11] indicating [1KB, 2MB] in powers of two
R0.2	31	**Object Outcode [VP.XMin].** This bit contains the logical OR of the VP.XMin vertex outcode over all the vertices of the object. It can be used as a hint to the 3D Clip algorithm (if zero, the object vertices are all on the visible side of the VP.XMin clip plane). **Pre-DevBW-E0,DevCL-A Errata:** All VP and GB ObjectOutcodes are UNDEFINED if any vertex of the object has a negative RHW component. See W Clipping Errata sections above for more information.
	30	**Object Outcode [VP.XMax]**
	29	**Object Outcode [VP.YMin]**
	28	**Object Outcode [VP.YMax]**

DWord Bit		Description
	27	**Object Outcode [VP.ZMin]**
	26	**Object Outcode [VP.ZMax]**
	25	**Object Outcode [GB.XMin].** This bit contains the logical OR of the GB.XMin vertex outcode over all the vertices of the object. It can be used as a hint to the 3D Clip algorithm (if zero, the object vertices are all on the visible side of the GB.XMin clip plane).
	24	**Object Outcode [GB.XMax]**
	23	**Object Outcode [GB.YMin]**
	22	**Object Outcode [GB.YMax]**
	21	**Object Outcode [UserClip7].** This bit contains the logical OR of the UserClip7 vertex outcode over all the vertices of the object. It can be used as a hint to the 3D Clip algorithm (if zero, the object vertices are all on the visible side of the UserClip7 clip plane). **Pre-DevBW-E0,DevCL-B Errata:** This bit is the logical OR of the NEGW outcodes over all the vertices of the object.
	20	**Object Outcode [UserClip6]**
	19	**Object Outcode [UserClip5]**
	18	**Object Outcode [UserClip4]**
	17	**Object Outcode [UserClip3]**
	16	**Object Outcode [UserClip2]**
	15	**Object Outcode [UserClip1]**
	14	**Object Outcode [UserClip0]**
	13	**Object Outcode [BAD]** **Note:** If set, all VP and GB-related Object Outcodes are UNDEFINED.
	12	**DevCTG+: Object Outcode [NEGW]** **Otherwise:** Reserved
	11:10	Reserved
	9	**Edge Indicator [1].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex2 to Vertex0 is an exterior edge of the polygon (i.e., this is the last or only triangle of the polygon). If clear, that edge is an interior edge. The CLIP kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved 0: V2→V0 is not an outside edge 1: V2→V0 is an outside edge

DWord	Bit	Description
	8	**Edge Indicator [0].** For POLYGON primitive objects, this bit indicates whether the edge from Vertex0 to Vertex1 is an exterior edge of the polygon (i.e., this is the first or only triangle of the polygon). If clear, that edge is an interior edge. The CLIP kernel can use this bit to control operations such as generating wireframe representations of polygon primitives. For all other Primitive Topology Types, this bit is Reserved 0: V0→V1 is not an outside edge 1: V0→V1 is an outside edge
	7:5	Reserved
	4:0	**Primitive Topology Type.** This field identifies the "basic" Primitive Topology Type associated with the primitive spawning this object. It indirectly specifies the number of input vertices included in the thread payload. Note that the CLIP unit may toggle this value between TRISTRIP and TRISTRIP_REV, as described in 6.7.1. Also, as part of adjacency removal, the CLIP unit will convert topology types with adjacency to the corresponding no-adjacency topology type (e.g., an incoming LINELIST_ADJ primitive will cause LINELIST – and not LINELIST_ADJ – to be passed in this field. Format: (See 3DPRIMITIVE command in *3D Pipeline*)
R0.1	31:0	Reserved
R0.0	31:23	Reserved
	22:16	**[Pre-DevILK]: Handle ID.** This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. Format: Reserved for Implementation Use [**DevILK+**]: Reserved
	15:9	Reserved
	8:0	**[Pre-DevILK]: URB Return Handle.** This is the initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the first destination URB entry. Format: U9 URB Handle [**DevILK+**]: Reserved
[Varies] optional	31:0	**Constant Data** (optional). Some amount of constant data (possible none) can be extracted from the URB and passed to the thread following the R0 Header. The data is read from the Constant URB Entry at some offset (**Constant URB Entry Read Offset** state) from the handle. The amount of data provided is defined by the **Constant URB Entry Read Length** state.
Varies	31:0	**Vertex Data.** There can be up to 3 vertices supplied, each with a size defined by the **Vertex URB Entry Read Length** state. The amount of data provided for each vertex is defined by the **Vertex URB Entry Read Length** state Vertex 0 DWord 0 is located at Rn.0, Vertex 0 DWord 1 is located at Rn.1, etc. Vertex 1 DWord 0 immediately follows the last DWord of Vertex 0, and so on. See Object Vertex Ordering (above) for a definition of the number and order of vertices passed in the payload.

6.8 CLIP Thread Execution [Pre-DevSNB]

A CLIP kernel can perform arbitrary operations on the input object. Input data is either passed directly in the thread payload (including the input object vertex data) or indirectly via pointers passed in the payload. It is anticipated that the CLIP kernel implement a 3D clipping algorithm though this is not strictly required. Definition of candidate algorithms is beyond the scope of this document.

[**Dev ILK+**]: Concurrent clip threads must use the FF_SYNC message (URB shared function) to request an initial VUE handle and synchronize output of VUEs to the pipeline (see *URB* in *Shared Functions*). Only two clip threads can be outputting VUEs to the pipeline at a time. In order to achieve parallelism, clip threads should perform the clipping algorithm (along with any other required functions) and buffer results (either in the GRF or scratch memory) before issuing the FF_SYNC message. The issuing clip thread will be stalled on the FF_SYNC writeback until it is that thread's turn to output VUEs. As only two threads outputs VUEs at a time, the post-FF_SYNC output portion of the kernel should be optimized as much as possible to maximize parallelism.

Refer to 3D Pipeline Stage Overview (*3D Overview*) for further information on FF-unit/Thread interactions.

6.8.1 Vertex Output

The CLIP thread can output a number (possibly zero) of destination VUEs. Refer to Thread Output Handling (*3D Overview*).

A GS or CLIP thread is restricted as to the number of URB handles it can retain. Here a "retained" handle refers to a URB handle that (a) has been pre-allocated or allocated and returned to the thread via the **Allocate** bit in the URB_WRITE message, and (b) has yet to be returned to the pipeline via the **Complete** bit in the URB_WRITE message.

- [**Pre-DevILK**]: When operating in single-thread mode (**Maximum Number of Threads** == 1), the number of retained handles must not exceed min(16, **Number of URB Entries**).
- [**Pre-DevILK**]: When operating in dual-thread mode (**Maximum Number of Threads** == 2), the number of retained handles must not exceed (**Number of URB Entries**/2).
- [**DevILK+**]: The number of retained handles must not exceed min(32, **Number of URB Entries**).

This restriction is not expected to be significant in that most/all GS/CLIP threads are expected to retain only a few (<=4) handles.

6.8.2 Thread Termination

CLIP threads must signal thread termination by issuing a URB_WRITE message to the URB shared function with the **EOT** and **Complete** bits set.

6.9 Thread-Generated Vertex Readback [Pre-DevSNB]

The CLIP unit performs a readback of the Vertex Header of each vertex output by a CLIP thread, as this information is required by the next stage (SF). Note that trivially-accepted vertices (not generated by a CLIP thread) already have been readback in the GS stage. See *Vertex Data Overview* for a description of the Vertex Header fields and how they are read-back and used by the CLIP unit.

The CLIP unit will extract the following per-vertex readback data and associate it with the generated vertex as it is sent down the pipeline:

- PrimTopologyType
- PrimStart
- PrimEnd
- Viewport Index
- RenderTarget Array Index
- PointWidth
- Vertex Position X,Y,Z,RHW (NDC coordinates only)

Note that the UserClipFlags are not read back as they are not relevant past the clip stage.

6.10 Primitive Output

(This section refers to output from the CLIP unit to the pipeline, not output from the CLIP thread)

The CLIP unit will output primitives (either passed-through or generated by a CLIP thread) in the proper order. This includes the buffering of a concurrent CLIP thread's output until the preceding CLIP thread terminates. Note that the requirement to buffer subsequent CLIP thread output until the preceding CLIP thread terminates has ramifications on determining the number of VUEs allocated to the CLIP unit and the number of concurrent CLIP threads allowed.

6.11 Other Functionality

6.11.1 Statistics Gathering

The CLIP unit includes logic to assist in the gathering of certain pipeline statistics, primarily in support of the Asynchronous Query function of the APIs. The statistics take the form of MI counter registers (see *Memory Interface Registers*), where the CLIP unit provides signals causing those counters to increment.

Software is responsible for controlling (enabling) these counters in order to provide the required statistics at the DDI level. For example, software might need to disable the statistics gathering before submitting non-API-visible objects (e.g., RECTLISTs) for processing.

The CLIP unit must be ENABLED (via the **CLIP Enable** bit of PIPELINED_STATE_POINTERS) in order to it to affect the statistics counters. This might lead to a pathological case where the CLIP unit needs to be ENABLED simply to provide statistics gathering. If no clipping functionality is desired, **Clip Mode** can be set to ACCEPT_ALL to effectively inhibit clipping while leaving the CLIP stage ENABLED.

The two statistics the CLIP unit affects (if enabled) are:

- CL_INVOCATION_COUNT:
    - [**Pre-DevCTG**]: Incremented for every CLIP thread spawned.
    - [**DevCTG**]: Incremented under kernel program control via the **Increment CL_INVOCATIONS** bit (M0.2<7>) of the URB_WRITE message header.
    - **[Dev ILK]** Incremented for every object received from the GS stage.
- [Pre-DevILK]: GS_PRIMITIVES_COUNT: Incremented for every object received from the GS stage.

[Pre-DevILK]: Implementation Note: The reason the CLIP unit counts GS-produced objects (and, similarly, why the SF unit counts CLIP-produced objects) is that a downstream Object Staging Buffer is an opportunistic place to count objects generated by threads in an upstream unit.

6.11.1.1 CL_INVOCATION_COUNT [Pre-DevCTG]

If the **Statistics Enable** bit (CLIP_STATE) is set, the CLIP unit increments the CL_INVOCATION_COUNT register each time a CLIP thread is spawned.

6.11.1.2 CL_INVOCATION_COUNT [DevCTG]

In DevCTG, the CL_INVOCATION_COUNT is incremented under GS kernel control via the **Increment CL_INVOCATIONS** bit (Bit M0.2<7>) of the URB_WRITE message header. However, the **Statistics Enable** bit (CLIP_STATE) must also be set to enable the increment.

6.11.1.3 CL_INVOCATION _COUNT [DevILK+]

If the **Statistics Enable** bit (CLIP_STATE) is set, the CLIP unit increments the CL_INVOCATION_COUNT register for every complete object received from the GS stage.

In order to maintain a count of application-generated objects, software will need to clear the CLIP unit's **Statistic Enable** whenever driver-generated objects are rendered.

6.11.1.4 GS_PRIMITIVES_ COUNT [Pre-DevILK]

If **GS Output Object Statistic Enable** is set (CLIP_STATE), the CLIP stage increments GS_PRIMITIVES_COUNT for every complete object received from the GS stage.

In order to maintain a count of objects generated by the API's Geometry Shader function (presumably the number of objects output by GS threads), software will need to clear the CLIP unit's **GS Output Object Statistic Enable** whenever the GS unit is DISABLED.

7. Strips and Fans (SF) Stage

7.1 Overview

The Strips and Fan (SF) stage of the GENx 3D pipeline is responsible for performing "setup" operations required to rasterize 3D objects.

[Pre-DevSNB]: This functionality is split between fixed-function hardware in the SF unit and SF (aka Setup) threads spawned to compute plane equations required for attribute interpolation.

Inputs from CLIP

The following table describes the per-vertex inputs passed to the SF unit from the previous (CLIP) stage of the pipeline.

Table 7-1. SF's Vertex Pipeline Inputs

Variable	Type	Description
primType	enum	Type of primitive topology the vertex belongs to. See Table 7-2 for a list of primitive types supported by the SF unit. See *3D Pipeline* for descriptions of these topologies. Notes: The CLIP unit will convert any primitive with adjacency (3DPRIMxxx_ADJ) it receives from the pipeline into the corresponding primitive without adjacency (3DPRIMxxx). QUADLIST, QUADSTRIP, LINELOOP primitives are not supported by the SF unit. Software must use a GS thread to convert these to some other (supported) primitive type. **[Dev ILK]** If an object is clipped by the hardware clipper, the CLunit would force this field to "3DPRIM_POLYGON". SFunit would process this incoming object just as it would any other "3DPRIM_POLYGON". SFunit selects vertex 0 as the provoking vertex.
primStart,primEnd	boolean	Indicate vertex's position within the primitive topology
vInX[]	float	Vertex X position (screen space or NDC space)
vInY[]	float	Vertex Y position (screen space or NDC space)
vInZ[]	float	Vertex Z position (screen space or NDC space)
vInInvW[]	float	Reciprocal of Vertex homogeneous (clip space) W
hVUE[]	URB address	Points to the vertex's data stored in the URB (one VUE handle per vertex)

Variable	Type	Description
renderTargetArrayIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in subsequent processing.
viewPortIndex	uint	Index of a viewport transform matrix within the SF_VIEWPORT structure used to perform Viewport Transformation on object vertices and scissor operations on an object. If this vertex is the leading vertex of an object within the primitive topology, this value will be associated with that object in the Viewport Transform and Scissor subfunctions, otherwise the value is ignored. Note that for primitive topologies with vertices shared between objects, this means a shared vertex may be subject to multiple Viewport Transformation operations if the viewPortIndex varies within the topology.
pointSize	uint	If this vertex is within a POINTLIST[_BF] primitive topology, this value specifies the screen space size (width,height) of the square point to be rasterized about the vertex position. Otherwise the value is ignored.

7.1.1 Attribute Setup/Interpolation Process

7.1.1.1 [Pre-DevSNB]

Required inputs to API Pixel Shader programs are the pixel's position and interpolated vertex attributes sampled at the pixel position. In order to produce these per-pixel parameters, certain setup calculations need to be performed within the SF stage to provide inputs for the subsequent interpolation process at in the WM stage. Where and how the setup calculations and interpolation are performed varies by attribute (due to various cost/performance tradeoffs), as outlined below:

- **Position X,Y**: The SF unit performs the position X,Y setup computations in fixed function hardware and passes these results directly to the WM stage. The WM unit interpolates position (i.e., rasterizes the object) in fixed-function hardware and passes pixel X,Y information to the WM (PS) thread in the thread's payload.
- **Position Z**: The handling of the position Z attribute is more complicated. The SF unit performs operations to compute Z at object vertices (e.g., viewport map, etc.). The object vertex's position Z values are then passed to the SF (Setup) thread in the fixed header portion of the thread payload. The SF thread is responsible for performing the setup computations for position Z and storing the required result values (plane equation coefficients) in the PUE. Subsequently the WM unit will directly read the position Z plane equation coefficients from the PUE (at a location programmed via WM_STATE). The WM unit will then perform interpolation of position Z (along with some other computations like Depth Offset, etc.) to derive per-pixel position Z. This value is then used for Early Depth Test, if applicable. The per-pixel position Z value ("source depth") can be optionally included in WM thread payloads for use by the thread.

- **Position 1/W**: This attribute could be handled like “other vertex attributes”, but as an optimization it is treated slightly differently. The position 1/W values of the object vertices are passed from the SF unit to the SF thread in the fixed header portion of the thread payload. These values are unmodified copies of the postion 1/W values read back from the object’s VUEs – so in theory the SF thread could use the values from the VUEs like it does for other attributes. However, having the SF unit insert them into the payload allows software to avoid having the 256-bit Vertex Headers read from the VUEs and placed in the SF thread payload, thus removing this traffic from the thread dispatch process. The SF thread performs setup computations on the position 1/W attributes and stores the results in the output PUE. Subsequently, the WM thread will use the position 1/W setup parameters to interpolate position 1/W to the pixel location. This is likely one of the first operations in the PS thread, as the pixel’s interpolated 1/W value is required to perform perspective-correct interpolation of other vertex attributes.
- **Other Vertex Attributes**: The handling of non-position vertex attributes (e.g., texture coordinates, colors, etc.) is straightforward. The SF unit is not directly involved with the setup computations for these attributes, and the WM unit is not directly involved with their interpolation. The SF thread will use the object’s vertex attributes provided in the VUE data in the thread payload, perform the setup computations as required, and store the results in the output PUE. The WM thread will use this PUE data to interpolate the attributes to the pixel location.

7.1.2 Outputs to WM

The outputs from the SF stage to the WM stage are mostly comprised of implementation-specific information required for the rasterization of objects. The type of information is summarized below, but as the interface is not exposed to software a detailed discussion is not relevant to this specification.

- PrimType of the object
- VPIndex, RTAIndex associated with the object
- **[Pre-DevSNB]:** Handle of the Primitive URB Entry (PUE) that was written by the SF (Setup) thread. This handle will be passed to all WM (PS) threads spawned from the WM’s rasterization process.
- Information regarding the X,Y extent of the object (e.g., bounding box, etc.)
- Edge or line interpolation information (e.g., edge equation coefficients, etc.)
- Information on where the WM is to start rasterization of the object
- Object orientation (front/back-facing)
- Last Pixel indication (for line drawing)

7.2 Primitive Assembly

The first subfunction within the SF unit is *Primitive Assembly*. Here 3D primitive vertex information is buffered and, when a sufficient number of vertices are received, converted into basic 3D objects which are then passed to the Viewport Transformation subfunction.

The number of vertices passed with each primitive is constrained by the primitive type and must conform to Table 7-2. Passing any other number of vertices results in UNDEFINED behavior. Note that this restriction only applies to primitive output by GS threads (which are under control of the GS kernel). See the Vertex Fetch chapter for details on how the VF unit automatically removes incomplete objects resulting from processing a 3DPRIMITIVE command.

Table 7-2. SF-Supported Primitive Types & Vertex Count Restrictions

primType	VertexCount Restriction
3DPRIM_TRILIST	nonzero multiple of 3
3DPRIM_TRISTRIP 3DPRIM_TRISTRIP_REVERSE	>=3
3DPRIM_TRIFAN 3DPRIM_TRIFAN_NOSTIPPLE 3DPRIM_POLYGON	>=3
3DPRIM_LINELIST	nonzero multiple of 2
3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	>=2
3DPRIM_RECTLIST	nonzero multiple of 3
3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF	nonzero

The 3D object types are listed in Table 7-3.

Table 7-3. 3D Object Types

objectType	generated by *primType*	Vertices/Object
3DOBJ_POINT	3DPRIM_POINTLIST 3DPRIM_POINTLIST_BF	1
3DOBJ_LINE	3DPRIM_LINELIST 3DPRIM_LINESTRIP 3DPRIM_LINESTRIP_CONT 3DPRIM_LINESTRIP_BF 3DPRIM_LINESTRIP_CONT_BF	2
3DOBJ_TRIANGLE	3DPRIM_TRILIST 3DPRIM_TRISTRIP 3DPRIM_TRISTRIP_REVERSE 3DPRIM_TRIFAN 3DPRIM_TRIFAN_NOSTIPPLE 3DPRIM_POLYGON	3
3DOBJ_RECTANGLE	3DPRIM_RECTLIST	3 (expanded to 4 in RectangleCompletion)

The outputs of Primitive Decomposition are listed in Table 7-4.

Table 7-4. Primitive Decomposition Outputs

Variable	Type	Description
objectType	enum	Type of object. See Table 7-3
nV	uint	The number of object vertices passed to Object Setup. See Table 7-3
v[0..nV-1]*	various	Data arrays associated with object vertices. Data in the array consists of X, Y, Z, invW and a pointer to the other vertex attributes. These additional attributes are not used by directly by the 3D fixed functions but are made available to the SF thread. The number of valid vertices depends on the object type. See Table 7-3
invertOrientation	enum	Indicates whether the orientation (CW or CCW winding order) of the vertices of a triangle object should be inverted. Ignored for non-triangle objects.
backFacing	enum	Valid only for points and line objects, indicates a back facing object. This is used later for culling.
provokingVtx	uint	Specifies the index (into the *v[]* arrays) of the vertex considered the “provoking” vertex (for flat shading). The selection of the provoking vertex is programmable via SF_STATE (**xxx Provoking Vertex Select** state variables.)

Variable	Type	Description
polyStippleEnable	boolean	TRUE if Polygon Stippling is enabled. FALSE for TRIFAN_NOSTIPPLE. Ignored for non-triangle objects.
continueStipple	boolean	Only applies to line objects. TRUE if Line Stippling should be continued (i.e., not reset) from where the previous line left off. If FALSE, Line Stippling is reset for each line object.
renderTargetIndex	uint	Index of the render target (array element or 3D slice), clamped to 0 by the GS unit if the max value was exceeded. This value is simply passed in SF thread payloads and not used within the SF unit.
viewPortIndex	uint	Index of a viewport transform matrix within the SF_VIEWPORT structure used to perform Viewport Transformation on object vertices and scissor operations on an object.
pointSize	unit	For point objects, this value specifies the screen space size (width,height) of the square point to be rasterized about the vertex position. Otherwise the value is ignored.

The following table defines, for each primitive topology type, which vertex's VPIndex/RTAIndex applies to the objects within the topology.

Table 7-5. VPIndex/RTAIndex Selection

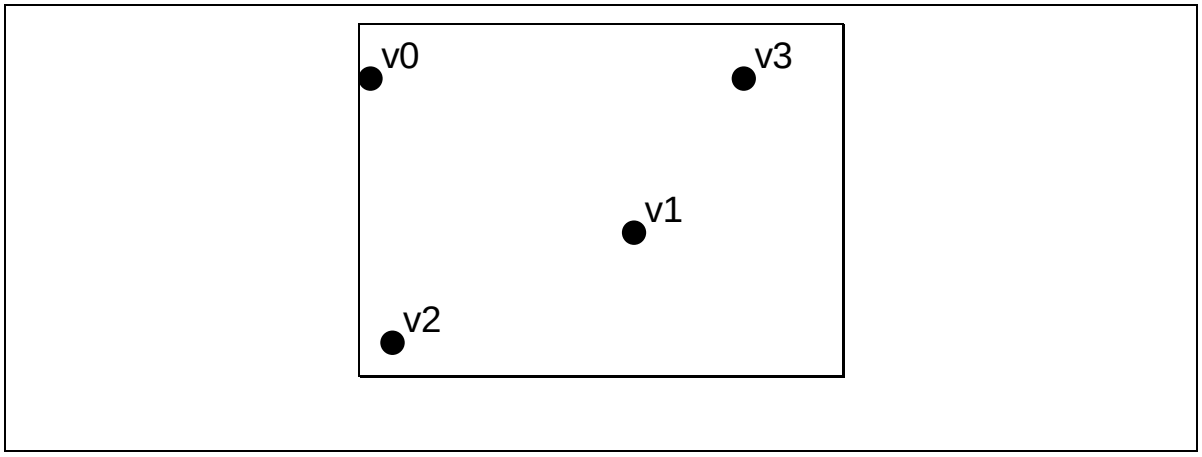
PrimTopologyType	Viewport Index Usage
POINTLIST POINTLIST_BF	Each vertex supplies the VPIndex for the corresponding point object
LINELIST	The leading vertex of each line supplies the VPIndex for the corresponding line object. V0.VPIndex → Line(V0,V1) V2.VPIndex → Line(V2,V3) …
LINESTRIP LINESTRIP_BF LINESTRIP_CONT LINESTRIP_CONT_BF	The leading vertex of each line segment supplies the VPIndex for the corresponding line object. V0.VPIndex → Line(V0,V1) V1.VPIndex → Line(V1,V2) … NOTE: If the VPIndex changes within the topology, shared vertices will be processed (mapped) multiple times.
TRILIST RECTLIST	The leading vertex of each triangle/rect supplies the VPIndex for the corresponding triangle/rect objects. V0.VPIndex → Tri(V0,V1,V2) V3.VPIndex → Tri(V3,V4,V5) …
TRISTRIP TRISTRIP_REVERSE	The leading vertex of each triangle supplies the VPIndex for the corresponding triangle object.

PrimTopologyType	Viewport Index Usage
	V0.VPIndex → Tri(V0,V1,V2) V1.VPIndex → Tri(V1,V2,V3) … NOTE: If the VPIndex changes within the primitive, shared vertices will be processed (mapped) multiple times.
TRIFAN TRIFAN_NOSTIPPLE POLYGON	The first vertex (V0) supplies the VPIndex for all triangle objects.

7.2.1 Point List Decomposition

The 3DPRIM_POINTLIST and 3DPRIM_POINTLIST_BACKFACING primitives specify a list of independent points.

Figure 7-1. 3DPRIM_POINTLIST Primitive



The decomposition process divides the list into a series of basic 3DOBJ_POINT objects that are then passed individually and in order to the Object Setup subfunction. The *provokingVertex* of each object is, by definition, v[0].

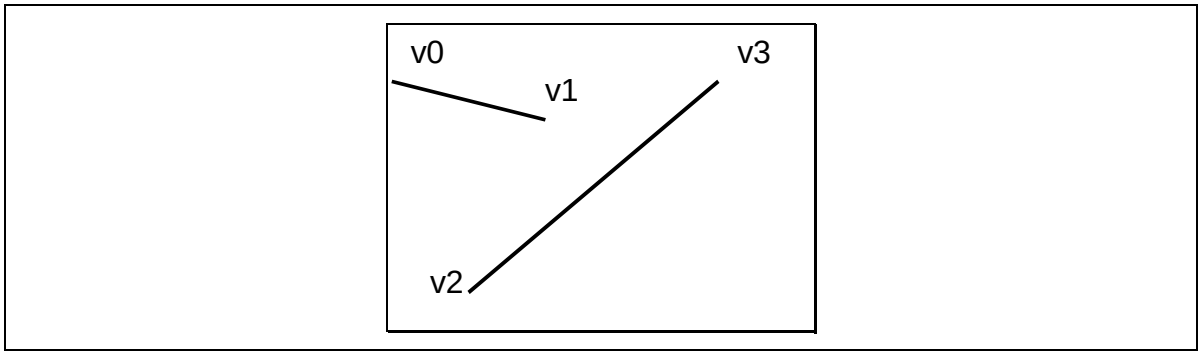
Points have no winding order, so the primitive command is used to explicitly state whether they are back-facing or front-facing points. Primitives of type 3DPRIM_POINTLIST_BACKFACING are decomposed exactly the same way as 3DPRIM_POINTLIST primitives, but the *backFacing* variable is set for resulting point objects being passed on to object setup.

```
PointListDecomposition() {
       objectType = 3DOBJ_POINT
       nV = 1
       provokingVtx = 0
       if (primType == 3DPRIM_POINTLIST)
              backFacing = FALSE
       else  // primType == 3DPRIM_POINTLIST_BACKFACING
              backFacing = TRUE
       for each (vertex I in [0..vertexCount-1]) {
              v[0] ← vIn[i]             // copy all arrays (e.g., v[]X, v[]Y,
etc.)
              ObjectSetup()
       }
}
```

7.2.2 Line List Decomposition

The 3DPRIM_LINELIST primitive specifies a list of independent lines.

Figure 7-2. 3DPRIM_LINELIST Primitive



The decomposition process divides the list into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0, v1; v2, v3; and so on. The *provokingVertex* of each object is taken from the **Line List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

```
LineListDecomposition() {
       objectType = 3DOBJ_LINE
       nV = 2
       provokingVtx = Line List/Strip Provoking Vertex Select
       continueStipple = FALSE
       for each (vertex I in [0..vertexCount-2] by 2) {
              v[0] arrays ← vIn[i] arrays
              v[1] arrays ← vIn[i+1] arrays
              ObjectSetup()
       }
}
```

7.2.3 Line Strip Decomposition

The 3DPRIM_LINESTRIP, 3DPRIM_LINESTRIP_CONT, 3DPRIM_LINESTRIP_BF, and 3DPRIM_LINESTRIP_CONT_BF primitives specify a list of connected lines.

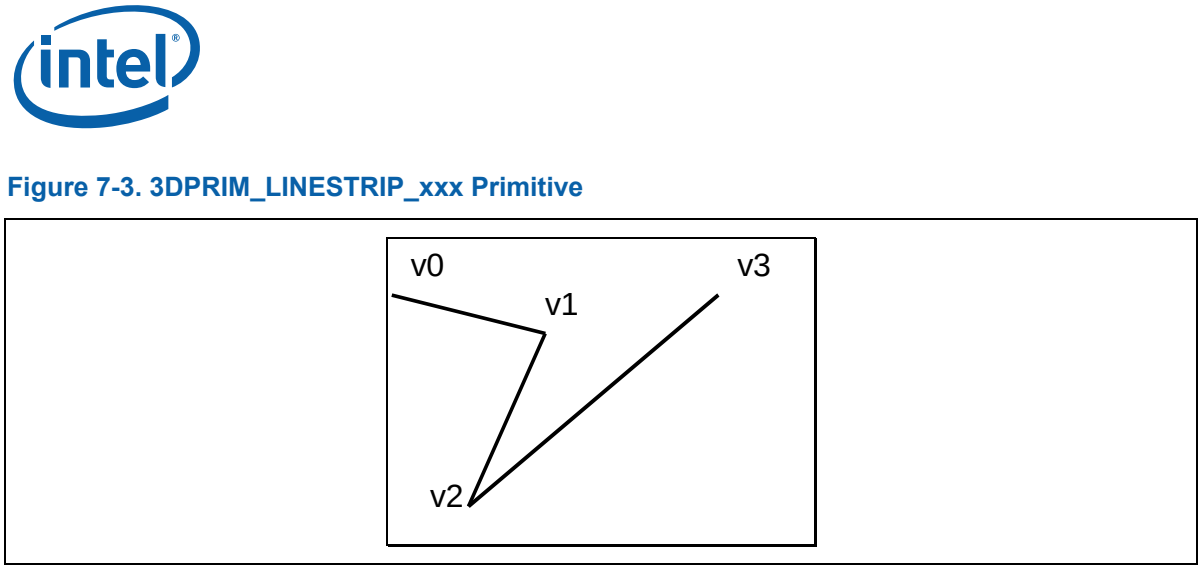


Figure 7-3. 3DPRIM_LINESTRIP_xxx Primitive

The decomposition process divides the strip into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0,v1; v1,v2; and so on. The *provokingVertex* of each object is taken from the **Line List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

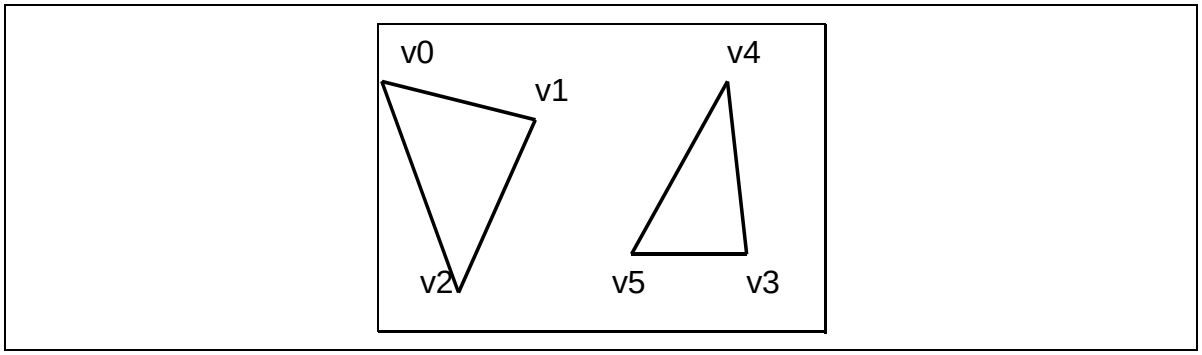
Lines have no winding order, so the primitive command is used to explicitly state whether they are back-facing or front-facing lines. Primitives of type 3DPRIM_LINESTRIP[_CONT]_BF are decomposed exactly the same way as 3DPRIM_LINESTRIP[_CONT] primitives, but the *backFacing* variable is set for the resulting line objects being passed on to object setup. Likewise 3DPRIM_LINESTRIP_CONT[_BF] primitives are decomposed identically to basic line strips, but the *continueStipple* variable is set to true so that the line stipple pattern will pick up from where it left off with the last line primitive, rather than being reset.

```
LineStripDecomposition() {
       objectType = 3DOBJ_LINE
       nV = 2
       provokingVtx = Line List/Strip Provoking Vertex Select
       if (primType == 3DPRIM_LINESTRIP) {
              backFacing = FALSE
              continueStipple = FALSE
       } else if (primType == 3DPRIM_LINESTRIP_BF) {
              backFacing = TRUE
              continueStipple = FALSE
       } else if (primType == 3DPRIM_LINESTRIP_CONT) {
              backFacing = FALSE
              continueStipple = TRUE
       } else if (primType == 3DPRIM_LINESTRIP_CONT_BF) {
              backFacing = TRUE
              continueStipple = TRUE
       }
       for each (vertex I in [0..vertexCount-1]) {
              v[0] arrays ← vIn[i] arrays
              v[1] arrays ← vIn[i+1] arrays
              ObjectSetup()
              continueStipple = TRUE
       }
}
```

7.2.4 Triangle List Decomposition

The 3DPRIM_TRILIST primitive specifies a list of independent triangles.

Figure 7-4. 3DPRIM_TRILIST Primitive



The decomposition process divides the list into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v3,v4,v5; and so on. The *provokingVertex* of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

```
TriangleListDecomposition() {
       objectType = 3DOBJ_TRIANGLE
       nV = 3
       invertOrientation = FALSE
       provokingVtx = Triangle List/Strip Provoking Vertex Select
       polyStippleEnable = TRUE
       for each (vertex I in [0..vertexCount-3] by 3) {
              v[0] arrays ← vIn[i] arrays
              v[1] arrays ← vIn[i+1] arrays
              v[2] arrays ← vIn[i+2] arrays
              ObjectSetup()
       }
}
```

7.2.5 Triangle Strip Decomposition

The 3DPRIM_TRISTRIP and 3DPRIM_TRISTRIP_REVERSE primitives specify a series of triangles arranged in a strip, as illustrated below.

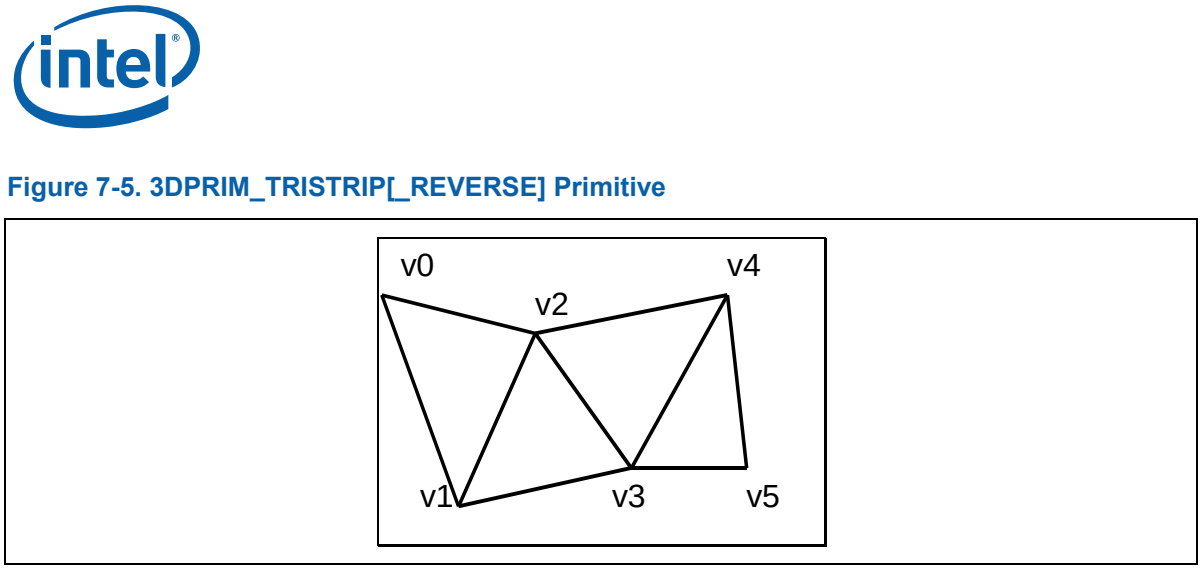


Figure 7-5. 3DPRIM_TRISTRIP[_REVERSE] Primitive

The decomposition process divides the strip into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v1,v2,v3; v2,v3,v4; and so on. Note that the *winding order* of the vertices alternates between CW (clockwise), CCW (counter-clockwise), CW, etc. The *provokingVertex* of each object is taken from the **Triangle List/Strip Provoking Vertex Select** state variable, as programmed via SF_STATE.

The 3D pipeline uses the winding order of the vertices to distinguish between front-facing and back-facing triangles. Therefore, the 3D pipeline must account for the alternation of winding order in strip triangles. The *invertOrientation* variable is generated and used for this purpose.

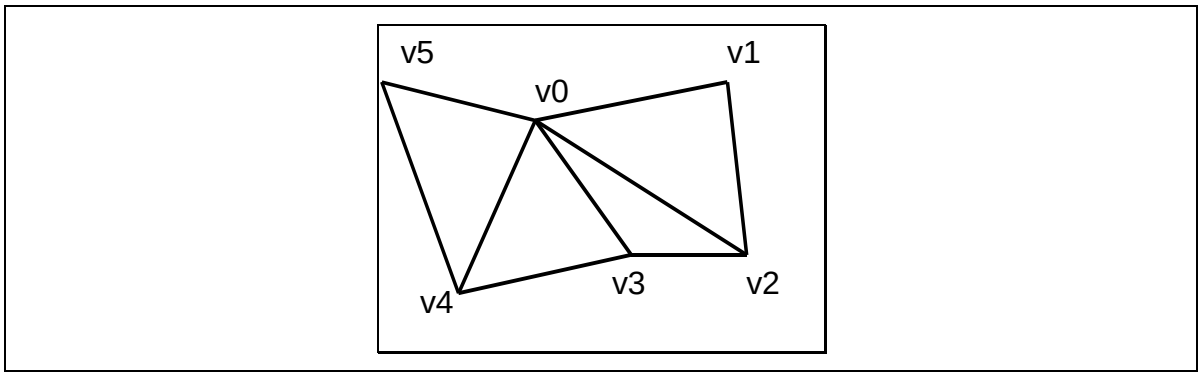
To accommodate the situation where the driver is forced to break an input strip primitive into multiple tristrip primitive commands (e.g., due to ring or batch buffer size restrictions), two tristrip primitive types are supported. 3DPRIM_TRISTRIP is used for the initial section of a strip, and wherever a continuation of a strip starts with a triangle with a CW winding order. 3DPRIM_TRISTRIP_REVERSE is used for a continuation of a strip that starts with a triangle with a CCW winding order.

```
TriangleStripDecomposition() {
       objectType = 3DOBJ_TRIANGLE
       nV = 3
       provokingVtx = Triangle List/Strip Provoking Vertex Select
       if (primType == 3DPRIM_TRISTRIP)
              invertOrientation = FALSE
       else  // primType == 3DPRIM_TRISTRIP_REVERSE
              invertOrientation = TRUE
       polyStippleEnable = TRUE
       for each (vertex I in [0..vertexCount-3]) {
              v[0] arrays ← vIn[i] arrays
              v[1] arrays ← vIn[i+1] arrays
              v[2] arrays ← vIn[i+2] arrays
              ObjectSetup()
              invertOrientation = ! invertOrientation
       }
}
```

7.2.6 Triangle Fan Decomposition

The 3DPRIM_TRIFAN and 3DPRIM_TRIFAN_NOSTIPPLE primitives specify a series of triangles arranged in a fan, as illustrated below.

Figure 7-6. 3DPRIM_TRIFAN Primitive



The decomposition process divides the fan into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v0,v2,v3; v0,v3,v4; and so on. As there is no alternation in the vertex winding order, the *invertOrientation* variable is output as FALSE unconditionally. The *provokingVertex* of each object is taken from the **Triangle Fan Provoking Vertex** state variable, as programmed via SF_STATE.

Primitives of type 3DPRIM_TRIFAN_NOSTIPPLE are decomposed exactly the same way, except the *polyStippleEnable* variable is FALSE for the resulting objects being passed on to object setup. This will inhibit polygon stipple for these triangle objects.

```
TriangleFanDecomposition() {
       objectType = 3DOBJ_TRIANGLE
       nV = 3
       invertOrientation = FALSE
       provokingVtx = Triangle Fan Provoking Vertex Select
       if (primType == 3DPRIM_TRIFAN)
             polyStippleEnable = TRUE
       else  // primType == 3DPRIM_TRIFAN_NOSTIPPLE
             polyStippleEnable = FALSE
       v[0] arrays ← vIn[0] arrays      // the 1st vertex is common
       for each (vertex I in [1..vertexCount-2]) {
             v[1] arrays ← vIn[i] arrays
             v[2] arrays ← vIn[i+1] arrays
             ObjectSetup()
       }
}
```

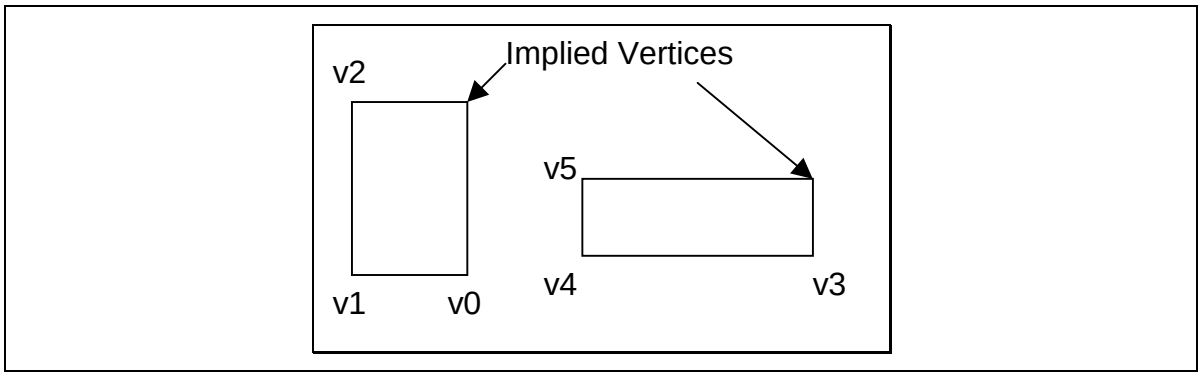
7.2.7 Pol ygon Decomposition

The 3DPRIM_POLYGON primitive is identical to the 3DPRIM_TRIFAN primitive with the exception that the *provokingVtx* is overridden with 0. This support has been added specifically for OpenGL support, avoiding the need for the driver to change the provoking vertex selection when switching between trifan and polygon primitives.

7.2.8 Rectangle List Decomposition

The 3DPRIM_RECTLIST primitive command specifies a list of independent, axis-aligned rectangles. Only the lower right, lower left, and upper left vertices (in that order) are included in the command – the upper right vertex is derived from the other vertices (in Object Setup).

Figure 7-7. 3DPRIM_RECTLIST Primitive



The decomposition of the 3DPRIM_RECTLIST primitive is identical to the 3DPRIM_TRILIST decomposition, with the exception of the *objectType* variable.

```
RectangleListDecomposition() {
       objectType = 3DOBJ_RECTANGLE
       nV = 3
       invertOrientation = FALSE
       provokingVtx = 0
       for each (vertex I in [0..vertexCount-3] by 3) {
              v[0] arrays ← vIn[i] arrays
              v[1] arrays ← vIn[i+1] arrays
              v[2] arrays ← vIn[i+2] arrays
              ObjectSetup()
       }
}
```

7.3 Object Setup

The Object Setup subfunction of the SF stage takes the post-viewport-transform data associated with each vertex of a basic object and computes various parameters required for scan conversion. This includes generation of implied vertices, translations and adjustments on vertex positions, and culling (removal) of certain classes of objects. The final object information is passed to the Windower/Masker (WM) stage where the object is rasterized into pixels.

7.3.1 Invalid Position Culling (Pre/Post-Transform)

At input the the SF stage, any objects containing a floating-point NaN value for Position X, Y, Z, or RHW will be unconditionally discarded. Note that this occurs on an object (not primitive) basis.

If Viewport Transformation is enabled, any objects containing a floating-point NaN value for post-transform Position X, Y or Z will be unconditionally discarded.

7.3.2 Viewport Transformation

If the **Viewport Transform Enable** bit of SF_STATE is ENABLED, a viewport transformation is applied to each vertex of the object.

The VPIndex associated with the leading vertex of the object is used to obtain the **Viewport Matrix Element** data from the corresponding element of the SF_VIEWPORT structure in memory. For each object vertex, the following scale and translate transformation is applied to the position coordinates:

x' = **m00** * x + **m30**

y' = **m11** * y + **m31**

z' = **m22** * z + **m32**

Software is responsible for computing the matrix elements from the viewport information provided to it from the API.

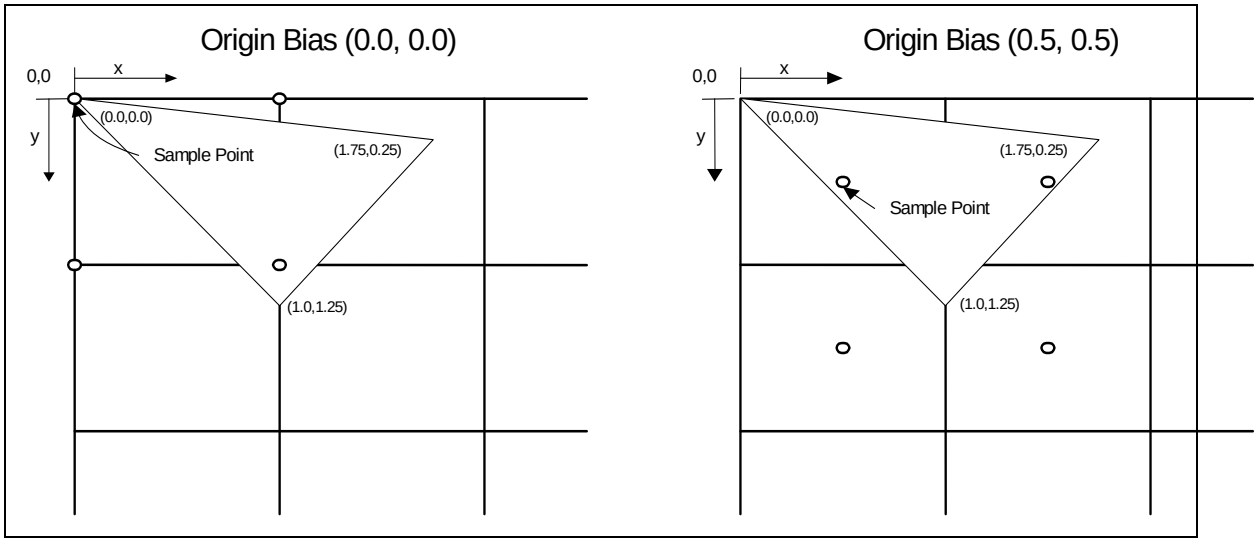
7.3.3 Destination Origin Bias

The positioning of the pixel sampling grid is programmable and is controlled by the **Destination Origin Horizontal/Vertical Bias** state variables (set via SF_STATE). If these bias values are both 0, pixels are sampled on an integer grid. Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0,0) falls within the primitive. This positioning of the sample grid corresponds with the rasterization rules, where "pixel centers are on an integer grid".

If the bias values are both 0.5, pixels are sampled on a “half” integer grid (i.e., X.5, Y.5). Pixel (0,0) will be considered inside the object if the sample point at XY coordinate (0.5,0.5) falls within the primitive. This positioning of the sample grid corresponds with the OpenGL rasterization rules, where “fragment centers” lay on a half-integer grid. It also corresponds with the Intel740 rasterizer (though that device did not employ “top left” rules).

Note that subsequent descriptions of rasterization rules for the various objects will be with reference to the pixel sampling grid.

Figure 7-8. Destination Origin Bias

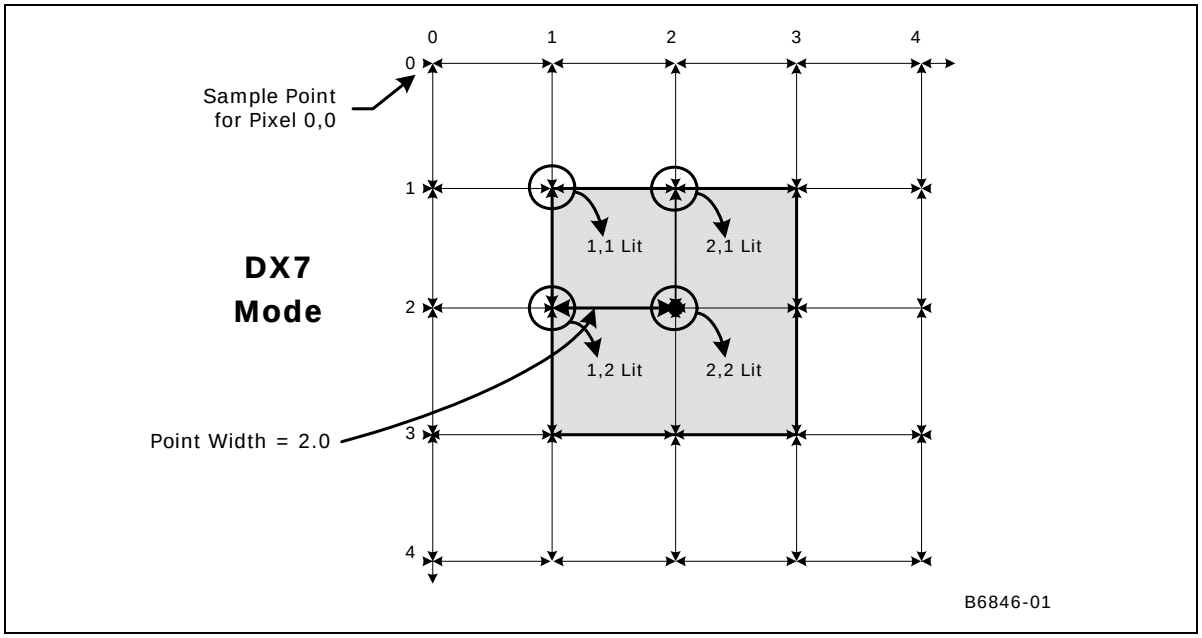


7.3.4 Point Rasterization Rule Adjustment

POINT objects are rasterized as square RECTANGLEs, with one exception: The **Point Rasterization Rule** state variable (in SF_STATE) controls the rendering of point object edges that fall directly on pixel sample points, as the treatment of these edge pixels varies between APIs.

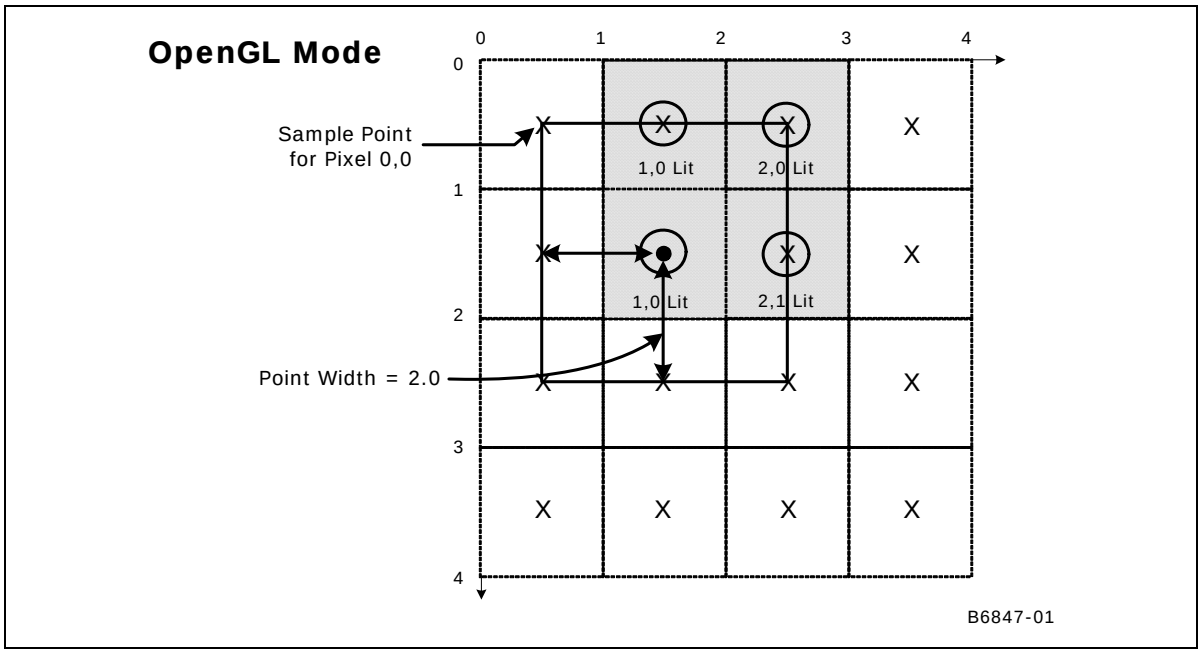
The following diagram shows the rasterization of a 2-pixel wide point centered at (2,2) given current rasterization rules (where the rasterization of points was changed to match the rasterization of an identical (square) polygon). Here the pixel sample grid coincides with the integer pixel coordinates, and the **Point Rasterization Rule** is set to RASTRULE_UPPER_LEFT. Note that the pixels which lie only on the upper and/or left edges are lit.

Figure 7-9. RASTRULE_UPPER_LEFT



The following diagram shows the rasterization of a 2-pixel wide point centered at (2,2) given "OpenGL" rasterization rules. Here the pixel sample grid coincides with half-integer pixel coordinates, and the **Point Rasterization Rule** is set to RASTRULE_UPPER_RIGHT. Note that the pixels which lie only on the upper and/or right edges are lit.

Figure 7-10. RASTRULE_UPPER_RIGHT



7.3.5 Drawing Rectangle Offset Application

The Drawing Rectangle Offset subfunction offsets the object's vertex X,Y positions by the pixel-exact, unclipped drawing rectangle origin (as programmed via the **Drawing Rectangle Origin X,Y** values in the 3DSTATE_DRAWING_RECTANGLE command). The Drawing Rectangle Offset subfunction (at least with respect to Color Buffer access) is unconditional, and therefore to (effectively) turn off the offset function the origin would need to be set to (0,0). A non-zero offset is typically specified when window-relative or viewport-relative screen coordinates are input to the device. Here the drawing rectangle origin would be loaded with the absolute screen coordinates of the window's or viewport's upper-left corner.

Clipping of objects which extend outside of the Drawing Rectangle occurs later in the pipeline. Note that this clipping is based on the "clipped" draw rectangle (as programmed via the **Clipped Drawing Rectangle** values in the 3DSTATE_DRAWING_RECTANGLE command), which must be clamped by software to the rendertarget boundaries. The unclipped drawing rectangle origin, however, can extend outside the screen limits in order to support windows whose origins are moved off-screen. This is illustrated in the following diagrams.

Figure 7-11. Onscreen Draw Rectangle

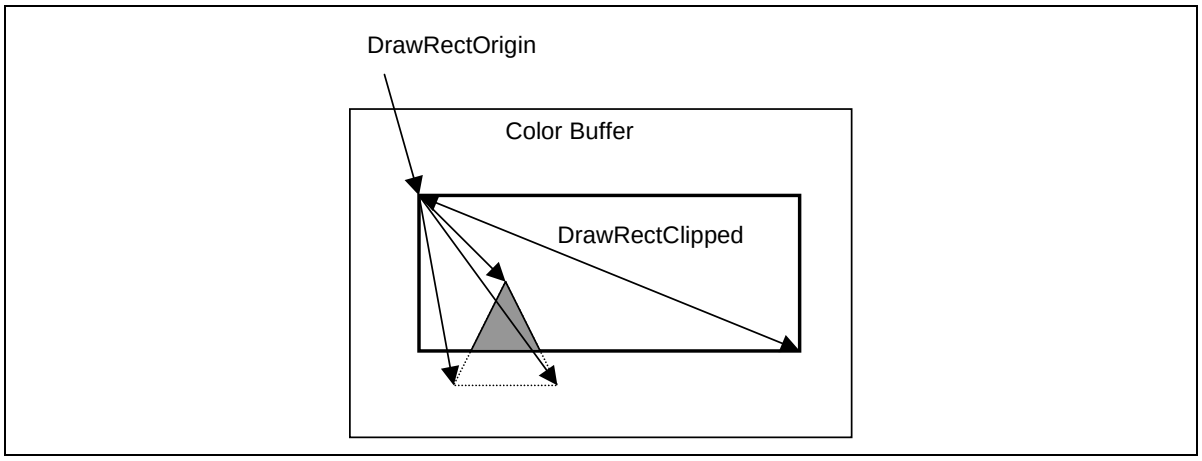
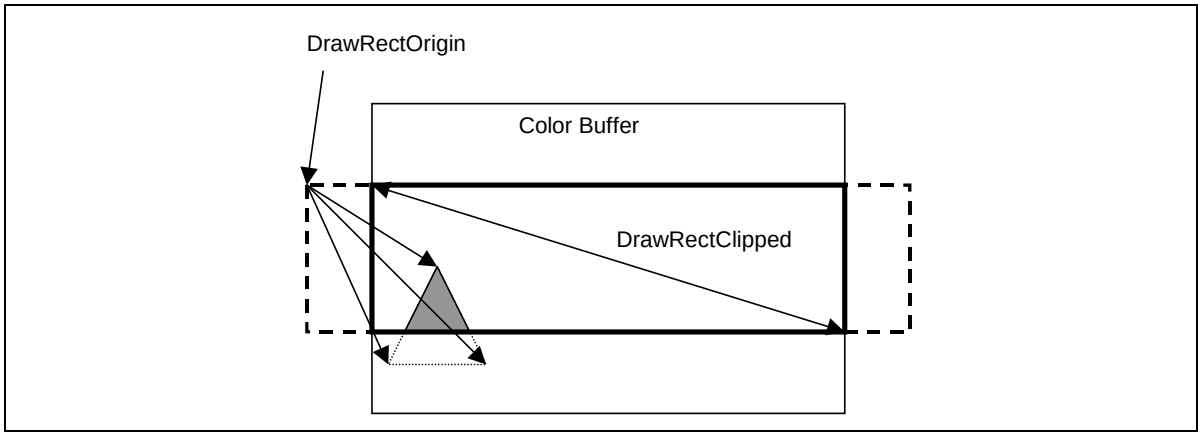


Figure 7-12. Partially-offscreen Draw Rectangle



7.3.5.1 3DST ATE_DRAWING_RECTANGLE

3DSTATE_DRAWING_RECTANGLE			
Project:	All	**Length Bias:**	2
The 3DSTATE_DRAWING_RECTANGLE command is used to set the 3D drawing rectangle and related state.			

DWord Bit		Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 00h 3DSTATE_DRAWING_RECTANGLE Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:16	**Clipped Drawing Rectangle Y Min** Project: All Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc Range [0,8191] (Device ignores bits 31:29) Specifies Ymin value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with Y coordinates *less than* Ymin will be clipped out. **Programming Notes** **[Pre-DevILK]:** This value must be *less than or equal to* **Clipped Drawing Rectangle Y Max.** If Ymin==Ymax, the clipped drawing rectangle is 1 pixel wide in the Y direction. **[DevILK+]:** This value can be *larger than* **Clipped Drawing Rectangle Y Max.** If Ymin>Ymax, the clipped drawing rectangle is null, all polygons are discarded. If Ymin==Ymax, the clipped drawing rectangle is 1 pixel wide in the Y direction.

<table>
<tr><th colspan="3">3DSTATE_DRAWING_RECTANGLE</th></tr>
<tr><td></td><td>15:0</td><td>Clipped Drawing Rectangle X Min
Project: All
Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc
Range [0,8191] (Device ignores bits 15:13)
Specifies Xmin value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with X coordinates less than Xmin will be clipped out.

<table>
<tr><th>Programming Notes</th><th>Project</th></tr>
<tr><td>This value must be less than or equal to Clipped Drawing Rectangle X Max. If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction.</td><td>Pre-DevILK</td></tr>
<tr><td>This value can be larger than Clipped Drawing Rectangle X Max. If Xmin>Xmax, the clipped drawing rectangle is null, all polygons are discarded. If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction.</td><td>DevILK+</td></tr>
</table></td></tr>
<tr><td>2</td><td>31:16</td><td>Clipped Drawing Rectangle Y Max
Project: All
Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc
Range [0,8191] (Device ignores bits 31:29)
Specifies Ymax value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with coordinates greater than Ymax will be clipped out.

<table>
<tr><th>Programming Notes</th></tr>
<tr><td>[Pre-DevILK]: This value must be greater than or equal to Clipped Drawing Rectangle Y Min. If Ymin==Ymax, the clipped drawing rectangle is 1 pixel wide in the Y direction.</td></tr>
<tr><td>[DevILK+]: This value can be less than Clipped Drawing Rectangle Y Min. If Ymax<Ymin, the clipped drawing rectangle is null, all polygons are discarded. If Ymin==Ymax, the clipped drawing rectangle is 1 pixel wide in the Y direction.</td></tr>
</table></td></tr>
</table>

3DSTATE_DRAWING_RECTANGLE

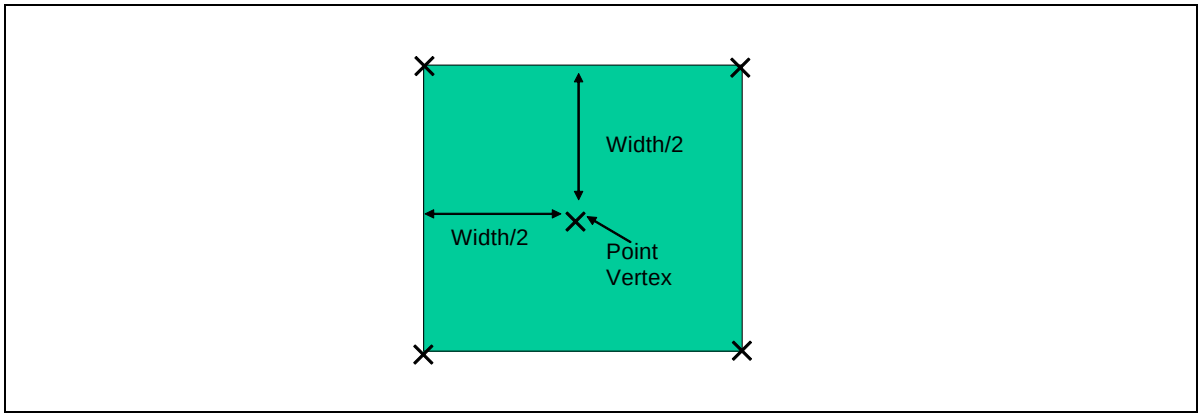
	15:0	**Clipped Drawing Rectangle X Max** Project: All Format: U16 in Pixels from Color Buffer origin (upper left corner) FormatDesc Range [0,8191] (Device ignores bits 15:13) Specifies Xmax value of (inclusive) intersection of Drawing rectangle with the Color (Destination) Buffer, used for clipping. Pixels with coordinates *greater than* Xmax will be clipped out. **Programming Notes** | **Project** This value must be *greater than or equal to* **Clipped Drawing Rectangle X Min.** If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction. | Pre-DevILK This value can be less *than* **Clipped Drawing Rectangle X Min.** If Xmax<Xmin, the clipped drawing rectangle is null, all polygons are discarded.If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction. | DevILK +
3	31:16	**Drawing Rectangle Origin Y** Project: All Format: S15 in Pixels from Color Buffer origin (upper left corner). FormatDesc Range [-8192,8191] (Bits 31:30 should be a sign extension) Specifies Y origin of Drawing Rectangle (in whole pixels) relative to origin of the Color Buffer, used to map incoming (Draw Rectangle-relative) vertex positions to the Color Buffer space.
	15:0	**Drawing Rectangle Origin X** Project: All Format: S15 in Pixels from Color Buffer origin (upper left corner). FormatDesc Range [-8192,8191] (Bits 15:14 should be a sign extension) Specifies X origin of Drawing Rectangle (in whole pixels) relative to origin of the Color Buffer, used to map incoming (Draw Rectangle-relative) vertex positions to the Color Buffer space.

7.3.6 Point Width Application

This stage of the pipeline applies only to 3DOBJ_POINT objects. Here the point object is converted from a single vertex to four vertices located at the corners of a square centered at the point's X,Y position. The width and height of the square are specified by a *point width* parameter. The **Use Point Width State** value in SF_STATE determines the source of the point width parameter: the point width is either taken from the **Point Width** value programmed in SF_STATE or the PointWidth specified with the vertex (as read back from the vertex VUE earlier in the pipeline).

The corner vertices are computed by adding and subtracting one half of the point width, as shown in Figure 7-13.

Figure 7-13. Point Width Application

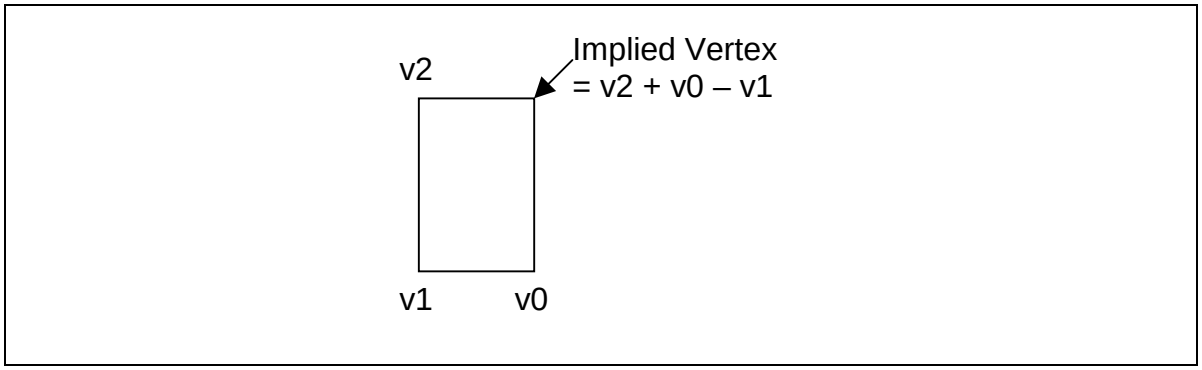


Z and W vertex attributes are copied from the single point center vertex to each of the four corner vertices.

7.3.7 Rectangle Completion

This stage of the pipeline applies only to 3DOBJ_RECTANGLE objects. Here the X,Y coordinates of the 4th (upper right) vertex of the rectangle object is computed from the first 3 vertices as shown in the following diagram. The other vertex attributes assigned to the implied vertex (v[3]) are UNDEFINED as they are not used. The Object Setup subfunction will use the values at only the first 3 vertices to compute attribute interpolants used across the entire rectangle.

Figure 7-14. Rectangle Completion



7.3.8 Vertex X,Y Clamping and Quantization

At this stage of the pipeline, vertex X and Y positions are in continuous screen (pixel) coordinates. These positions are quantized to subpixel precision by rounding the incoming values to the nearest subpixel (using round-to-nearest-or-even rules – this matched the reference device). The device supports rasterization with either 4 or 8 fractional (subpixel) position bits, as specified by the **Vertex SubPixel Precision Select** bit of SF_STATE.

The vertex X and Y screenspace coordinates are also **clamped** to the fixed-point "guardband" range supported by the rasterization hardware, as listed in the following table:

Table 7-6 Per-Device Guardband Extents

Device	Supported X,Y ScreenSpace "Guardband" Extent	Maximum Post-Clamp Delta (X or Y)
DevILK	[-8K,8K-1]	8K
DevILK	[-16K,16K-1]	16K

For [Pre-DevILK], an additional restriction effectively cuts the guardband extent in half: The screen-aligned 2D bounding-box of an object must not exceed 8K pixels in either X or Y. E.g., a line between (-6K,-6K) and (6K,6K) would not be rendered correctly, as its bounding box is 12K pixels in X and Y. This restriction effectively requires software to ensure all objects are contained within, or clipped to, a 2D region not exceeding 8K pixels in X or Y (even though that region can be located anywhere within the [-8K,8K-1] guardband extent). A similar restriction applies to [DevILK] though the guardband and maximum delta are doubled from [Pre-DevILK].

Note that this clamping occurs after the Drawing Rectangle Origin has been applied and objects have been expanded (i.e., points have been expanded to squares, etc.). In almost all circumstances, if an object's vertices are actually modified by this clamping (i.e., had X or Y coordinates outside of the guardband extent the rendered object will not match the intended result. Therefore software should take steps to ensure that this does not happen – e.g., by clipping objects such that they do not exceed these limits after the Drawing Rectangle is applied.

In addition, in order to be correctly rendered, objects must have a screenspace bounding box not exceeding 8K in the X or Y direction. This additional restriction must also be comprehended by software, i.e., enforced by use of clipping.

7.3.9 Degenerate Object Culling

At this stage of the pipeline, "degenerate" objects are discarded. This operation is automatic and cannot be disabled. (The object rasterization rules would by definition cause these objects to be "invisible" – this culling operation is mentioned here to reinforce that the device implementation optimizes these degeneracies as early as possible).

Degenerate objects are defined in Table 7-7.

Table 7-7. Degenerate Objects

objType	Degenerate Object Definition
3DOBJ_POINT	Two or more corner vertices are coincident (i.e., the radius quantized to zero)
3DOBJ_LINE	The endpoints are coincident
3DOBJ_TRIANGLE	All three vertices are collinear or any two vertices are coincident
3DOBJ_RECTANGLE	Two or more corner vertices are coincident

7.3.10 Triangle Orientation (Face) Culling

At this stage of the pipeline, 3DOBJ_TRIANGLE objects can be optionally discarded based on the "face orientation" of the object. This culling operation does not apply to the other object types.

This operation is typically called "back face culling", though front facing objects (or all 3DOBJ_TRIANGLE objects) can be selected to be discarded as well. Face culling is typically used to eliminate triangles facing away from the viewer, thus reducing rendering time.

The "winding order" of a triangle is defined by the the triangle vertex's 2D (X,Y) screen space position when traversed from v0 to v1 to v2. That traversal will proceed in either a clockwise (CW) or counter-clockwise (CCW) direction, as shown in Figure 7-15. (A degenerate triangle is considered to have a CW winding order).

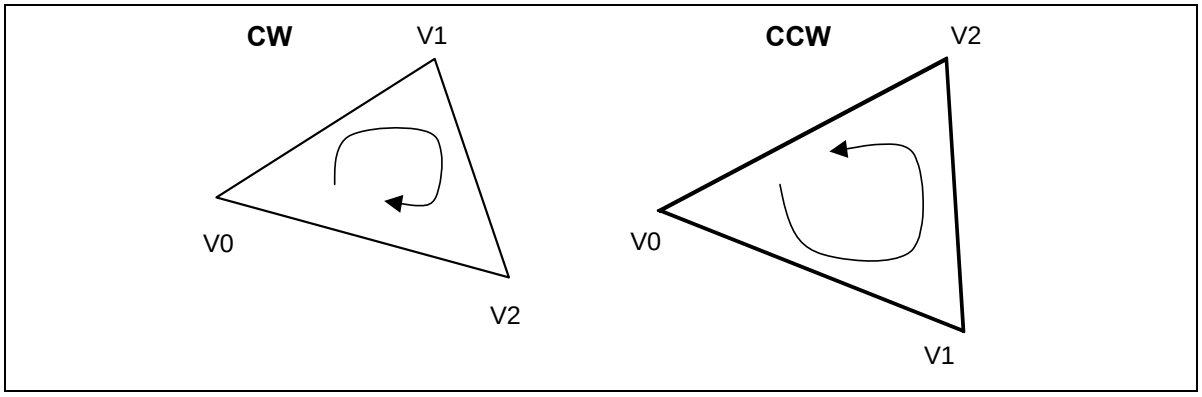


Figure 7-15. Triangle Winding Order

The **Front Winding** state variable in SF_STATE controls whether CW or CCW triangles are considered as having a "front-facing" orientation (at which point non-front-facing triangles are considered "back-facing"). The internal variable *invertOrientation* associated with the triangle object is then used to determine whether the orientation of a that triangle should be inverted. Recall that this variable is set in the Primitive Decomposition stage to account for the alternating orientations of triangles in strip primitives resulting form the ordering of the vertices used to process them.

The **Cull Mode** state variable in SF_STATE specifies how triangles are to be discarded according to their resultant orientation, as defined in Table 7-7.

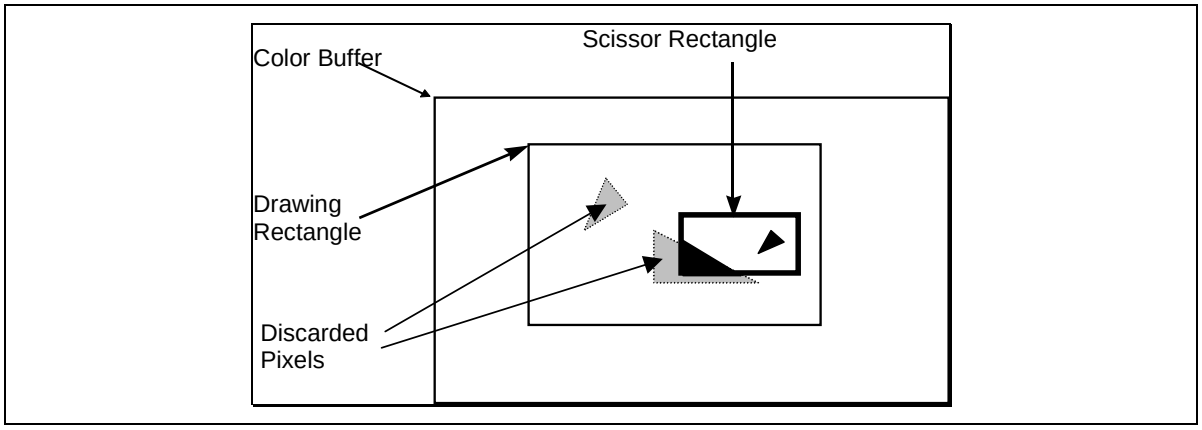
Table 7-8. Cull Mode

CullMode	Definition
CULLMODE_NONE	The face culling operation is disabled
CULLMODE_FRONT	Triangles with "front facing" orientation are discarded
CULLMODE_BACK	Triangles with "back facing" orientation are discarded
CULLMODE_BOTH	All triangles are discarded

7.3.11 Scissor Rectangle Clipping

A *scissor* operation can be used to restrict the extent of rendered pixels to a screen-space aligned rectangle. If the scissor operation is enabled, portions of objects falling outside of the intersection of the scissor rectangle and the clipped draw rectangle are clipped (pixels discarded).

The scissor operation is enabled by the **Scissor Rectangle Enable** state variable in SF_STATE. If enabled, the VPIndex associated with the leading vertex of the object is used to select the corresponding SF_VIEWPORT structure. Up to 16 structures are supported. The **Scissor Rectangle X,Y Min,Max** fields of the SF_VIEWPORT structure defines a scissor rectangle as a rectangle in integer pixel coordinates relative to the (unclipped) origin of the Drawing Rectangle. The scissor rectangle is defined relative to the Drawing Rectangle to better support the OpenGL API. (OpenGL specifies the "Scissor Box" in window-relative coordinates). This allows instruction buffers with embedded Scissor Rectangle definitions to remain valid even after the destination window (drawing rectangle) moves.



Specifying either scissor rectangle xmin > xmax or ymin > ymax will cause all polygons to be discarded for a given viewport (effectively a null scissor rectangle).

7.3.12 Line Rasterization

The device supports three styles of line rendering: *zero-width* (*cosmetic*) lines, *non-antialiased* lines, and *antialiased* lines. Non-antialiased lines are rendered as a polygon having a specified width as measured parallel to the major axis of the line. Antialiased lines are rendered as a rectangle having a specified width measured perpendicular to the line connecting the vertices.

The functions required to render lines is split between the SF and WM units. The SF unit is responsible for computing the overall geometry of the object to be rendered, including the pixel-exact bounding box, edge equations, etc., and therefore is provided with the screen-geometry-related state variables. The WM unit performs the actual scan conversion, determining the exact pixel included/excluded and coverage value for anti-aliased lines.

7.3.12.1 Zero-Width (Cosmetic) Line Rasterization

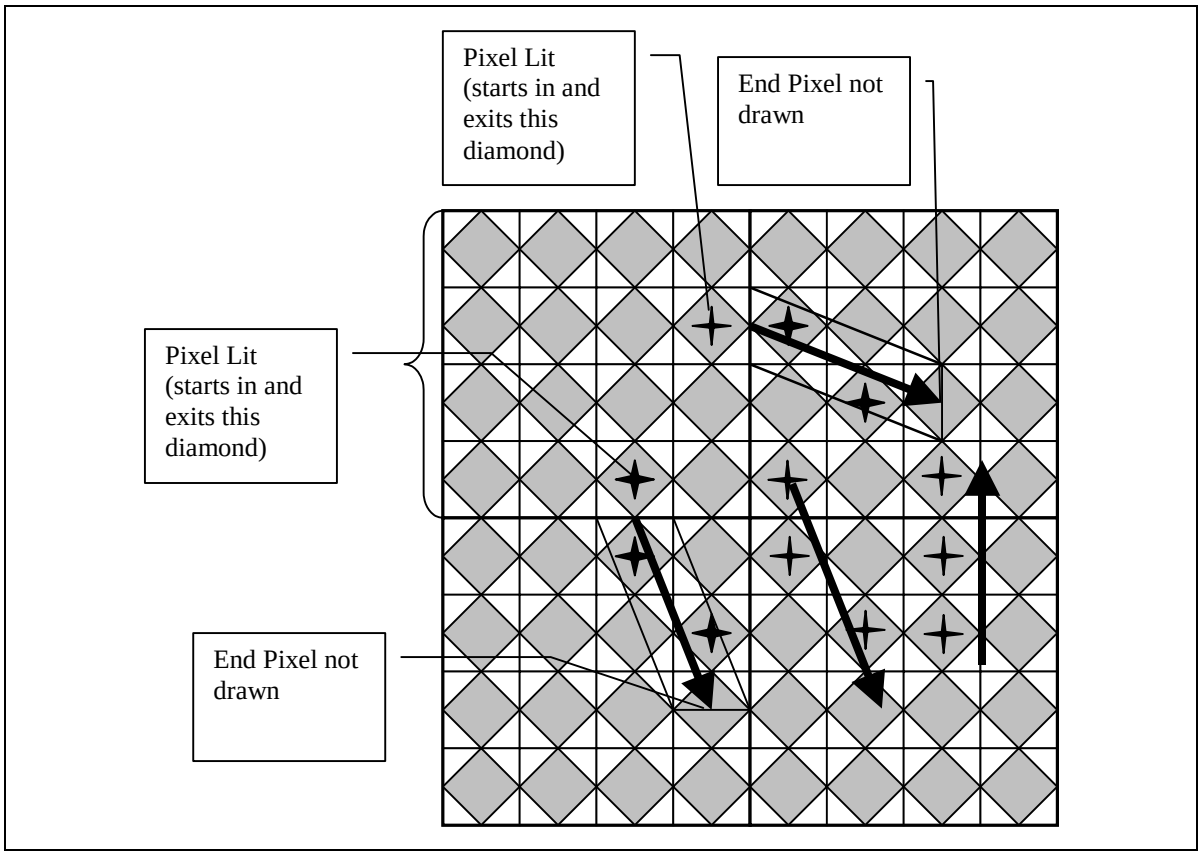
(The specification of zero-width line rasterization would be more correctly included in the WM Unit chapter, though is being included here to keep it with the rasterization details of the other line types).

When the **Line Width** is set to zero, the device will use special rules to rasterize zero-width ("cosmetic") lines. The **Anti-Aliasing Enable** state variable is ignored when **Line Width** is zero.

When the *LineWidth* is set to zero, the device will use special rules to rasterize "cosmetic" lines. The rasterization rules also comply with the OpenGL conformance requirements (for 1-pixel wide non-smooth lines). Refer to the appropriate API specifications for details on these requirements.

The GIQ rules basically intersect the directed, ideal line connecting two endpoints with an array of diamond-shaped areas surrounding pixel sample points. Wherever the line exits a diamond (including passing through a diamond), the corresponding pixel is lit. Special rules are used to define the subpixel locations which are considered interior to the diamonds, as a function of the slope of the line. When a line ends in a diamond (and therefore does not exit that diamond), the corresponding pixel is not drawn. When a line starts in a diamond and exits that diamond, the corresponding pixel is drawn.

The following diagram shows some examples of GIQ-rendered lines.

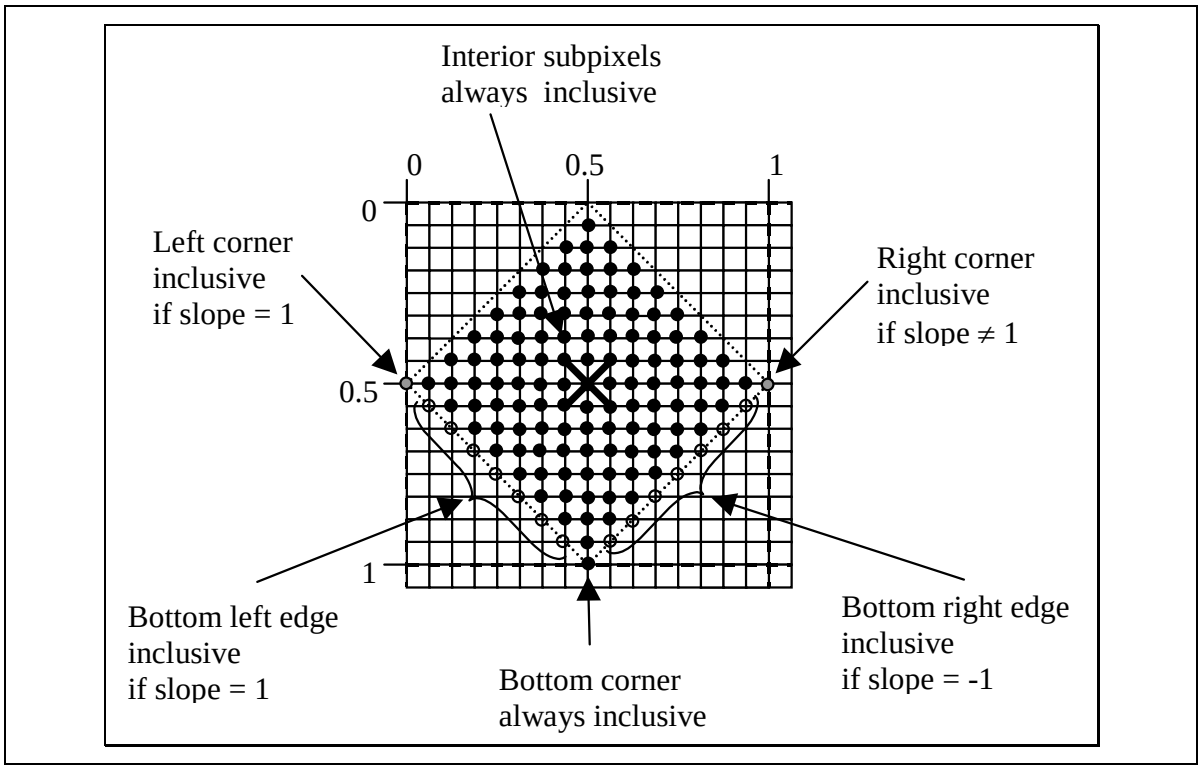


The following subsections describe the GIQ rules in more detail.

7.3.12.2 GIQ (Diamond) Sampling Rules – Legacy Mode

When the **Legacy Line Rasterization Enable** bit in WM_STATE is ENABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.



The solid-colored subpixels are considered “interior” to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.
2. **The right corner subpixel is inclusive as long as the line slope is not exactly one, in which case the left corner subpixel is inclusive.** Including the right corner subpixel ensures that lines with slopes in the range (1, +infinity] or [-infinity, -1) touch a diamond even when they cross exactly between pixel diamonds. Including the left corner on slope=1 lines is required for proper handling of slope=1 lines (see (3) below) – where if the right corner was inclusive, a slope=1 line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).
3. **The subpixels along the bottom left edge are inclusive only if the line slope = 1.** This is to correctly handle the case where a slope=1 line falls enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One slope=1 line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would case the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside" for slope=1 lines, we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.
4. **The subpixels along the bottom right edge are inclusive only if the line slope = -1.** Similar case as (3), except slope=-1 lines require the bottom right edge to be considered inclusive.

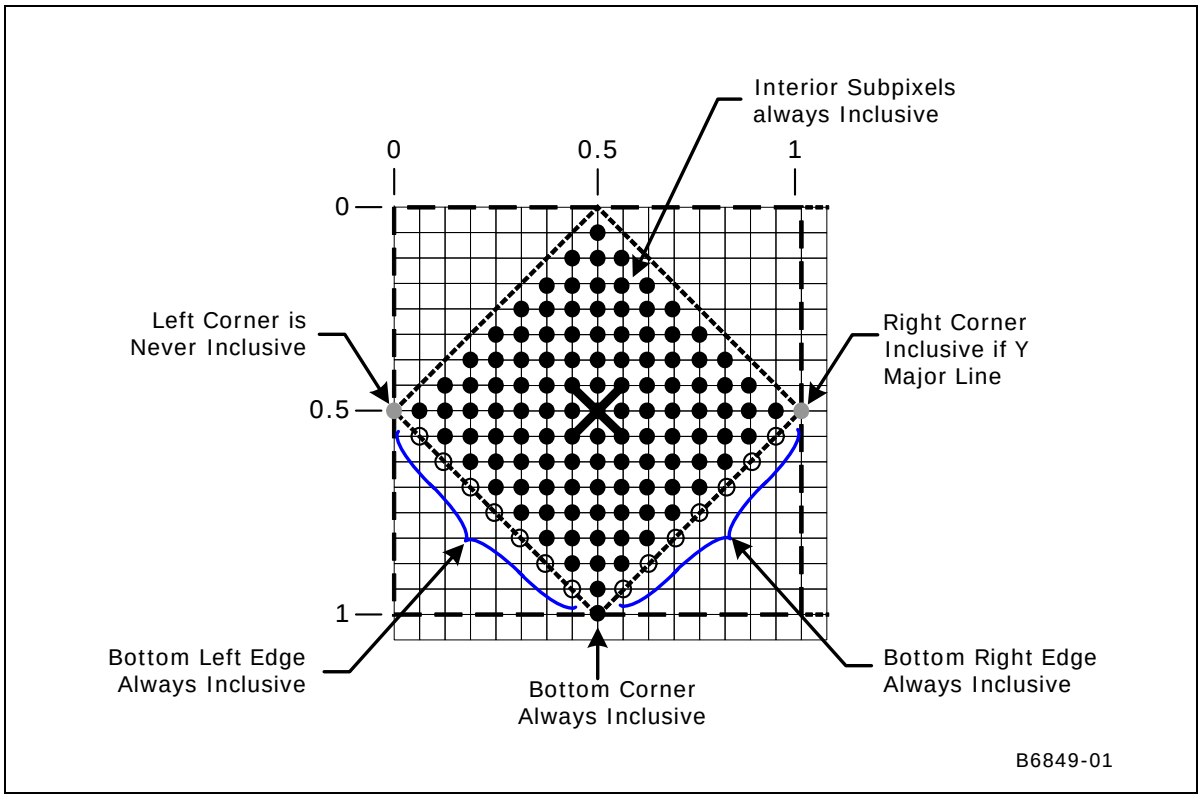
The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (slopePosOne, slopeNegOne).

```
delta_x             = point.x – sample.x
delta_y             = point.y – sample.y
distance            = abs(delta_x) + abs(delta_y)
interior            = (distance < 0.5)
bottom_corner       = (delta_x == 0.0)   && (delta_y == 0.5)
left_corner         = (delta_x == –0.5) && (delta_y == 0.0)
right_corner        = (delta_x == 0.5)   && (delta_y == 0.0)
bottom_left_edge    = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge   = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)
inside = interior ||
         bottom_corner ||
         (slopePosOne ? left_corner : right_corner) ||
         (slopePosOne && left_edge) ||
         (slopeNegOne && right_edge)
```

7.3.12.3 GIQ (Diamond) Sampling Rules

When the **Legacy Line Rasterization Enable** bit in WM_STATE is DISABLED, zero-width lines are rasterized according to the algorithm presented in this subsection. Also note that the **Last Pixel Enable** bit of SF_STATE controls whether the last pixel of the last line in a LINESTRIP_xxx primitive or the last pixel of each line in a LINELIST_xxx primitive is rendered.

Refer to the following figure, which shows the neighborhood of subpixels around a given pixel sample point. Note that the device divides a pixel into a 16x16 array of subpixels, referenced by their upper left corners.



The solid-colored subpixels are considered “interior” to the diamond centered on the pixel sample point. Here the Manhattan distance to the pixel sample point (center) is less than ½.

The subpixels falling on the edges of the diamond (Manhattan distance = ½) are exclusive, with the following exceptions:

1. **The bottom corner subpixel is always inclusive.** This is to ensure that lines with slopes in the open range (-1,1) touch a diamond even when they cross exactly between pixel diamonds.

2. **The right corner subpixel is inclusive as long as the line is not X Major (X Major is defined as -1 <= slope <= 1).** Including the right corner subpixel ensures that lines with slopes in the range (>1, +infinity] or [-infinity, <-1) touch a diamond even when they cross exactly between pixel diamonds.

3. The left corner subpixel is never inclusive. For Y Major lines, having the right corner subpixel as always inclusive requires that the left corner subpixel should never be inclusive, since a line falling exactly between pixel centers would wind up lighting pixel on both sides of the line (not desired).

4. The subpixels along the bottom left edge are always inclusive. This is to correctly handle the case where a line enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this "passing through" the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would case the pixel to be lit twice – breaking the rule that connected line segments should not cause double-hits or missing pixels. So, by considering the entire bottom left edge as "inside", we will only light the pixel when a line passes through the entire edge, or starts on the edge (or the left or bottom corner) and exits the diamond.

5. **The subpixels along the bottom right edge are always inclusive.** Same as case as (4), except slope=-1 lines require the bottom right edge to be considered inclusive.

The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (XMajor).

```
delta_x             = point.x – sample.x
delta_y             = point.y – sample.y
distance            = abs(delta_x) + abs(delta_y)
interior            = (distance < 0.5)
bottom_corner       = (delta_x == 0.0)   && (delta_y == 0.5)
left_corner         = (delta_x == –0.5) && (delta_y == 0.0)
right_corner        = (delta_x == 0.5)   && (delta_y == 0.0)
bottom_left_edge    = (distance == 0.5) && (delta_x < 0) && (delta_y > 0)
bottom_right_edge   = (distance == 0.5) && (delta_x > 0) && (delta_y > 0)
inside = interior ||
         bottom_corner ||
         (!XMajor && right_corner) ||
         ( bottom_left_edge) ||
         ( bottom_right_edge)
```

7.3.12.4 Non-Antialiased Wide Line Rasterization

Non-anti-aliased, non-zero-width lines are rendered as parallelograms that are centered on, and aligned to, the line joining the endpoint vertices. Pixels sampled interior to the parallelogram are rendered; pixels sampled exactly on the parallelogram edges are rendered according to the polygon "top left" rules.

The parallelogram is formed by first determining the major axis of the line (diagonal lines are considered x-major). The corners of the parallelogram are computed by translating the line endpoints by +/-(**Line Width** / 2) in the direction of the minor axis, as shown in the following diagram.

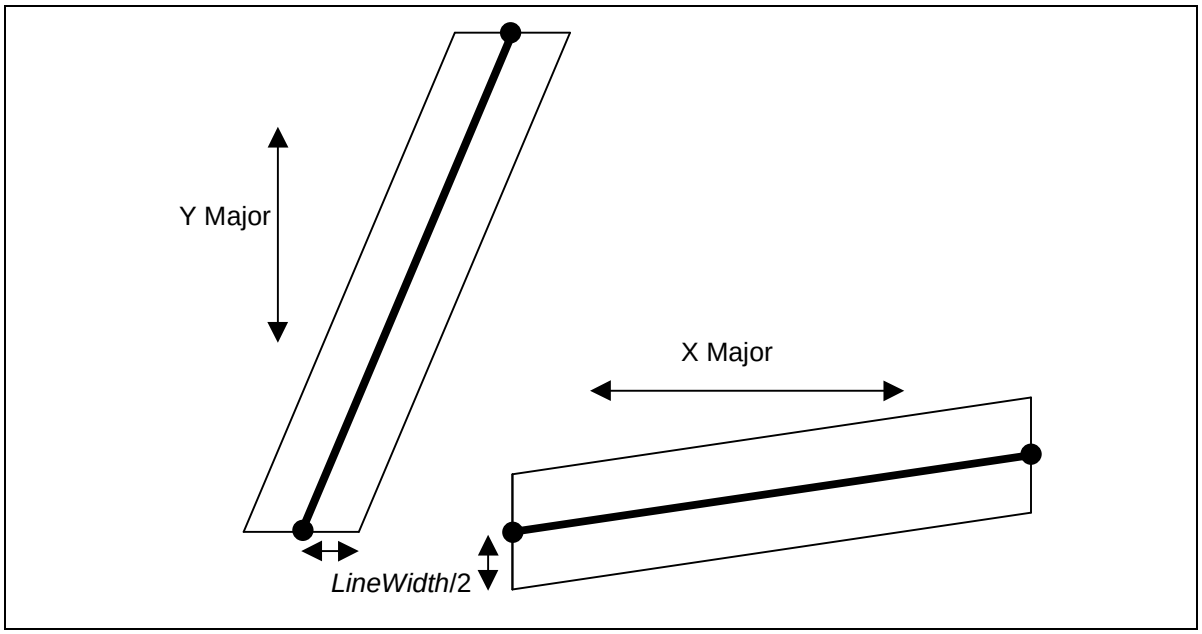


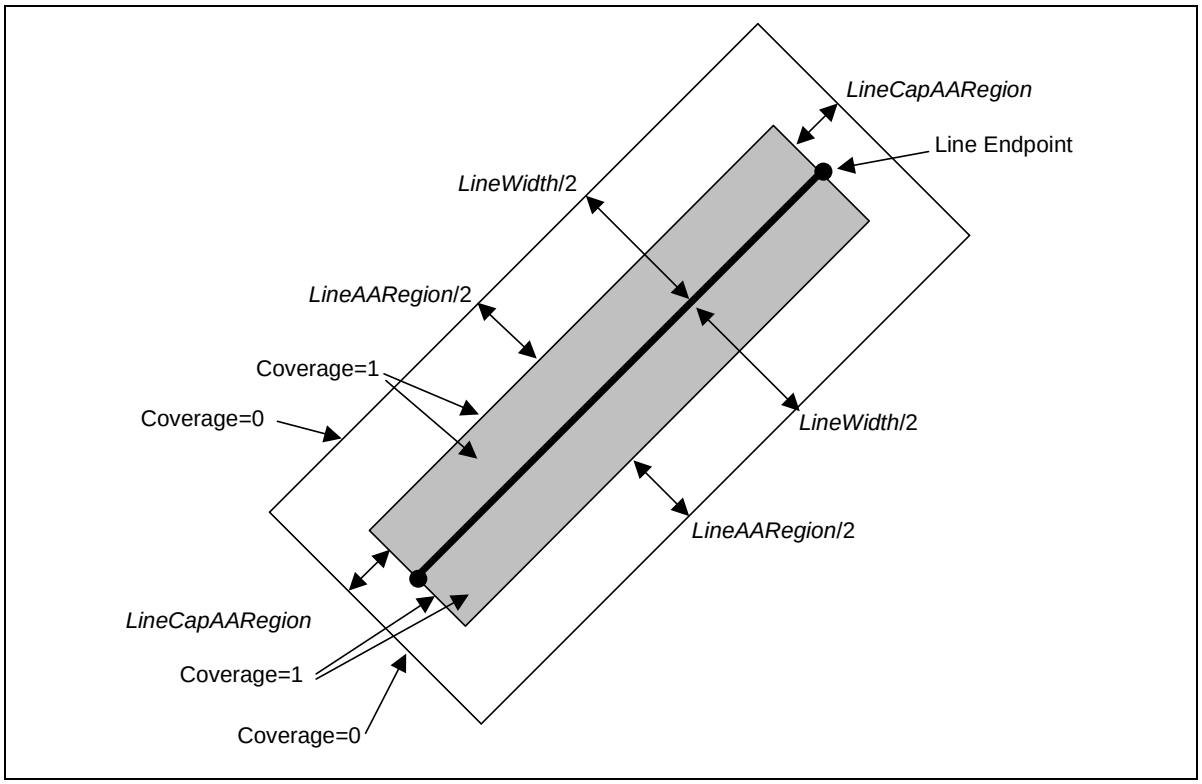
Figure 7-16. Non-Antialiased Line Rasterization

7.3.12.5 Anti-aliased Line Rasterization

Anti-aliased lines are rendered as rectangles that are centered on, and aligned to, the line joining the endpoint vertices. For each pixel in the rectangle, a fractional coverage value (referred to as Antialias Alpha) is computed – this coverage value will normally be used to attenuate the pixel's alpha in the pixel shader thread. The resultant alpha value is therefore available for use in those downstream pixel pipeline stages in order to generate the desired effect (e.g., use the attenuated alpha value to modulate the pixel's color, and add the result to the destination color, etc.). Note that software is required to explicitly program the pixel shader and pixel pipeline to obtain the desired anti-aliasing effect – the device will simply make the coverage-attenuated pixel alpha values available for use in the pixel shader.

The dimensions of the rendered rectangle, and the parameters controlling the coverage value computation, are programmed via the **Line Width**, **Line AA Region**, and **Line Cap AA Region** state variables, as shown below. The edges parallel to the line are located at the distance (*LineWidth*/2) from the line (measured in screen pixel units perpendicular to the line). The end-cap edges are perpendicular to the line and located at the distance (*LineCapAARegion*) from the endpoints.

Figure 7-17. Anti-aliased Line Rasterization



Along the parallel edges, the coverage values ramp from the value 0 at the very edges of the rectangle to the value 1 at the perpendicular distance (*LineAARegion*/2) from a given edge (in the direction of the line). A pixel's coverage value is computed with respect to the closest edge. In the cases where (*LineAARegion*/2) < (*LineWidth*/2), this results in a region of fractional coverage values near the edges of the rectangle, and a region of "fully-covered" coverage values (i.e., the value 1) at the interior of the line. When (*LineAARegion*/2) == (*LineWidth*/2), only pixel sample points falling exactly on the line can generate fully-covered coverage values. If (*LineAARegion*/2) > (*LineWidth*/2), no pixels can be fully-covered (it is expected that this case is not typically desired).

Along the end cap edges, the coverage values ramp from the value 1 at the line endpoint to the value 0 at the cap edge – itself at a perpendicular distance (*LineCapAARegion*) from the endpoint. Note that, unlike the line-parallel edges, there is only a single parameter (*LineCapAARegion*) controlling the extension of the line at the end caps and the associated coverage ramp.

The regions near the corners of the rectangle have coverage values influenced by distances from both the line-parallel and end cap edges – here the two coverage values are multiplied together to provide a composite coverage value.

The computed coverage value for each pixel is passed through the Windower Thread Dispatch payload. The Pixel Shader kernel should be passed (unmodified) by the shader to the Render Cache as part of its output message.

7.3.12.5.1 Anti-aliased Line Distance Mode

In [DevBW] and [DevCL], the distance from a pixel to the line is approximated by the "Manhattan Distance" (abs(delta_x)+abs)delta_y). In [DevCTG+] devices, a better approximation to the true perpendicular distance has been added for better visual quality and API compliance. On those devices, the **AA Line Distance Mode** bit in SF_STATE can be used to select between the legacy and improved distance calculations.

7.4 SF Pipeline State Summary

7.4.1 SF_STATE [Pre-DevGT]

SF_STATE [Pre-DevGT]		
Project: [Pre-De vGT]	**Length Bias:**	2

The SF_STATE structure defines the layout and function of the data referenced by the **Pointer to SF State** field of the PIPELINE_STATE_POINTERS command.

Note: The majority of the fields in DWords 0-4 have a common definition among the various 3D pipeline FF units. Refer to *3D Pipeline* for a general description of these fields.

DWord	Bit	Description
0	31:6	**Kernel Start Pointer** Project: All Format: **[Pre-DevILK]:** GeneralStateOffset[31:6] FormatDesc **[DevILK]:** InstructionBaseOffset[31:6] This field specifies the starting location (1st GENx core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the **General State Base Address [Pre-DevILK]** or **Instruction Base Address [DevILK]**. See *3D Pipeline* for more information.
	5:4	**Reserved** Project: All Format: MBZ

Errata	Description	Project
Errata BWT007	D: Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.)	**[DevBW]**

SF_STATE [Pre-DevGT]

	3:1	**GRF Register Count** Project: All Format: U3 register block count - 1 Range: [0,7] = [16,128] GRF registers Defines the number of GRF Register Blocks used by the kernel. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. See *3D Pipeline* for more information.
	0	**Reserved** Project: All Format: MBZ
1	30:26	**Reserved** Project: All Format: MBZ
	25:18	**Binding Table Entry Count** Format: U8 FormatDesc Range [0,255] Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** For kernels using a large number of binding table entries, it may be advantageous to set this field to zero to avoid prefetching too many entries and thrashing the state cache. See *3D Pipeline* for more information. [DevILK] MBZ
	17	**Thread Priority** Specifies the priority of the thread for dispatch

Value Name	Description	Project
0h Normal	Normal Priority	All
1h High	High Priority	All

Programming Notes
[Pre-DevILK]: this field must be zero.

	16	**Floating Point Mode** Specifies the floating point mode used by the dispatched thread **Value Name** / **Description** / **Project** 0h — IEEE-754 rules — Use IEEE-754 Rules — All 1h — Alternate rules — Use alternate rules — All
	15:14	**Reserved** Project: All Format: MBZ
	13	**Illegal Opcode Exception Enable** Project: All Format: Enable This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.
	12	**Reserved** Project: All Format: MBZ
	11	**MaskStack Exception Enable** Format: Enable FormatDesc This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*.
	10:8	**Reserved** Project: All Format: MBZ
	7	**Software Exception Enable** Format: Enable FormatDesc This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*.
	6:1	**Reserved** Project: All Format: MBZ
2	31:10	**Scratch Space Base Pointer** Format: GeneralStateOffset[31:10] FormatDesc Specifies the 1K-byte aligned offset of the scratch space area allocated to this FF unit. If required, each thread spawned by this FF unit will be allocated some portion of this space, as specified by **Per-Thread Scratch Space**. It is specified as a 1K-byte-granular offset from the **General State Pointer**.
	9:4	**Reserved** Project: All Format: MBZ

SF_STATE [Pre-DevGT]

	3:0	**Per Thread Scratch Space** Format: U32 FormatDesc Range [0,11] indicating [1k bytes, 2M bytes] in powers of two Specifies the amount of scratch space allowed to be used by each thread spawned by this FF unit. The driver must allocate enough contiguous scratch space, starting at the **Scratch Space Base Pointer**, to ensure that the **Maximum Number of Threads** each get **Per Thread Scratch Space** size without exceeding the driver-allocated scratch space. **Programming Notes** This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it.
3	30:25	**Constant URB Entry Read Length** Format: U6 FormatDesc Range: [0,63] Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments.
	24	**Reserved** Project: All Format: MBZ
	23:18	**Constant URB Entry Read Offset** Format: U6 FormatDesc Range: [0,63] Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload.
	17	**Reserved** Project: All Format: MBZ
	16:11	**Vertex URB Entry Read Length** Format: U32 FormatDesc Range [1,63] Specifies the amount of URB data read and passed in the thread payload for each Vertex URB entry, in 256-bit register increments. Programming Notes: It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read and passed to the thread.
	10	**Reserved** Project: All Format: MBZ

SF_STATE [Pre-DevGT]		
	9:4	**Vertex URB Entry Read Offset** Format: U6 FormatDesc Range: [0,63] Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB before being included in the thread payload. This offset applies to all Vertex URB entries passed to the thread.
	3:0	**Dispatch GRF Start Register for URB Data** Format: U4 FormatDesc Range: [0,15] Specifies the starting GRF register number for the URB portion (Constant + Vertices) of the thread payload.
4	30:25	**Maximum Number of Threads** Format: U6 representing thread count - 1 Range **[Pre-ILK]:** Range = [0,23] indicating thread count of [1,24] **[DevILK]:** Range = [0,47] indicating thread count of [1,48] Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock.
	24:19	**URB Entry Allocation Size** Format: U6 FormatDesc Range **[Pre-DevILK]:** Range = [0,31] indicating [1,32] 512-bit register increments **[DevILK]:** Range = [0,63] indicating [1,64] 512-bit register increments Specifies the length of each URB entry used by the unit, in 512-bit register increments – **[DevCTG+]:** If the Transposed URB Read feature is used, the URB entry size can be based on the non-transposed coefficient data storage layout, as this is how data is physically stored in the URB. Note that this layout uses ¾ of the storage compared to the transpose-on-write layout. **Programming Notes** Any change to this value requires a subsequent URB_FENCE command (see *Graphics Processing Engine*).

SF_STATE [Pre-DevGT]

	18:11	**Number of URB Entries** Format: U8 FormatDesc Range **[Pre-HVNABD]** Range = [1,64] **[HVNABD]** Range = [1,128] Specifies the number of URB entries that are used by the unit. **Programming Notes** Any change to this value requires a subsequent URB_FENCE command (see *Graphics Processing Engine*).
	10	**Statistics Enable** Format: Enabled FormatDesc If ENABLED, this FF unit will increment CL_PRIMITIVES_COUNT on behalf of the CLIP stage. If DISABLED, CL_PRIMITIVES_COUNT will be left unchanged. **Programming Notes** This bit should be set whenever clipping is enabled and the **Statistics Enable** bit is set in CLIP_STATE. It should be cleared if clipping is disabled *or* **Statistics Enable** in CLIP_STATE is clear.
	9:0	**Reserved** Project: All Format: MBZ
5	31:5	**Setup Viewport State Offset** Format: GeneralStateOffset[31:5] FormatDesc Specifies the 32-byte aligned offset of SF_VIEWPORT. This offset is relative to the **General State Base Address**. **Errata De / scription / Project** **Errata BWT007** — SF_VIEWPORT data pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) — **[DevBW-A]**
	4:2	**Reserved** Project: All Format: MBZ
	1	**Viewport Transform Enable** Project: All Format: Enable This bit controls the Viewport Transform function.

SF_STATE [Pre-DevGT]

0 — Front Winding

Determines whether a triangle object is considered "front facing" if the screen space vertex positions, when traversed in the order, result in a clockwise (CW) or counter-clockwise (CCW) winding order. Does not apply to points or lines.

Value Name	Description	Project
0h	FRONTWINDING_CW	All
1h	FRONTWINDING_CCW	All

6 — 31 — Anti-aliasing Enable

Format: Enable FormatDesc

: This field enables "alpha-based" line antialiasing.

Programming Notes
This field must be disabled if any of the render targets have integer (UINT or SINT) surface format.

30:29 — Cull Mode

Format: 3D_CullMode FormatDesc

: Controls removal (culling) of triangle objects based on orientation. The cull mode only applies to triangle objects and does not apply to lines, points or rectangles.

Value Name	Description	Project
0	CULLMODE_BOTH — All triangles are discarded (i.e., no triangle objects are drawn)	All
1	CULLMODE_NONE — No triangles are discarded due to orientation	All
2	CULLMODE_FRONT — Triangles with a front-facing orientation are discarded	
3	CULLMODE_BACK — Triangles with a back-facing orientation are discarded	

Programming Notes
Orientation determination is based on the setting of the **Front Winding** state.

28 — Reserved

27:24 — Line Width

Format: U3.1 (Units: pixels) FormatDesc

Range: [0.0, 7.5]

Controls width of line primitives.

Setting a Line Width of 0.0 specifies the rasterization of the "thinnest" (one-pixel-wide), non-antialiased lines. Note that this effectively overrides the effect of *AAEnable* (though the AAEnable state variable is not modified).

SF_STATE [Pre-DevGT]

	23:22	Line End Cap Antialiasing Region Width Format: 0: 0.5 pixels FormatDesc 1: 1.0 pixels 2: 2.0 pixels 3: 4.0 pixels This field specifies the distances over which the coverage of anti-aliased line end caps are computed. Note: this state is duplicated in the windower state descriptor
	21:20	**Point Rasterization Rule** Format: 3D_RasterizationRule FormatDesc **:** This field specifies the rasterization rules to be applied whenever the edges of a point primitive fall exactly on a pixel sampling point.

Value Name	Description	Project
0 RASTRULE_UPPER_LEFT	To match “normal” upper left rules for surface primitives	All
1 RASTRULE_UPPER_RIGHT	To match OpenGL point rasterization rules (round to + infinity, where this is the upper right direction wrt OpenGL screen origin of lower left).	All
2 Reserved	(RASTRULE_LOWER_LEFT not seen as useful)	
3 Reserved	(RASTRULE_LOWER_RIGHT not seen as useful))	

	18	**Reserved** Format: Disable FormatDesc Disables the culling of some zero-pixel triangles that otherwise might be discarded by the “Small Triangle Filter”. The 2x2 Triangle Filter attempts to remove triangles with bounding boxes of 2x2 pixels or less and that do not light any pixels.

Programming Notes
Disabling this filter should not affect the image rendered – only the performance of the device should be impacted.

SF_STATE [Pre-DevGT]

	17	**Scissor Rectangle Enable** Project: All Format: Enable **:** Enables operation of Scissor Rectangle.
	16:13	**Destination Origin (Pixel Sample Point) Horizontal Bias** Format: U0.4 FormatDesc Range 0.0 (ox0) or 0.5 (0x8) **:** This value is used to specify the horizontal subpixel position of the pixel sampling points (grid) used during rasterization. It is used in conjunction with the vertical bias (below) to position the pixel-sampling grid to provide the rasterization required by the API or the operation at hand. E.g., when rendering triangles, pixels will only be lit when their corresponding sample points fall within the triangle or exactly along certain edges of the triangle – and repositioning the sampling grid will yield somewhat different results. The unbiased sampling points (i.e., when this bias is (0.0,0.0)) are located at the upper-left corner of each screen space pixel. This places the sampling points at the intersections of the integer screen space coordinate grid – which is typically required to meet Windows GDI rules. Biasing by (0.5.,0.5) positions the pixel sampling points at the center of each screen space pixel (halfway between the integer coordinate grid) – which is typically required to meet OpenGL rasterization rules.
	12:9	**BitFieldName** Format: U0.4 FormatDesc Range: 0.0 (ox0) or 0.5 (0x8) **Destination Origin (Pixel Sample Point) Vertical Bias:** This value is used to specify the vertical position of the pixel sampling points (grid) used during rasterization. (See above description of the horizontal bias).
	8:0	**Reserved** Project: All Format: MBZ
7	31	**Last Pixel Enable** Project: All Format: Enable If ENABLED, the last pixel of a diamond line will be lit. This state will only affect the rasterization of Diamond lines (will not affect wide lines or anti-aliased lines). Programming Notes: Last pixel is applied to all lines of a LINELIST, and only the last line of a LINESTRIP.
	30:29	**Triangle Strip/List Provoking Vertex Select** Format: 0-based vertex index FormatDesc Selects which vertex of a triangle (in a triangle strip or list primitive) is considered the “provoking vertex”. Used for flat shading of primitives.

Value Name	Description	Project
0h	Vertex 0	All
1h	Vertex 1	All
2h	Vertex 2	All
3h	Vertex 3	All

SF_STATE [Pre-DevGT]

28:27 **Line Strip/List Provoking Vertex Select**

Format: 0-based vertex index FormatDesc

Selects which vertex of a line (in a line strip or list primitive) is considered the “provoking vertex”.

Value Name	Description	Project
0h Disable	Vertex 0	All
1h Enable	Vertex 1	All
2h	Reserved	
3h	Reserved	

26:25 **Triangle Fan Provoking Vertex Select**

Format: 0-based vertex index FormatDesc

Value Name	Description	Project
0h	Vertex 0	All
1h	Vertex 1	All
	Vertex 2	
	Reserved	

24:15 **Reserved** Project: All Format: MBZ

14 **AA Line Distance Mode**

Project: [DevCTG+]

Format: FormatDesc

This bit controls the distance computation for antialiased lines

[DevBW,DevCL]:

Reserved: MBZ

Value Name	Description	Project
1h E AALINEDISTANCE_TRUE nable	True distance computation. This is the normal setting which should yield WHQL compliance.	All

13 **Sprite Point Enable** Project: All Format: Enable

This bit is passed into the Setup thread payload for use by the Setup kernel as a hint to the setup kernel to overload texture coordinate setup to map some/all texture coordinates to full range (though there is no hardware requirement to do so). Software is free to use this bit for other purposes – it is simply inserted into SF thread payloads.

12 **Vertex Sub Pixel Precision Select**

Format: FormatDesc

Selects the number of fractional bits maintained in the vertex data

Value Name	Description	Project
0	8 sub pixel precision bits maintained	All
1	4 sub pixel precision bits maintained	All

11 **Use Point Width State**

Controls whether the point width passed on the vertex or from state is used for rendering point primitives

Value Name	Description	Project
0h	Use Point Width on Vertex	All
1h	Use Pointwidth from State	All

10:0 **Point Width**

Format: U8.3 FormatDesc

Range: [0.125, 255.875] pixels

This field specifies the size (width) of point primitives in pixels. This field is overridden (though not overwritten) whenever point width information is passed in the FVF.

7.4.1.1 [DevGT]

For **[DevGT]**, the state used by the SF stage is defined with this inline state packet.

3DSTATE_SF			
Project:	[DevGT]	**Length Bias:**	2

Bit		Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 0h 3DSTATE_PIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 13h 3DSTATE_SF Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 12h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:28	**Reserved** Project: All Format: MBZ

3DSTATE_SF		
	27:22	**Number of SF Output Attributes** Project: All Format: U6 Count of attributes Range [0,48] Specifies the number of vertex attributes passed from the SF stage to the WM stage (does not include Position). The actual number of attributes specified by this field must be set the same as the **Number of SF Output Attributes** field in 3DSTATE_WM. In the range description below, "swizzling" refers to the operations controlled by the following state fields: • **Attribute n Component Override X/Y/Z/W** • **Attribute n Constant Source** • **Attribute n Swizzle Select** • **Attribute n Source Attribute** • **Attribute n WrapShortest Enables** 0: Specifies no attributes (beyond position) are associated with vertices. 1-16: Specifies 1-16 attributes. Swizzling performed on Attributes 0-15 (as required). 17-32: Specifies 17-32 attributes. Swizzling performed on Attributes 0-15. Attributes 16-31 (as required) passed through unmodified. 33-48: Specifies 17-32 attributes (# attributes = field value – 16). Swizzling performed on Attributes 16-31 (as required) only. Attributes 0-15 passed through unmodified. Note: • Attribute n Component Override and Constant Source states apply to Attributes 16-31 (as required) instead of Attributes 0-15. E.g., this allows an Attribute 16-31 component to be overridden with the PrimitiveID value. • Attribute n WrapShortest Enables still apply to Attributes 0-15. • Attribute n Swizzle Select and Attribute n Source Attribute states are ignored and none of the swizzling functions available through these controls are performed.
	21	**Attribute Swizzle Enable** Project: All Format: Enable FormatDesc Enables the SF to perform swizzling on vertex attributes. See **Number of SF Output Attributes** field. If DISABLED, all vertex attributes are passed through.

3DSTATE_SF

	20	**Point Sprite Texture Coordinate Origin** Project: All Format: U1 enumerated type FormatDesc This state controls how Point Sprite Texture Coordinates are generated (when enabled on a per-attribute basis by **Point Sprite Texture Coordinate Enable**). **Value Name** / **Description** / **Project** 0h UPPERLEFT — Top Left = (0,0,0,1); Bottom Left = (0,1,0,1); Bottom Right = (1,1,0,1) — All 1h LOWERLEFT — Top Left = (0,1,0,1); Bottom Left = (0,0,0,1); Bottom Right = (1,0,0,1) — All
	19:16	**Reserved** Project: All Format: MBZ
	15:11	**Vertex URB Entry Read Length** Project: All Format: U5 FormatDesc Range [1,16] Specifies the amount of URB data read for each Vertex URB entry, in 256-bit register increments. **Programming Notes** / **Project** It is UNDEFINED to set this field to 0 indicating no Vertex URB data to be read. — All
	10	**Reserved** Project: All Format: MBZ
	9:4	**Vertex URB Entry Read Offset** Project: All Format: U6 FormatDesc Range [0,63] Specifies the offset (in 256-bit units) at which Vertex URB data is to be read from the URB.
	3:0	**Reserved** Project: All Format: MBZ
2	31:12	**Reserved** Project: All Format: MBZ
	11	**Legacy Global Depth Bias Enable** Project: All Format: Enable FormatDesc Enables the SF to use the Global Depth Offset Constant state unmodified. If this bit is not set, the SF will scale the Global Depth Offset Constant as described in section 8.4.2 of this document.

3DSTATE_SF		
	10	**Statistics Enable** Project: All Format: Enable FormatDesc If ENABLED, this FF unit will increment CL_PRIMITIVES_COUNT on behalf of the CLIP stage. If DISABLED, CL_PRIMITIVES_COUNT will be left unchanged. **Programming Notes** | **Project** This bit should be set whenever clipping is enabled and the **Statistics Enable** bit is set in CLIP_STATE. It should be cleared if clipping is disabled *or* **Statistics Enable** in CLIP_STATE is clear. | All
	9	**Global Depth Offset Enable Solid** Project: All Format: Enable FormatDesc Enables computation and application of Global Depth Offset for SOLID objects.
	8	**Global Depth Offset Enable Wireframe** Project: All Format: Enable FormatDesc Enables computation and application of Global Depth Offset when triangles are rendered in WIREFRAME mode.
	7	**Global Depth Offset Enable Point** Project: All Format: Enable FormatDesc Enables computation and application of Global Depth Offset when triangles are rendered in POINT mode.

3DSTATE_SF

	6:5	**FrontFace Fill Mode** Project: All Format: U2 enumerated type FormatDesc This state controls how front-facing triangle and rectangle objects are rendered.

Value Name		Description	Project
0h	SOLID	Any triangle or rectangle object found to be front-facing is rendered as a solid object. This setting is required when rendering rectangle (RECTLIST) objects.	All
1h	WIREFRAME	Any triangle object found to be front-facing is rendered as a series of lines along the triangle boundaries (as determined by the topology type and controlled by the vertex EdgeFlags).	All
2h	POINT	Any triangle object found to be front-facing is rendered as a set of point primitives at the triangle vertices (as determined by the topology type and controlled by the vertex EdgeFlags).	All
3h		Reserved	All

	4:3	**BackFace Fill Mode** Project: All Format: U2 enumerated type FormatDesc This state controls how back-facing triangle and rectangle objects are rendered.

Value Name		Description	Project
0h	SOLID	Any triangle or rectangle object found to be back-facing is rendered as a solid object. This setting is required when rendering rectangle (RECTLIST) objects.	All
1h	WIREFRAME	Any triangle object found to be back-facing is rendered as a series of lines along the triangle boundaries (as determined by the topology type and controlled by the vertex EdgeFlags).	All
2h	POINT	Any triangle object found to be back-facing is rendered as a set of point primitives at the triangle vertices (as determined by the topology type and controlled by the vertex EdgeFlags).	All
3h		Reserved	All

	2	**Reserved** Project: All Format: MBZ

3DSTATE_SF

	1	**Viewport Transform Enable** Project: All Format: Enable FormatDesc This bit controls the Viewport Transform function.
	0	**Front Winding** Project: All Determines whether a triangle object is considered "front facing" if the screen space vertex positions, when traversed in the order, result in a clockwise (CW) or counter-clockwise (CCW) winding order. Does not apply to points or lines.

Value Name	Description	Project
0h	FRONTWINDING_CW	All
1h	FRONTWINDING_CCW	All

3	31	**Anti-aliasing Enable** Project: All Format: Enable FormatDesc This field enables "alpha-based" line antialiasing.

Programming Notes
This field must be disabled if any of the render targets have integer (UINT or SINT) surface format.
This field is ignored when **Multisample Rasterization Mode** is MSRASTMODE_ON_xx.

	30:29	**Cull Mode** Project: All Format: 3D_CullMode FormatDesc Controls removal (culling) of triangle objects based on orientation. The cull mode only applies to triangle objects and does not apply to lines, points or rectangles.

Value Name	Description	Project
0h CULLMODE_BOTH	All triangles are discarded (i.e., no triangle objects are drawn)	All
1h CULLMODE_NONE	No triangles are discarded due to orientation	All
2h CULLMODE_FRONT	Triangles with a front-facing orientation are discarded	All
3h CULLMODE_BACK	Triangles with a back-facing orientation are discarded	All

Programming Notes	Project
Orientation determination is based on the setting of the **Front Winding** state.	All

	28	**Reserved**

3DSTATE_SF

27:18 Line Width

Project: All

Format: U3.7 FormatDesc

Range [0.0, 7.9921875]

Controls width of line primitives.

Setting a Line Width of 0.0 specifies the rasterization of the “thinnest” (one-pixel-wide), non-antialiased lines. Note that this effectively overrides the effect of *AAEnable* (though the AAEnable state variable is not modified).

Programming Notes	Project
Software must not program a value of 0.0 when running in MSRASTMODE_ON_xxx modes – zero-width lines are not available when multisampling rasterization is enabled.	All

17:16 Line End Cap Antialiasing Region Width

Project: All

Format: U2 FormatDesc

This field specifies the distances over which the coverage of anti-aliased line end caps are computed.

Note: this state is duplicated in 3DSTATE_WM.

Value Name	Description	Project
0h	0.5 pixels	All
1h	1.0 pixels	All
2h	2.0 pixels	All
3h	4.0 pixels	All

15:14 Reserved Project: All Format: MBZ

13 Reserved

12 Reserved

11 Scissor Rectangle Enable

Project: All

Format: Enable FormatDesc

Enables operation of Scissor Rectangle.

10 Reserved Project: All Format: MBZ

9:8 Multisample Rasterization Mode

Project: All

Format: U2 enumerated type FormatDesc

This state is duplicated in 3DSTATE_WM and both must be set to the same value. See the field in 3DSTATE_WM for definition details.

3DSTATE_SF

	7:0	**Reserved** Project: All Format: MBZ
4	31	**Last Pixel Enable** Project: All Format: Enable FormatDesc If ENABLED, the last pixel of a diamond line will be lit. This state will only affect the rasterization of Diamond lines (will not affect wide lines or anti-aliased lines). **Programming Notes** Last pixel is applied to all lines of a LINELIST, and only the last line of a LINESTRIP.
	30:29	**Triangle Strip/List Provoking Vertex Select** Project: All Format: 0-based vertex index FormatDesc Selects which vertex of a triangle (in a triangle strip or list primitive) is considered the “provoking vertex”. Used for flat shading of primitives.

Value Name	Description	Project
0h	Vertex 0	All
1h	Vertex 1	All
2h	Vertex 2	All
3h	Reserved	All

28:27	**Line Strip/List Provoking Vertex Select** Project: All Format: 0-based vertex index FormatDesc Selects which vertex of a line (in a line strip or list primitive) is considered the “provoking vertex”.

Value Name	Description	Project
0h	Vertex 0	All
1h	Vertex 1	All
2h	Reserved	All
3h	Reserved	All

3DSTATE_SF

26:25 Triangle Fan Provoking Vertex Select

Project: All

Format: 0-based vertex index FormatDesc

Selects which vertex of a triangle (in a triangle fan primitive) is considered the “provoking vertex”.

Value Name	Description	Project
0h	Vertex 0	All
1h	Vertex 1	All
2h	Vertex 2	All
3h	Reserved	All

24:15 Reserved Project: All Format: MBZ

14 AA Line Distance Mode

Project: All

Format: U1 FormatDesc

This bit controls the distance computation for antialiased lines.

Value Name	Description	Project
1h AALINEDISTANCE_TRUE	True distance computation. This is the normal setting which should yield WHQL compliance.	All

13 Reserved Project: All Format: MBZ

12 Vertex Sub Pixel Precision Select

Project: All

Format: U1 FormatDesc

Selects the number of fractional bits maintained in the vertex data

Value Name	Description	Project
0h	8 sub pixel precision bits maintained	All
1h	4 sub pixel precision bits maintained	All

11 Use Point Width State

Project: All

Format: U1 FormatDesc

Controls whether the point width passed on the vertex or from state is used for rendering point primitives.

Value Name	Description	Project
0h	Use Point Width on Vertex	All
1h	Use Point Width from State	All

3DSTATE_SF		
	10:0	**Point Width** Project: All Format: U8.3 FormatDesc Range [0.125, 255.875] pixels This field specifies the size (width) of point primitives in pixels. This field is overridden (though not overwritten) whenever point width information is passed in the FVF.
5	31:0	**Global Depth Offset Constant** Project: All Format: IEEE_FP FormatDesc Specifies the constant term in the Global Depth Offset function.
6	31:0	**Global Depth Offset Scale** Project: All Format: IEEE_FP FormatDesc Specifies the scale term used in the Global Depth Offset function.
7	31:0	**Global Depth Offset Clamp** Project: All Format: IEEE_FP FormatDesc Specifies the clamp term used in the Global Depth Offset function.
8	31	**Attribute 1 Component Override W** Project: All Format: Enable FormatDesc If set, the W component of output Attribute 1 or 17 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the W component of the constant vector specified by ConstantSource[1].
	30	**Attribute 1 Component Override Z** Project: All Format: Enable FormatDesc If set, the Z component of output Attribute 1 or 17 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the Z component of the constant vector specified by ConstantSource[1].
	29	**Attribute 1 Component Override Y** Project: All Format: Enable FormatDesc If set, the Y component of output Attribute 1 or 17 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the Y component of the constant vector specified by ConstantSource[1].
	28	**Attribute 1 Component Override X** Project: All Format: Enable FormatDesc If set, the X component of output Attribute 1 or 17 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the X component of the constant vector specified by ConstantSource[1].

3DSTATE_SF

	27	**Reserved** Project: All Format: MBZ
	26:25	**Attribute 1 Constant Source** Project: All Format: U2 enumerated type FormatDesc This state selects a constant vector which can be used to override individual components of Attribute 1 or 17 (refer to **Number of SF Output Attributes** field for information on which attribute is affected)

Value Na	me	Description	Project
0h	CONST_0000	Constant.xyzw = 0.0,0.0,0.0,0.0	All
1h	CONST_0001_FLOAT	Constant.xyzw = 0.0,0.0,0.0,1.0	All
2h	CONST_1111_FLOAT	Constant.xyzw = 1.0,1.0,1.0,1.0	All
3h	PRIM_ID	Constant.xyzw = PrimID (replicated)	All

	24	**Reserved** Project: All Format: MBZ
	23:22	**Attribute 1 Swizzle Select** Project: All Format: U2 enumerated type FormatDesc This state, along with Attribute 1 Source Attribute, specifies the source for output Attribute 1 or 17 (refer to **Number of SF Output Attributes** field for information on which attribute is affected).

Value Na	me	Description	Project
0h	INPUTATTR	This attribute is sourced from AttrInputReg[SourceAttribute]	All
1h	INPUTATTR_FACING	If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. If the object is back-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1].	All
2h	INPUTATTR_W	This attribute is sourced from AttrInputReg[SourceAttribute]. The W component is copied to the X component.	All
3h	INPUTATTR_FACING_W	If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. If the object is back-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1]. The W component is copied to the X component.	All

3DSTATE_SF		
	21	**Reserved** Project: All Format: MBZ
	20:16	**Attribute 1 Source Attribute** Project: All Format: U5 FormatDesc This field selects the source attribute for Attribute 1 or 17 (refer to **Number of SF Output Attributes** field for information on which attribute is affected). Source attribute 0 corresponds to the attribute immediately following the 4D homogeneous coordinate.
	15	**Attribute 0 Component Override W** Project: All Format: Enable FormatDesc If set, the W component of output Attribute 0 or 16 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the W component of the constant vector specified by ConstantSource[0].
	14	**Attribute 0 Component Override Z** Project: All Format: Enable FormatDesc If set, the Z component of output Attribute 0 or 16 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the Z component of the constant vector specified by ConstantSource[0].
	13	**Attribute 0 Component Override Y** Project: All Format: Enable FormatDesc If set, the Y component of output Attribute 0 or 16 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the Y component of the constant vector specified by ConstantSource[0].
	12	**Attribute 0 Component Override X** Project: All Format: Enable FormatDesc If set, the X component of output Attribute 0 or 16 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) is overridden by the X component of the constant vector specified by ConstantSource[0].
	11	**Reserved** Project: All Format: MBZ

10:9 **Attribute 0 Constant Source**

Project: All

Format: U2 enumerated type FormatDesc

This state selects a constant vector which can be used to override individual components of Attribute 0 or 16 (refer to **Number of SF Output Attributes** field for information on which attribute is affected)

Value Name	Description	Project
0h CONST_0000	Constant.xyzw = 0.0,0.0,0.0,0.0	All
1h CONST_0001_FLOAT	Constant.xyzw = 0.0,0.0,0.0,1.0	All
2h CONST_1111_FLOAT	Constant.xyzw = 1.0,1.0,1.0,1.0	All
3h PRIM_ID	Constant.xyzw = PrimID (replicated)	All

8 **Reserved** Project: All Format: MBZ

7:6 **Attribute 0 Swizzle Select**

Project: All

Format: U2 enumerated type FormatDesc

This state, along with Attribute 0 Source Attribute, specifies the source for output Attribute 0 or 16 (refer to **Number of SF Output Attributes** field for information on which attribute is affected).

Value Name	Description	Project
0h INPUTATTR	This attribute is sourced from AttrInputReg[SourceAttribute]	All
1h INPUTATTR_FACING	If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. If the object is back-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1].	All
2h INPUTATTR_W	This attribute is sourced from AttrInputReg[SourceAttribute]. The W component is copied to the X component.	All
3h INPUTATTR_FACING_W	If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. If the object is back-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1]. The W component is copied to the X component.	All

5 **Reserved** Project: All Format: MBZ

3DSTATE_SF		
	4:0	**Attribute 0 Source Attribute** Project: All Format: U5 FormatDesc This field selects the source attribute for Attribute 0 or 16 (refer to **Number of SF Output Attributes** field for information on which attribute is affected). Source attribute 0 corresponds to the attribute immediately following the 4D homogeneous coordinate.
9	31:0	**Attribute Control for Attributes 2,3** Project: All Format: see DW 8 FormatDesc
10	31:0	**Attribute Control for Attributes 4,5** Project: All Format: see DW 8 FormatDesc
11	31:0	**Attribute Control for Attributes 6,7** Project: All Format: see DW 8 FormatDesc
12	31:0	**Attribute Control for Attributes 8,9** Project: All Format: see DW 8 FormatDesc
13	31:0	**Attribute Control for Attributes 10,11** Project: All Format: see DW 8 FormatDesc
14	31:0	**Attribute Control for Attributes 12,13** Project: All Format: see DW 8 FormatDesc
15	31:0	**Attribute Control for Attributes 14,15** Project: All Format: see DW 8 FormatDesc
16	31:0	**Point Sprite Texture Coordinate Enable** Project: All Format: 32-bit bitmask FormatDesc When processing point primitives, the attributes from the incoming point vertex are typically copied to the point object corner vertices. However, if a bit is set in this field, the corresponding Attribute is selected as a Point Sprite Texture Coordinate, in which case each corner vertex is assigned a pre-defined texture coordinate as defined by the **Point Sprite Texture Coordinate Origin** state bit. Bit 0 corresponds to output Attribute 0.
17	31:0	**Constant Interpolation Enable[31:0]** Project: All This field is a bitmask containing a Constant Interpolation Enable bit for each corresponding attribute. If a bit is set, that attribute will undergo constant interpolation, and the corresponding **WrapShortest Enable** bits (if defined) will be ignored. If a bit is clear, components which are not enabled for WrapShortest interpolation (if defined) will be linearly interpolated.

3DSTATE_SF		
18	31:28	**Attribute 7 WrapShortest Enables** Project: All Format: 4-bit bitmask FormatDesc This state selects which components (if any) of Attribute 7 or 23 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) are to be interpolated in a “wrap shortest” fashion. Operation is UNDEFINED if any of these bits are set and the **Constant Interpolation Enable** bit associated with this attribute is set. Bit 0: WrapShortest X Component Bit 1: WrapShortest Y Component Bit 2: WrapShortest Z Component Bit 3: WrapShortest W Component
	27:24	**Attribute 6 WrapShortest Enables** Project: All (See above).
	23:20	**Attribute 5 WrapShortest Enables** Project: All (See above).
	19:16	**Attribute 4 WrapShortest Enables** Project: All (See above).
	15:12	**Attribute 3 WrapShortest Enables** Project: All (See above).
	11:8	**Attribute 2 WrapShortest Enables** Project: All (See above).
	7:4	**Attribute 1 WrapShortest Enables** Project: All (See above).
	3:0	**Attribute 0 WrapShortest Enables** Project: All (See above).

3DSTATE_SF		
19	31:28	**Attribute 15 WrapShortest Enables** Project: All Format: 4-bit bitmask FormatDesc This state selects which components (if any) of Attribute 15 or 31 (refer to **Number of SF Output Attributes** field for information on which attribute is affected) are to be interpolated in a “wrap shortest” fashion. Operation is UNDEFINED if any of these bits are set and the **Constant Interpolation Enable** bit associated with this attribute is set. Bit 0: WrapShortest X Component Bit 1: WrapShortest Y Component Bit 2: WrapShortest Z Component Bit 3: WrapShortest W Component
	27:24	**Attribute 14 WrapShortest Enables** Project: All (See above).
	23:20	**Attribute 13 WrapShortest Enables** Project: All (See above).
	19:16	**Attribute 12 WrapShortest Enables** Project: All (See above).
	15:12	**Attribute 11 WrapShortest Enables** Project: All (See above).
	11:8	**Attribute 10 WrapShortest Enables** Project: All (See above).
	7:4	**Attribute 9 WrapShortest Enables** Project: All (See above).
	3:0	**Attribute 8 WrapShortest Enables** Project: All (See above).

7.4.2 SF_VIEWPORT

The viewport-specific state used by the SF unit (SF_VIEWPORT) is stored as an array of up to 16 elements, each of which contains the DWords described below. The start of each element is spaced 8 DWords apart. The location of first element of the array, as specified by **Setup Viewport State Offset**, is aligned to a 32-byte boundary.

DWord	Bit	Description
0	31:0	**Viewport Matrix Element m00** Format = IEEE_Float
1	31:0	**Viewport Matrix Element m11** Format = IEEE_Float
2	31:0	**Viewport Matrix Element m22** Format = IEEE_Float
3	31:0	**Viewport Matrix Element m30** Format = IEEE_Float
4	31:0	**Viewport Matrix Element m31** Format = IEEE_Float
5	31:0	**Viewport Matrix Element m32** Format = IEEE_Float
6	31:16	**Scissor Rectangle Y Min ([Pre-DevSNB]) :** Specifies Y Min coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) Y coordinates *less than* **Y Min** will be clipped out if Scissor Rectangle is enabled. NOTE: If Y Min is set to a value greater than Y Max, all primitives will be discarded for this viewport. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]
	15:0	**Scissor Rectangle X Min ([Pre-DevSNB]):** Specifies X Min coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) X coordinates *less than* **X Min** will be clipped out if Scissor Rectangle is enabled. NOTE: If X Min is set to a value greater than X Max, all primitives will be discarded for this viewport. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]
7	31:16	**Scissor Rectangle Y Max ([Pre-DevSNB]):** Specifies Y Max coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) Y coordinates *greater than* **Y Max** will be clipped out if Scissor Rectangle is enabled. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]

DWord	Bit	Description
	15:0	**Scissor Rectangle X Max ([Pre-DevSNB]):** Specifies X Max coordinate of (inclusive) Scissor Rectangle used for scissor test. Pixels with (Draw Rectangle-relative) Y coordinates *greater than* **X Max** will be clipped out if Scissor Rectangle is enabled. Format = U16 in Pixels from Drawing Rectangle origin (upper left corner). Range = [0,8191]

7.5 The SF Thread -- Interpolation Coefficient Calculation [Pre-DevSNB]

The final step in object setup is to calculate the interpolation coefficients. This must be done separately (though hopefully in parallel) for each vertex attribute, and is performed by a thread running on an execution unit.

7.5.1 SF Setup Parameters Passed to SF Thread

This section describes some of the parameters computed by the SF unit and passed to SF threads.

7.5.1.1 TRIANGLE Parameters

The SF unit reorders triangle vertices prior to setup computation. The "start vertex" (V0) is defined as being the top-most (least positive Y position) vertex. If more than one vertex shares this Y position, the left-most (least positive Z position) vertex is selected. Once the start vertex is determined, V1 is the next vertex in the clockwise direction, and V2 is the remaining vertex. (Note that degenerate triangles will have been removed by this point, therefore there is no ambiguity in vertex reordering.)

Once the vertices are reordered into V0,V1,V2, the SF unit computes the **Y2-Y0**, **Y1-Y0**, **X2-X0**, **X1-X0**, and **Determinant** values (described in the thread payload below).

The SF unit will use the V0,V1,V2 ordering for the VUE data that follows the thread payload (and possibly the CURBE portion of the payload).

7.5.1.2 RECT ANGLE Parameters

With regard to SF thread payload, RECTANGLE objects are handled just like TRIANGLE objects. The 3 vertices supplied for the object are subject to reordering and used in SF unit setup computations. The same parameters are passed in the thread payload as for TRIANGLE objects, and the 3 (possibly reordered) VUEs are included in the payload.

7.5.1.3 POINT Parameters

Point width is applied to POINT objects, expanding them to screen-aligned squares. The SF unit selects the following vertices for the normal setup computations: Upper-left = V0, Lower-right = V1, Lower-left = V2.

In this respect they appear as RECTANGLES in the SF thread payload. However, only the single original object vertex (the center) is passed as VUE data.

The **Sprite Point Enable** bit from SF_STATE is passed in the SF thread header to assist in the support of API "sprite points," where some/all texture coordinates are set to full-range over the point square vs. all corners being assigned the constant value provided by the object (center) vertex.

7.5.1.4 LINE Parameters

The SF unit reorders line vertices prior to setup computation. The "start vertex" (V0) is defined as being the top-most (least positive Y position) vertex. If the other vertex shares this Y position, the left-most (least positive Z position) vertex is selected. Once the start vertex is determined, V1 is the remaining vertex. (Note that degenerate lines will have been removed by this point, therefore there is no ambiguity in vertex reordering.)

Once the vertices are reordered into V0,V1, the SF unit computes the **Y1-Y0**, **X1-X0**, and **Determinant** values (described in the thread payload below).

The SF unit will use the V0,V1 ordering for the VUE data that follows the thread payload (and possibly the CURBE portion of the payload).

7.5.2 SF (Setup) Thread Payload

DWord	Bit	Description
R0.7	31	Reserved
	30:0	Reserved
R0.6	31:24	Reserved
	23:0	**Thread ID:**This field uniquely identifies this thread within the threads spawned by this FF unit, over some period of time. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer:**Specifies the 1K-byte aligned offset (from the **General State Base Address**) to the scratch space allocated to this thread. Format = GeneralStateOffset[31:10]
	9:8	Reserved
	7:0	**FFTID:** This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: U8
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:4	Reserved

DWord	Bit	Description
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:0	Reserved
R0.1	31:0	Reserved
R0.0	31:16	**Handle ID:** This ID is assigned by the fixed function unit and links the thread to a specific entry within the fixed function unit.
	15:0	**URB Return Handle:** This is the URB handle where the thread's results are to be placed (aka the Primitive URB Entry, or PUE).
R1.7	31:0	Reserved
R1.6	31:0	Y2-Y0 (aka dY2) : For TRIANGLE, RECT and POINT objects: This field contains the value (Y2 – Y0) , where the indices are relative to the "start" vertex. This value is also known as "dY2" , where the "2" is the relative order of the delta term around a triangle, not a vertex index. For LINE objects: Reserved Format: FLOAT32
R1.5	31:0	**Y1-Y0** (aka dY0) **:** For all objects: This field contains the value (Y1 – Y0) , where the indices are relative to the "start" vertex. This value is also known as "dY0" , where the "0" is the relative order of the delta term around a triangle, not a vertex index. Format: FLOAT32
R1.4	31:0	X2-X0 (aka dX2) : For TRIANGLE, RECT and POINT objects: This field contains the value (X2 – X0) , where the indices are relative to the "start" vertex. This value is also known as "dX2" , where the "2" is the relative order of the delta term around a triangle, not a vertex index. For LINE objects: Reserved Format: FLOAT32
R1.3	31:0	X1-X0 (aka dX0) : For all objects: This field contains the value (X1 – X0) , where the indices are relative to the "start" vertex. This value is also known as "dX0" , where the "0" is the relative order of the delta term around a triangle, not a vertex index. Format: FLOAT32
R1.2	31:0	Determinant For TRIANGLE, RECT and POINT objects: (X1-X0)(Y2-Y0) – (X2-X0)(Y1-Y0) For LINE objects: (X1-X0)(X1-X0) + (Y1-Y0)(Y1-Y0) Format: FLOAT32

DWord Bit		Description
R1.1	31:0	**Provoking Vertex:** This field contains the relative index (0-2) of the reordered vertex considered the "provoking" vertex, given the PrimType and related SF_STATE state variables (**xxx Provoking Vertex Select**). The SF thread can use this value when performing setup computations for "constant-interpolated" vertex attributes. 0: V0 1: V1 2: V2
R1.0	31:18	Reserved
	17	**Front/Back Facing Polygon:** Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use. 0: Front Facing 1: Back Facing
	16	**Sprite Point Enable:**This a copy of the **Sprite Point Enable** bit in SF_STATE. It is passed in the payload strictly for use by the SF (Setup) thread – there is no other hardware function involved. For example (and the expected usage model), a setup kernel processing a point object could overload texture coordinate setup to map texture to full range, thus mapping a texture to the sprite point. Format: Enable
	15:0	**Primitive Type:** This is the unmodified PrimType of the primitive topology containing the object, as received from the 3D pipeline. E.g., a point object within a POINTLIST will have POINTLIST passed in this field even though the point is expanded to a square. Format: See 3DPRIMITIVE description in *Vertex Fetch* for encoding
R2.7	31:0	Reserved
R2.6	31:0	**[Dev ILK]: b0_vertex2:** sfunit sends in the b0 term into the setup kernel for vertex2 of the line or triangle.
R2.5	31:0	Inverse W2: For TRIANGLE, RECTANGLE and POINT objects: This is the position 1/W value associated with V2. The SF thread can use this value (passed directly from the SF unit) in order to avoid having to have the Vertex Header portions of the object vertex VUEs from being included in the VUE portion of the SF thread payload. For LINE objects: Reserved Format: FLOAT32
R2.4	31:0	Z2: For TRIANGLE, RECTANGLE and POINT objects: This is the position Z value associated with V2. The SF unit computes this value given the position Z value from the VUE Vertex Header and state information, etc. For LINE objects: Reserved Format: FLOAT32
R2.3	31:0	Inverse W1 : For all objects: This is the position 1/W value associated with V1. See **Inverse W2.** Format: FLOAT32

DWord	Bit	Description
R2.2	31:0	Z1 For all objects: This is the position Z value associated with V1. See **Z2.** Format: FLOAT32
R2.1	31:0	Inverse W0 For all objects: This is the position 1/W value associated with V0. See **Inverse W2.** Format: FLOAT32
R2.0	31:0	Z0 For all objects: This is the position Z value associated with V0. See **Z2.** Format: FLOAT32
R3.7	31:0	**[Dev ILK]: b0_vertex1:** sfunit sends in the b0 term into the setup kernel for vertex1 of the line or triangle.
R3.6	31:0	**[DevILK]: b0_vertex0:** sfunit sends in the b0 term into the setup kernel for vertex0 of the line or triangle.
R3.5	31:0	**[DevILK]: b2_vertex2:** sfunit sends in the b2 term for bary interpolation in the setup kernel for vertex2 of the triangle. Field ignored for a line.
R3.4	31:0	**[DevILK]: vertex2:** sfunit sends in the b1 term for bary interpolation in the setup kernel for vertex2 of the triangle. Field ignored for a line.
R3.3	31:0	**[DevILK]: b2_vertex1:** sfunit sends in the b2 term for bary interpolation in the setup kernel for vertex1 of the line or triangle.
R3.2	31:0	**[DevILK]: b1_vertex1:** sfunit sends in the b1 term for bary interpolation in the setup kernel for vertex1 of the line or triangle.
R3.1	31:0	**[DevILK]: b2_vertex0:** sfunit sends in the b2 term for bary interpolation in the setup kernel for vertex0 of the line or triangle.
R3.0	31:0	**[DevILK]: b1_vertex0:** sfunit sends in the b1 term for bary interpolation in the setup kernel for vertex0 of the line or triangle.
[varies]	31:0	Constant Data from CURBE URB Entry (optional)
varies	31:0	V0 Vertex Attribute (VUE) Data from URB (for all objects)
[varies]	31:0	V1 Vertex Attribute (VUE) Data from URB (for all objects except POINTs)
[varies]	31:0	V2 Vertex Attribute (VUE) Data from URB (for TRIANGLE and RECTANGLE objects only)

7.5.3 SF Thread Execution

The kernel that performs coefficient interpolation must be supplied by the jitter. As a usage note, it generally needs to loop through the entire set of vertex attributes, calculating a C0, Cx and Cy for each. It must take into account whether or not “wrap shortest” mode is on, if flat (rather than gouraud) shading has been selected, whether (separately for each attribute) interpolation should be done in a perspective correct manner, if point sprites are enabled, and must operate appropriately for the primitive type (triangle, line or point.)

7.5.4 SF Thread Output [DevBW, DevCL]

The SF thread must send a URB_WRITE to the URB shared function in order to pass results for use in subsequent PS threads spawned in the rasterization of the object. This information will be read from the URB as part of WM thread dispatch and thus included in the WM thread payload.

DWord	Bit	Description
M1.7	31:0	Cx[7] Gradient in X for attribute 7. Format = IEEE_Float
M1.6	31:0	Cx[6]
M1.5	31:0	Cx[5]
M1.4	31:0	Cx[4]
M1.3	31:0	Cx[3]
M1.2	31:0	Cx[2]
M1.1	31:0	Cx[1]
M1.0	31:0	Cx[0]
M2.7	31:0	Cy[7] Gradient in Y for attribute 7. Format = IEEE_Float
M2.6	31:0	Cy[6]
M2.5	31:0	Cy[5]
M2.4	31:0	Cy[4]
M2.3	31:0	Cy[3]
M2.2	31:0	Cy[2]
M2.1	31:0	Cy[1]
M2.0	31:0	Cy[0]
M3.7	31:0	Co[7] Value of attribute 7 at the start vertex (V0) Format = IEEE_Float
M3.6	31:0	Co[6]
M3.5	31:0	Co[5]
M3.4	31:0	Co[4]
M3.3	31:0	Co[3]
M3.2	31:0	Co[2]
M3.1	31:0	Co[1]

DWord	Bit	Description
M3.0	31:0	Co[0]
M4…		Additional attributes Additional attributes beyond the first 8 are sent in subsequent message registers following the same format as the first 8.

The message descriptor of this URB_WRITE message should set **Swizzle Control** to URB_TRANSPOSE in order to re-arrange the interpolation coefficients by attribute instead of by coefficient type (C0, Cx and Cy) as shown above. See *URB* chapter. This functionality is provided as a performance enhancement; the coefficient interpolation code could send the coefficients in the desired format, but having it re-arrange the coefficients is not as efficient as relying on this hardware mechanism.

Assuming the interpolation coefficient generation thread sent the preceding message with *SF to Windower transpose* swizzle, the resulting URB contents would look like this:

Co3	null	Cy3	Cx3	Co2	null	Cy2	Cx2	Co1	null	Cy1	Cx1	Co0	null	Cy0	Cx0
Co7	null	Cy7	Cx7	Co6	null	Cy6	Cx6	Co5	null	Cy5	Cx5	Co4	null	Cy4	Cx4

SetupURBOutput

This is the most efficient arrangement for the windower interpolation code (“jitted” code placed before the pixel shader).

Note: In order for the WM unit to read back Z plane equation coefficients (as it interpolates Z), the Setup thread must have those coefficients stored in the low-order 4 DWs of a URB row (corresponding to an even-numbered attribute in the diagram above).

7.5.5 SF Thread Output [DevCTG+]

The SF thread must send a URB_WRITE to the URB shared function in order to pass results for use in subsequent PS threads spawned in the rasterization of the object. This information will be read from the URB as part of WM thread dispatch and thus included in the WM thread payload (possibly only a portion of this is actually included in the WM thread payload depending on WM state fields).

7.5.5.1 Hardware Interpreted Fields

Certain fields output from the SF thread must appear in defined positions due to their use by WM hardware. The following table indicates the hardware interpreted fields and their position (attribute index) relative to the **Depth Coefficient URB Read Offset** (in WM unit’s state):

Attribute Index	Attribute	Projects
0	Z	all
1	1/W	all

Fields that will be used by hardware based on WM state fields must be computed by the SF thread and written to the position indicated in the table above. Fields not used by hardware but preceding other fields that are used by hardware need not be computed by the SF thread, but the position must still exist. Fields not used by hardware and preceding only other fields also not used by hardware need not have positions allocated. Attributes that are not interpreted by hardware (used by the WM thread) may follow the last field in the above table.

Programming Notes:

- **Z,W Plane Coefficients**: In order for the WM unit to read back Z plane equation coefficients (as it interpolates Z), the Setup thread must have those coefficients stored in the low-order 4 DWs of a URB row (corresponding to an even-numbered attribute in the diagram below). Also, the W plane equation coefficients must immediately follow the Z plane coefficients (in the high-order 4 DWs of the same URB row).

7.5.5.2 Transposed URB Read

A Transposed URB Read feature has been added. This feature is a performance enhancement over the previous transpose-on-write (URB_TRANSPOSE message to URB). Note that the transpose-on-write capability is still supported.

To use the transposed-read feature, the Setup kernel will write coefficients into the URB using the setup-friendly coefficient-major ordering as shown in Table 7-9. Note that this is the layout currently used for the URB_TRANSPOSE write message, but as the Setup thread will use the URB_NOSWIZZLE message, the data will be written to the URB unmodified (not transposed).

Table 7-9. Coefficient-Major (untransposed) Coefficient Write Data Layout

DWord								
Row	7	6 5 4 3 2 1 0						
0	Cx[7]	Cx[6]	Cx[5]	Cx[4]	Cx[3]	Cx[2]	Cx[1]	Cx[0]
1	Cy[7]	Cy[6]	Cy[5]	Cy[4]	Cy[3]	Cy[2]	Cy[1]	Cy[0]
2	Cz[7]	Cz[6]	Cz[5]	Cz[4]	Cz[3]	Cz[2]	Cz[1]	Cz[0]
3	Cx[15]	Cx[14]	Cx[13]	Cx[12]	Cx[11]	Cx[10]	Cx[9]	Cx[8]
4	Cy[15]	Cy[14]	Cy[13]	Cy[12]	Cy[11]	Cy[10]	Cy[9]	Cy[8]
5	Cz[15]	Cz[14]	Cz[13]	Cz[12]	Cz[11]	Cz[10]	Cz[9]	Cz[8]
…	...	...	...	...	...	...	...	...

When using transposed-read, the allocation size of the Setup URB entries should be sized according to this format. Note that this format requires less (3/4) URB storage than when transposed-write is employed.

Although physically stored in coefficient-major order, when transposed-read is enabled the coefficient data appears to consumers in the attribute-major (transposed/padded) format as if URB_TRANSPOSE had been used, as show below. URB Read Offset states associated with these URB entries should be programmed as if the data was actually stored in this format (i.e., with the same values as if URB_TRANSPOSE had been used).

Table 7-10. Attribute-Major (Transposed) Coefficient Read Data Layout

DWord								
Row	7	6 5 4 3 2 1 0						
0	Cz[1]	0	Cy[1]	Cx[1]	Cz[0]	0	Cy[0]	Cx[0]
1	Cz[3]	0	Cy[3]	Cx[3]	Cz[2]	0	Cy[2]	Cx[2]
2	Cz[5]	0	Cy[5]	Cx[5]	Cz[4]	0	Cy[4]	Cx[4]
3	Cz[7]	0	Cy[7]	Cx[7]	Cz[6]	0	Cy[6]	Cx[6]
4	Cz[9]	0	Cy[9]	Cx[9]	Cz[8]	0	Cy[8]	Cx[8]
5	Cz[11]	0	Cy[11]	Cx[11]	Cz[10]	0	Cy[10]	Cx[10]
6	Cz[13]	0	Cy[13]	Cx[13]	Cz[12]	0	Cy[12]	Cx[12]
7	Cz[15]	0	Cy[15]	Cx[15]	Cz[14]	0	Cy[14]	Cx[14]
…	…	…	…	…	…	…	…	…

The **Transposed URB Read Enable** (WM_STATE) is used to enable the transposed read. This bit will be forwarded along to the URB shared function to control the transpose operation when setup data is read from URB (upon PS dispatch and when the WM reads depth coefficients).

With this additional feature, software has three operating modes to get coefficient data to WM:

1. **Transpose-on-Read** (new): The Setup kernel passes data in coefficient-major order using URB_NOSWIZZLE. The URB SF writes the data in this format (no transpose). Transposed URB Read Enable must be ENABLED to cause the data to be transposed whenever it is read.
2. **Transpose-on-Write**: The Setup kernel passes data in coefficient-major order using URB_TRANSPOSE. The URB SF transposes the data as it is written. Transposed URB Read Enable must be DISABLED.
3. **Kernel-Transposed**: The Setup kernel can transpose the data programmatically, and then use URB_NOSWIZZLE to write the pre-transposed attrbitute-major data into the URB. Transposed URB Read Enable must be DISABLED.

7.5.6 Attribute Swizzling

The first or last set of 16 attributes can be swizzled according to certain state fields. **Attribute Swizzle Enable** enables the swizzling for all 16 of these attributes, and each of the attributes has a 2-bit **Swizzle Select** field that controls swizzling with the following settings:

- INPUTATTR – This attribute is sourced from AttrInputReg[SourceAttribute].
- INPUTATTR_FACING – This attribute is sourced from AttrInputReg[SourceAttribute] if the object is front-facing, otherwise it is sourced from AttrInputReg[SourceAttribute+1].
- INPUTATTR_W – This attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination).
- INPUTATTR_FACING – If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute]. WYZW (the W component of the source is copied to the X component of the destination). If the object is front-facing, this attribute is sourced from AttrInputReg[SourceAttribute+1]. WYZW.

Each of the first or last set of 16 attributes also has a 5-bit **Source Attribute** field which specify, per output attribute (not component), which input attribute sources the output attribute when INPUTATTR is selected for **Swizzle Select**. A **Source Attribute** value of 0 corresponds to the 128-bit attribute immediately following the vertex 4D position. If INPUTATTR_FACING is selected, this specifies the first of two consecutive (front,back) input attributes, where the SourceAttribute value can be an odd or even number (just not 31, as that would place the back-face input attribute past the end of the input max complement of input attributes).

Constant overriding is also available for the first or last set of 16 attributes. Each attribute has a **Constant Source** field which specifies the constant values per swizzled attribute, with the following settings available:

- XYZW = 0000
- XYZW = 0001
- XYZW = 1111

Each channel of each attribute has a **Component Override** field to control whether the corresponding channel is overridden with the constant value defined in **Constant Source**.

7.5.7 Interpolation Modes

All 32 attributes have a **Constant Interpolation Enable** state field bit to specify whether all components of the post-swizzled attribute are to be interpolated as constant values (not varying over the pixels of the object). If set, the attribute at the provoking vertex is copied to a0, and a1 and a2 are set to zero – this results in a

constant interpolation of the provoking vertex value. If clear, the attribute is linearly interpolated. Attributes 0-15 are further subjected to Wrap Shortest processing on a per-component basis, via the **Attribute WrapShortest Enables** state bitfields. WrapShortest processing modifies the a1 and/or a2 values depending on attribute deltas. All

The table below indicates the output values of a0, a1, and a2 depending on interpolation mode settings.

a0		a1	a2
Constant	A0	0.0	0.0
Linear	A0	A1-A0	A2-A0
Wrap Shortest	A0	(A1-A0)+1 (A1-A0) <= -0.5 (A1-A0)-1 (A1-A0) >= 0.5 (A1-A0) otherwise	(A2-A0)+1 (A2-A0) <= -0.5 (A2-A0)-1 (A2-A0) >= 0.5 (A2-A0) otherwise

7.5.8 Point Sprites

Normally all vertex attributes (including texture coordinates) other than position are simply replicated from the incoming point center vertex to the generated point object (corner) vertices. However OGL supports “sprite points”, where some/all texture coordinates are replaced with full-scale 2D texture coordinates.

A 32-bit **PointSprite TextureCoordinate Enable** bit mask controls whether the corresponding vertex attribute is to be replaced by a sprite point texture coordinate. The global (not per-attribute) **Point Sprite TextureCoordinate Origin** field controls how the point object vertex (top/bottom, left/right) texture coordinates are generated:

UPPERLEFT	Left	Right
Top	(0,0,0,1)	(1,0,0,1)
Bottom	(0,1,0,1)	(1,1,0,1)
LOWERLEFT	Left	Right
Top	(0,1,0,1)	(1,1,0,1)
Bottom	(0,0,0,1)	(1,0,0,1)

7.6 Other SF Functions

7.6.1 Statistics Gathering

The SF stage itself does not have any associated pipeline statistics; however, it counts the number of objects being output by the clipper on the clipper's behalf, since it less feasible to have the CLIP unit figure out how many objects have been output by a clip thread. It is easy for the SF unit to count the number of objects it receives from the CLIP stage since it is decomposing the output primitive topologies into objects anyway.

If the **Statistics Enable** bit is set in SF_STATE, then SF will increment the CL_PRIMITIVES_COUNT Register (see Memory Interface Registers in Volume Ia, *GPU*) once for each object in each primitive topology it receives from the CLIP stage. This bit should always be set if clipping is enabled and pipeline statistics are desired.

Software should always clear the **Statistics Enable** bit in SF_STATE if the clipper is disabled since objects SF receives are not considered "primitives output by the clipper" unless the clipper is enabled. Note that the clipper can be disabled either using bypass mode via a PIPELINE_STATE_POINTERS command with **Clip Enable** clear *or* by setting **Clip Mode** in CLIP_STATE to CLIPMODE_ACCEPT_ALL.

8. Windower (WM) Stage

8.1 Overview

As mentioned in the *SF Unit* chapter, the SF stage prepares an object for scan conversion by the Window/Masker (WM) unit. Refer to the *SF Unit* chapter for details on the screen-space geometry of objects to be rendered. The WM unit uses the parameters provided by the SF unit in the object-specific rasterization algorithms.

The WM stage of the GENx 3D pipeline performs the following operations (at a high level)

- Pre-scan-conversion modification of some primitive attributes, including
  - Application of Depth Offset to the position Z attribute
- Scan-conversion of the various primitive types, including
  - 2D clipping to the scissor/draw rectangle intersection
- Spawning of Pixel Shader (PS) threads to process the pixels resulting from scan-conversion

The spawned Pixel Shader (PS) threads are responsible for the following (high-level) operations

- interpolation of vertex attributes (other than X,Y,Z) to the pixel location
- performing any “Pixel Shader” operations dictated by the API PS program
  - Using the Sampler shared function to sample data from “texture” surfaces
  - Using the DataPort to perform general memory I/O
- Submitting the shaded pixel results to the DataPort for any subsequent “blending” (aka Output Merger) operation and write to the RenderCache.

The WM unit keeps a scoreboard of pixels being processed in outstanding PS threads in order to guarantee in-order rasterization results. This allows the WM unit to overlap processing of several objects.

8.1.1 Inputs from SF to WM

The outputs from the SF stage to the WM stage are mostly comprised of implementation-specific information required for the rasterization of objects. The types of information is summarized below, but as the interface is not exposed to software a detailed discussion is not relevant to this specification.

- PrimType of the object
- VPIndex, RTAIndex associated with the object
- Handle of the Primitive URB Entry (PUE) that was written by the SF (Setup) thread. This handle will be passed to all WM (PS) threads spawned from the WM's rasterization process.
- Information regarding the X,Y extent of the object (e.g., bounding box, etc.)
- Edge or line interpolation information (e.g., edge equation coefficients, etc.)
- Information on where the WM is to start rasterization of the object
- Object orientation (front/back-facing)
- Last Pixel indication (for line drawing)

8.2 Windower Pipelined State

8.2.1 WM_STATE [Pre-DevSNB]

DWord	Bit	Description
0	31:6	**Kernel Start Pointer[0]:** Specifies the 64-byte aligned address offset of the first instruction in the kernel[0]. This pointer is relative to the **General State Base Address [Pre-DevILK]** or **Instruction Base Address [DevILK]**. **[DevBW] Errata BWT007:** Instructions pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) **[Pre-DevILK]:** Format = GeneralStateOffset[31:6] **[DevILK]:** Format = InstructionBaseOffset[31:6]
	5:4	Reserved : MBZ
	3:1	**GRF Register Count[0]:** Defines the number of GRF Register Blocks used by the kernel[0]. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. Format = U3 register block count - 1 Range = [0,7] corresponding to [1,8] 16-register blocks
	0	Reserved : MBZ
1	31	**Single Program Flow (SPF) :** Specifies whether the kernel program has a single program flow (SIMDnxm with m = 1) or multiple program flows (SIMDnxm with m > 1). See CR0 description in *ISA Execution Environment*. 0: Multiple Program Flows 1: Single Program Flow

DWord Bit		Description
	30:26	Reserved : MBZ
	25:18	**Binding Table Entry Count:** Specifies how many binding table entries the kernel uses. Used only for prefetching of the binding table entries and associated surface state. **Note:** for kernels using a large number of binding table entries, it may be wise to set this field to zero to avoid prefetching too many entries and thrashing the state cache. Format = U8 Range = [0,255] [DevILK]-A,B] MBZ
	17	**Thread Priority:** Specifies the priority of the thread for dispatch 0: Normal Priority 1: High Priority **Programming Notes:** • **[Pre-DevILK]**: this field must be zero.
	16	**Floating Point Mode:** Specifies the floating point mode used by the dispatched thread. 0: Use IEEE-754 Rules 1: Use alternate rules
	15:14	Reserved : MBZ
	13:8	**Depth Coefficient URB Read Offset:** Specifies the offset (in 256-bit units) at which the depth coefficient URB data is to be read from the URB and used by the FF to interpolate depth. The WM unit interprets the low order 128 bits of this URB row as containing the plane coefficients of Z depth. This places a restriction on the Setup thread to write the URB in such a way as to place these coefficients in the low order DWords of a URB row. See *Strip and Fan Unit* and *URB* chapters for details on Setup threads and the TRANSPOSED URB write operation. **[DevCTG+]:** This offset is programmed according to the transposed (attribute-major) layout, regardless of the setting of **Transposed URB Read Enable** Format = U6 Range = [0,63]
	7:5	Reserved : MBZ
	4	**Illegal Opcode Exception Enable.** This bit gets loaded into EU CR0.1[12] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable
	3	Reserved : MBZ
	2	**MaskStack Exception Enable.** This bit gets loaded into EU CR0.1[11]. See *Exceptions* and *ISA Execution Environment*. Format: Enable
	1	**Software Exception Enable.** This bit gets loaded into EU CR0.1[13] (note the bit # difference). See *Exceptions* and *ISA Execution Environment*. Format: Enable

DWord	Bit	Description
	0	Reserved : MBZ
2	31:10	**Scratch Space Base Pointer:** Specifies the 1k-byte aligned address offset to scratch space for use by the kernel. This pointer is relative to the **General State Base Address**. **Programming Note:** • **[DevBW-A] A0 Erratum BWT005**: If **Per Thread Scratch Space** is programmed to 256KB, this pointer must be 8M-aligned. Format = GeneralStateOffset[31:10]
	9:4	Reserved : MBZ
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by each thread. The driver must allocate enough contiguous scratch space, pointed to by the Scratch Space Pointer, to ensure that the Maximum Number of Threads each get Per Thread Scratch Space size without exceeding the driver-allocated scratch space. Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two **Programming Note:** • **[DevBW-A] A0 Erratum BWT005**: The range [0,11] for this register indicates [1KB, 12KB] in 1K byte increments. If MMIO register 21D0h bit 3 is set, then value 11 is an exception and indicates a 256KB space instead of 12KB. Note that Scratch Space Base Pointer must be 8MB-aligned in order to set the 256KB scratch space. Format = U4
3	31	Reserved : MBZ
	30:25	**Constant URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for the Constant URB entry, in 256-bit register increments. Format = U6 Range = [0,63]
	24	Reserved : MBZ
	23:18	**Constant URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Constant URB data is to be read from the URB before being included in the thread payload. Format = U6 Range = [0,63]
	17:11	**Setup URB Entry Read Length:** Specifies the amount of URB data read and passed in the thread payload for each Setup URB entry, in 256-bit register increments. **Programming Notes:** • It is UNDEFINED to set this field to 0 indicating no Setup URB data to be read and passed to the PS thread. Format = U7 **[Pre-DevILK]** Range = [1,63] **[DevILK]:** Range = [1,65]
	10	Reserved : MBZ

DWord	Bit	Description
	9:4	**Setup URB Entry Read Offset:** Specifies the offset (in 256-bit units) at which Setup URB data is to be read from the URB before being included in the thread payload. This offset applies to all Setup URB entries passed to the thread. **[DevCTG+]:** This offset is programmed according to the transposed (attribute-major) layout, regardless of the setting of **Transposed URB Read Enable** Format = U6 Range = [0,63]
	3:0	**Dispatch GRF Start Register for URB Data:** Specifies the starting GRF register number for the URB portion (Constant + Setup) of the thread payload. Format = U4 Range = [0,15] **[DevBW,DevCL]**: If 32 pixel dispatch is enabled, the maximum range is [0,7]
4	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned address offset of the sampler state table. This pointer is relative to the **General State Base Address**. **[DevBW-A] Errata BWT007:** Sampler state pointed at by offsets from General State Base must be contained within 32-bit physical address space (that is, must map to memory pages under 4G.) Format = GeneralStateOffset[31:5]
	1	Reserved : MBZ
	0	**Statistics Enable:** If ENABLED, the Windower will engage in statistics gathering. If DISABLED, statistics information associated with this FF stage will be left unchanged. See *Statistics Gathering*. **Programming Notes:** • If this field is enabled, **Statistics Enable** in CC_STATE should also be set, and when this field is disabled, **Statistics Enable** in CC_STATE should also be clear. Both functions contribute to the PS_DEPTH_COUNT, so having either one set without the other set will result in an UNPREDICTABLE value for PS_DEPTH_COUNT. • **[DevBW-A] A0 Erratum BWT004**: If no pixel shader is desired (a "null" pixel shader), this bit must be *cleared* so that PS_INVOCATIONS will not be incremented for the "dummy" PS dispatches. • [DevILK]: This bit must be disabled if either of these bits is set: **Depth Buffer Clear, Hierarchical Depth Buffer Resolve Enable or Depth Buffer Resolve Enable.** Format = Enabled
5	31:25	**Maximum Number of Threads:** Specifies the maximum number of simultaneous threads allowed to be active. Used to avoid using up the scratch space, or to avoid potential deadlock. Format = U7 representing (thread count – 1) Range = [0, n-1] where n = (# EUs) * (# threads/EU). See *Graphics Processing Engine* for listing of #EUs and #threads in each device.

DWord Bit		Description
	24	**[DevCTG+]:** **Transposed URB Read Enable**: If set, coefficient data will be transposed whenever it is read from the URB. The Setup thread must write the data using URB_NOSWIZZLE, with the payload data in coefficient-major order. The data will be written untransposed, and later transposed by the URB SF upon being read (either in PS thread dispatch or when the WM reads depth coefficients). If clear, the URB data will be read directly (no transpose on read). This mode should be used when the Setup thread (a) uses URB_TRANSPOSED to write the URB (in which case the data is transposed before being written into the URB) or (b) uses URB_NOSWIZZLE to write kernel-pre-transposed data. **Programming Notes:** • This state bit must be programmed to match the operation of the current Setup kernel. • When enabled, the URB Allocation Size of the Setup URB entry should be programmed according to the coefficient-major layout. • **[DevCTG-A] Errata:** If this field is set, CURBE data into the pixel shader is also transposed on dispatch. • URB Read Offsets must always be programmed according to the transposed (attribute-major) layout, regardless of the setting of this bit. **[DevBW,DevCL]:** Reserved : MBZ
	23	**Legacy Diamond Line Rasterization:** This bit, if ENABLED, indicates that the Windower will rasterize zero width lines using the rasterization rules. If DISABLED, the Windower will rasterize zero width lines using the rasterization rules (see *Strips Fans* chapter). Format = Enable
	22	**Pixel Shader Kill Pixel:** This bit, if ENABLED, indicates that the PS kernel has the ability to kill (discard) pixels, e.g., as required by the presence of a “killpix” or “discard” instruction in the API PS program, or JITTER-introduced code to kill pixels due to ClipDistance clipping. If DISABLED, the PS kernel may not, under any circumstances, kill pixels. This bit must also be ENABLED if a sampler has chroma key enabled with kill pixel mode. Format = Enable
	21	**Pixel Shader Computed Depth:** This bit, if ENABLED, indicates that the PS kernel computes a depth value. It is used to disable the depth/stencil test in the Early Depth Test function. Format = Enable **Programming Notes:** • If a NULL Depth Buffer is selected, the **Pixel Shader Computed Depth** field must be set to disabled. • **[DevBW-A] Errata:** If both **Depth Test Enable** and **Depth Write Enable** are disabled, this field must be disabled.
	20	**Pixel Shader Uses Source Depth:** This bit, if ENABLED, indicates that the PS kernel requires the source depth value (vPos.z) to be passed in the payload. Format = Enable

DWord Bit		Description
	19	**Thread Dispatch Enable:** This bit, if set, indicates that it is possible for a PS thread to modify a render target, i.e.,at least one render target is enabled (is not of type SURFTYPE_NULL and has at least one channel enabled for writes) and the PS kernel contains a code path that may issue a write to that/those enabled RTs. **Programming Notes:** • This bit is used for performance optimizations and does not directly control writing to render targets. If this bit is DISABLED, no pixel shader threads will be dispatched.. • For correct behavior, this bit must be set consistently with the behavior of the PS kernel, i.e. if this bit is DISABLED the PS kernel must not write color or depth to any render targets. Format = Enable
	18	**Early Depth Test Enable:** This bit enables the Early Depth Test (aka Intermediate Z, or IZ) function. Note: This bit should always be ENABLED – at least there are no known conditions underwhich disabling the Early Depth Test is required. Format = Enable
	17:16	**Line End Cap Antialiasing Region Width:** This field specifies the distances over which the coverage of anti-aliased line end caps are computed. Format = 0: 0.5 pixels 1: 1.0 pixels 2: 2.0 pixels 3: 4.0 pixels **Note:** This state is duplicated in the SF_STATE state descriptor
	15:14	**Line Antialiasing Region Width:** This field specifies the distance over which the anti-aliased line coverage is computed. Format = 0: 0.5 pixels 1: 1.0 pixels 2: 2.0 pixels 3: 4.0 pixels
	13	**Polygon Stipple Enable:** Enables the Polygon Stipple function. Format = Enable
	12	**Global Depth Offset Enable:** Enables computation and application of Global Depth Offset. Format = Enable
	11	**Line Stipple Enable:** Enables the Line Stipple function. Format = Enable

DWord Bit		Description
	10	**Legacy Global Depth Bias Enable:** Enables the Windower to use the Global Depth Offset Constant state unmodified. If this bit is not set, the Windower will scale the Global Depth Offset Constant as described in section 1.4.2 of this document. Format = Enable
	9	**Hierarchical Depth Buffer Resolve Enable [DevILK]** When set, the hierarchical depth buffer is made to be consistent with the depth buffer as a side-effect of rendering pixels. This is intended to be used when the depth buffer has been modified outside of the 3D rendering operation. Format = Enable **Programming Notes:** • If this field is enabled, the **Depth Buffer Clear** and **Depth Buffer Resolve Enable** fields must both be disabled. Refer to section 8.4.4.3 “Hierarchical Depth Buffer Resolve” for additional restrictions when this field is enabled. • If **Hierarchical Depth Buffer Enable** is disabled, enabling this field will have no effect. • **Performance Note:** expect the hierarchical depth buffer’s impact on performance to be reduced for some period of time after this operation is performed, as the hierarchical depth buffer is initialized to a state that makes it ineffective. Further rendering will tend to bring the hierarchical depth buffer back to a more effective state. **[Pre-DevILK]:** Reserved : MBZ
	8	**Depth Buffer Resolve Enable [DevILK]** When set, the depth buffer is made to be consistent with the hierarchical depth buffer as a side-effect of rendering pixels. This is intended to be used when the depth buffer is to be used as a surface outside of the 3D rendering operation. Fomat = Enable **Programming Notes:** • If this field is enabled, the **Depth Buffer Clear** and **Hierarchical Depth Buffer Resolve Enable** fields must both be disabled. Refer to section “Depth Buffer Resolve” for additional restrictions when this field is enabled. • If **Hierarchical Depth Buffer Enable** is disabled, enabling this field will have no effect. **[Pre-DevILK]:** Reserved : MBZ
	7	**Depth Buffer Clear [DevILK]** When set, the depth buffer is initialized as a side-effect of rendering pixels. Format = Enable **Programming Notes:** • If this field is enabled, the **Depth Test Enable** field in COLOR_CALC_STATE must be disabled. • Refer to section 0 “Depth Buffer Clear” for additional restrictions when this field is enabled. **[Pre-DevILK]:** Reserved : MBZ

DWord Bit		Description
	6	**Fast Span Coverage Enable [DevILK]** When set, all aligned 4X4 pixel blocks rasterized will be fully covered (4X4). Format = Enable **Programming Notes:** • If this field is enabled, the **Depth Buffer Clear** must be enabled. • This field should only be enabled if the rectangle being rendered is aligned on all edges with a 4x4 pixel grid. **[Pre-DevILK]:** Reserved : MBZ
	5	Reserved : MBZ
	4	**Contiguous 64-Pixel Dispatch Enable:** (**[DevCTG+]** only) Enables the Windower to dispatch 16 subspans in one payload, contiguous as a 4x4 block of subspans (8x8 block of pixels) only. 0: Contiguous 64 pixel dispatch disabled 1: Contiguous 64 pixel dispatch enabled Note: See For [DevILK+], each of the four KSP values is separately specified. In **addition,** each kernel has a separately-specified GRF register count, whereas on [Pre-DevILK], all **kernels** share the same GRF register count field, with the one with the maximum register count required applying to all. Table 8-1 for valid pixel dispatch combinations. This field is Reserved : MBZ on [DevBW] and [DevCL]
	3	**Contiguous 32 Pixel Dispatch Enable:** (**[DevCTG+]** only) Enables the Windower to dispatch 8 subspans in one payload, contiguous as a 4x2 block of subspans (8x4 block of pixels) only. 0: Contiguous 32 pixel dispatch disabled 1: Contiguous 32 pixel dispatch enabled Note: See For [DevILK+], each of the four KSP values is separately specified. **In** addition, each kernel has a separately-specified GRF register count, whereas on [Pre-DevILK], **all** kernels share the same GRF register count field, with the one with the maximum register count required applying to all. Table 8-1 for valid pixel dispatch combinations. This field is Reserved : MBZ on [DevBW] and [DevCL]

DWord Bit		Description
	2	**32 Pixel Dispatch Enable:** Enables the Windower to dispatch 8 subspans in one payload 0: 32 pixel dispatch disabled 1: 32 pixel dispatch enabled Note: See For [DevILK+], each of the four KSP values is separately specified. **In** addition, each kernel has a separately-specified GRF register count, whereas on [Pre-DevILK], **all** kernels share the same GRF register count field, with the one with the maximum register count required applying to all. Table 8-1 for valid pixel dispatch combinations.
	1	**16 Pixel Dispatch Enable:** Enables the Windower to dispatch 4 subspans in one payload (typical operation) 0: 16 pixel dispatch disabled 1: 16 pixel dispatch enabled Note: See For [DevILK+], each of the four KSP values is separately specified. **In** addition, each kernel has a separately-specified GRF register count, whereas on [Pre-DevILK], **all** kernels share the same GRF register count field, with the one with the maximum register count required applying to all. Table 8-1 for valid pixel dispatch combinations.
	0	**8 Pixel Dispatch Enable:** Enables the Windower to dispatch 2 subspans in one payload 0: 8 pixel dispatch disabled 1: 8 pixel dispatch enabled Note: See For [DevILK+], each of the four KSP values is separately specified. In addition, each kernel has a separately-specified GRF register count, whereas on **[Pre-DevILK],** all kernels share the same GRF register count field, with the one with the maximum register count required applying to all. Table 8-1 for valid pixel dispatch combinations.

DWord	Bit	Description
6	31:0	**Global Depth Offset Constant:** Specifies the constant term in the GlobalDepthOffset function. Format = IEEE_FP
7	31:0	**Global Depth Offset Scale:** This field specifies the *GlobalDepthOffsetScale* term used in the Global Depth Offset Function Format = IEEE_FP
		Following DWords for **[DevILK]** Only
8	31:6	**Kernel Start Pointer[1]:** Specifies the 64-byte aligned address offset of the first instruction in kernel[1]. This pointer is relative to the **General State Base Address [Pre-DevILK] or Instruction Base Address [DevILK]**. **[Pre-DevILK]:** Format = GeneralStateOffset[31:6] **[DevILK]:** Format = InstructionBaseOffset[31:6]
	5:4	Reserved : MBZ
	3:1	**GRF Register Count[1]:** Defines the number of GRF Register Blocks used by kernel[1]. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. Format = U3 register block count - 1 Range = [0,7] corresponding to [1,8] 16-register blocks
	0	Reserved : MBZ
9	31:6	**Kernel Start Pointer[2]:** Specifies the 64-byte aligned address offset of the first instruction in kernel[2]. This pointer is relative to the **General State Base Address [Pre-DevILK] or Instruction Base Address [DevILK]**. **[Pre-DevILK]:** Format = GeneralStateOffset[31:6] **[DevILK]:** Format = InstructionBaseOffset[31:6]
	5:4	Reserved : MBZ
	3:1	**GRF Register Count[2]:** Defines the number of GRF Register Blocks used by kernel[2]. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. Format = U3 register block count - 1 Range = [0,7] corresponding to [1,8] 16-register blocks
	0	Reserved : MBZ
10	31:6	**Kernel Start Pointer[3]:** Specifies the 64-byte aligned address offset of the first instruction in kernel[3]. This pointer is relative to the **General State Base Address [Pre-DevILK]nstruction Base Address [DevILK]**. **[Pre-DevILK]:** Format = GeneralStateOffset[31:6] **[DevILK]:** Format = InstructionBaseOffset[31:6]
	5:4	Reserved : MBZ

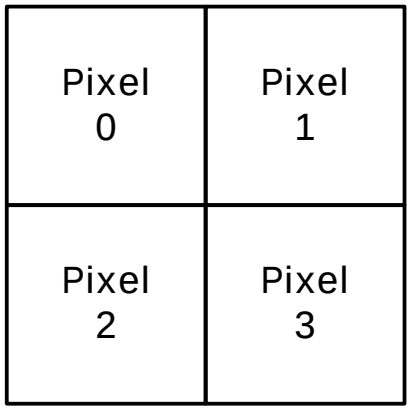
DWord Bit		Description
	3:1	**GRF Register Count[3]:** Defines the number of GRF Register Blocks used by kernel[3]. A register block contains 16 registers. A kernel using a register count that is not a multiple of 16 must round up to the next multiple of 16. Format = U3 register block count - 1 Range = [0,7] corresponding to [1,8] 16-register blocks
	0	Reserved : MBZ

8.3 Rasterization

The WM unit uses the setup computations performed by the SF unit to rasterize objects into the corresponding set of pixels. Most of the controls regarding the screen-space geometry of rendered objects are programmed via the SF unit.

The rasterization process generates pixels in 2x2 groups of pixels called *subspans* (see Figure 8-1) which, after being subjected to various inclusion/discard tests, are grouped and passed to spawned Pixel Shader (PS) threads for subsequent processing. Once these PS threads are spawned, the WM unit provides only bookkeeping functions on the pixels. Note that the WM unit can proceed on to rasterize subsequent objects while PS threads from previous objects are still executing.

Figure 8-1. Pixels with a SubSpan



8.3.1 Drawing Rectangle Clipping

The Drawing Rectangle defines the maximum extent of pixels which can be rendered. Portions of objects falling outside the Drawing Rectangle will be clipped (pixels discarded). Implementations will typically discard objects falling completely outside of the Drawing Rectangle as early in the pipeline as possible. There is no control to turn off Drawing Rectangle clipping – it is unconditional.

For the purposes of clipping, the Drawing Rectangle must itself be clipped to the destination buffer extents. (The Drawing Rectangle Origin, used to offset relative X,Y coordinates earlier in the pipeline, is permitted to lie offscreen). The **Clipped Drawing Rectangle X,Y Min,Max** state variables (programmed via 3DSTATE_DRAWING_RECTANGLE – See *SF Unit*) defines the intersection of the Drawing Rectangle and the Color Buffer. It is specified with non-negative integer pixel coordinates relative to the Destination Buffer upper-left origin.

Pixels with coordinates outside of the Drawing Rectangle cannot be rendered (i.e., the rectangle is inclusive). For example, to render to a full-screen 1280x1024 buffer, the following values would be required: Xmin=0, Ymin=0, Xmax=1279 and Ymax=1023.

For "full screen" rendering, the Drawing Rectangle coincides with the screen-sized buffer. For "front-buffer windowed" rendering it coincides with the destination "window".

8.3.2 Line Rasterization

See *SF Unit* chapter for details on the screen-space geometry of the various line types.

8.3.2.1 Coverage Values for Anti-Aliased Lines

The WM unit is provided with both the **Line Anti-Aliasing Region Width** and **Line End Cap Anti-aliasing Region Width** state variables (in WM_STATE) in order to compute the coverage values for anti-aliased lines.

8.3.2.2 3DST ATE_AA_LINE_PARAMS [DevCTG+]

3DSTATE_AA_LINE_PARAMETERS			
Project:	[DevCTG+]	**Length Bias:**	2
The 3DSTATE_AA_LINE_PARAMS command is used to specify the slope and bias terms used in the improved alpha coverage computation (specifically for WHQL compliance). Note that in these devices the coverage values passed to PS threads are full U0.8 values, versus [DevBW]/[DevCL] where U0.4 values are passed.			

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 0Ah 3DSTATE_AA_LINE_PARAMS Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:24	**Reserved** Project: All Format: MBZ
	23:16	**AA Coverage Bias** Project: All Format: U0.8 FormatDesc This field specifies the bias term to be used in the aa coverage computation for edges 0 and 3.

3DSTATE_AA_LINE_PARAMETERS		
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**AA Coverage Slope** Project: All Format: U0.8 FormatDesc This field specifies the slope term to be used in the aa coverage computation for edges 0 and 3. If this field is zero, the Windower will revert to legacy aa line coverarge computation (though still output expanded U0.8 coverage values).
2	31:24	**Reserved** Project: All Format: MBZ
	23:16	**AA Coverage EndCap Bias** Project: All Format: U0.8 FormatDesc This field specifies the bias term to be used in the aa coverage computation for edges 1 and 2.
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**AA Coverage EndCap Slope** Project: All Format: U0.8 FormatDesc This field specifies the slope term to be used in the aa coverage computation for edges 1 and 2.

The slope and bias values should be computed to closely match the reference rasterizer results. Based on empirical data, the following recommendations are offered:

The final alpha for the center of the line needs to be 148 to match the reference rasterizer. In this case, the Lo to edge 0 and edge 3 will be the same. Since the alpha for each edge is multiplied together, we get:

edge0alpha * edge1alpha = 148/255 = 0.580392157

Since edge0alpha = edge3alpha we get:

$(\text{edge0alpha})^2 = 0.580392157$

edge0alpha = sqrt(0.580392157) = 0.761834731 at the center pixel

The desired alpha for pixel 1 = 54/255 = 0.211764706

The slope is (0.761834731 – 0.211764706) = 0.550070025

Since we are using 8 bit precision, the slope becomes

AA Coverage [EndCap] Slope = 0.55078125

The alpha value for Lo = 0 (second pixel from center) determines the bias term and is equal to

(0.211764706 – 0.550070025) = -0.338305319

With 8 bits of precision the programmed bias value

AA Coverage [EndCap] Bias = 0.33984375

8.3.2.3 Line Stipple

Line stipple, controlled via the **Line Stipple Enable** state variable in WM_STATE, discards certain pixels that are produced by non-AA line rasterization.

The line stipple rule is specified via the following state variables programmed via 3DSTATE_LINE_STIPPLE: the 16-bit **Line Stipple Pattern** (p), **Line Stipple Repeat Count** I, and **Line Stipple Inverse Repeat Count.** Sofware must compute **Line Stipple Inverse Repeat Count** as 1.0f / **Line Stipple Repeat Count** and then converted from float to the required fixed point encoding (see 3STATE_LINE_STIPPLE).

The WM unit maintains an internal Line Stipple Counter state variable (s). The initial value of s is zero; s is incremented after production of each pixel of a line segment (pixels are produced in order, beginning at the starting point and working towards the ending point). s is reset to 0 whenever a new primitive is processed (unless the primitive type is LINESTRIP_CONT or LINESTRIP_CONT_BF), and before every line segment in a group of independent segments (LINELIST primitive).

During the rasterization of lines, the WM unit computes:

$$b = \lfloor s/r \rfloor \bmod 16,$$

A pixel is rendered if the bth bit of p is 1, otherwise it is discarded. The bits of p are numbered with 0 being the least significant and 15 being the most significant.

8.3.2.4 3DST ATE_LINE_STIPPLE

3DSTATE_LINE_STIPPLE		
Project:	**Length Bias:**	2
The 3DSTATE_LINE_STIPPLE command is used to specify state variables used in the Line Stipple function.		

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 08h 3DSTATE_LINE_STIPPLE Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31	**Modify Enable (Current Repeat Counter, Current Stipple Index)** Project: All Format: Enable FormatDesc Modify enable for **Current Repeat Counter** and **Current Stipple Index** fields. **Programming Notes** Software should never set this field to enabled. It is provided only for HW-generated commands as part of context save/restore.
	30	**Reserved** Project: All Format: MBZ
	29:21	**Current Repeat Counter** Project: All Format: U9 FormatDesc This field sets the HW-internal repeat counter state. Note: Software should never attempt to set this value – this state is only provided for HW-generated commands as part of context save/restore.
	20	**Reserved** Project: All Format: MBZ

3DSTATE_LINE_STIPPLE		
	19:16	**Current Stipple Index** Project: All Format: U4 FormatDesc This field sets the HW-internal stipple pattern index. Note: Software should never attempt to set this value – this state is only provided for HW-generated commands as part of context save/restore.
	15:0	**Line Stipple Pattern** Project: All Format: 16 bit mask. Bit 15 = most significant bit, Bit 0 = least significant bit FormatDesc Specifies a pattern used to mask out bit specific pixels while rendering lines.
2	31:16	**Line Stipple Inverse Repeat Count** Project: All Format: U1.13 FormatDesc Range [0.00390625, 1.0] Specifies the inverse (truncated) of the repeat count for the line stipple function.
	15:9	**Reserved** Project: All Format: MBZ
	8:0	**Line Stipple Repeat Count** Project: All Format: U9 FormatDesc Range [1, 256] Specifies the repeat count for the line stipple function.

8.3.3 Polygon (Triangle and Rectangle) Rasterization

The rasterization of LINE, TRIANGLE, and RECTANGLE objects into pixels requires a “pixel sampling grid” to be defined. This grid is defined as an axis-aligned array of pixel sample points spaced exactly 1 pixel unit apart. If a sample point falls within one of these objects, the pixel associated with the sample point is considered “inside” the object, and information for that pixel is generated and passed down the pipeline.

For TRIANGLE and RECTANGLE objects, if a sample point intersects an edge of the object, the associated pixel is considered “inside” the object if the intersecting edge is a “left” or “top” edge (or, more exactly, the intersected edge is not a “right” or “bottom” edge). Note that “top” and “bottom” edges are by definition exactly horizontal. The following diagram identifies the edge types for representative TRIANGLE and RECTANGLE objects (solid edges are inclusive, dashed edges are exclusive).

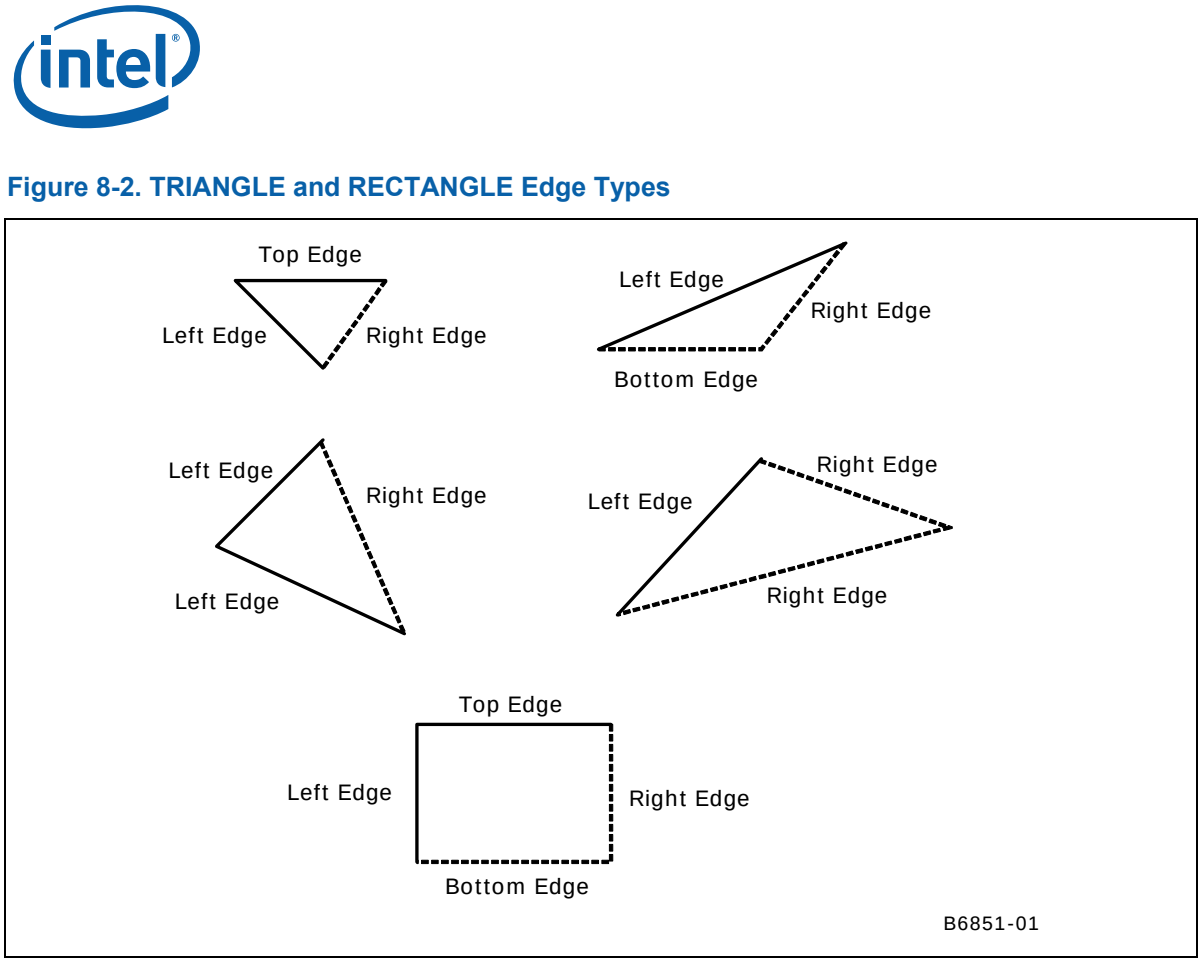


Figure 8-2. TRIANGLE and RECTANGLE Edge Types

8.3.3.1 Pol ygon Stipple

The *Polygon Stipple* function, controlled via the **Polygon Stipple Enable** state variable in WM_STATE, allows only selected pixels of a repeated 32x32 pixel pattern to be rendered. Polygon stipple is applied only to the following primitive types:

3DPRIM_POLYGON
3DPRIM_TRIFAN
3DPRIM_TRILIST
3DPRIM_TRISTRIP
3DPRIM_TRISTRIP_REVERSE

Note that the 3DPRIM_TRIFAN_NOSTIPPLE object is never subject to polygon stipple.

The stipple pattern is defined as a 32x32 bit pixel mask via the 3DSTATE_POLY_STIPPLE_PATTERN command. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

The origin of the pattern is specified via **Polygon Stipple X,Y Offset** state variables programmed via the 3DSTATE_POLY_STIPPLE_OFFSET command. The offsets are pixel offsets from the Color Buffer origin to the upper left corner of the stipple pattern. This is a non-pipelined command which incurs an implicit pipeline flush when executed.

8.3.3.2 3DST ATE_POLY_STIPPLE_OFFSET

3DSTATE_POLY_STIPPLE_OFFSET			
Project:	All	**Length Bias:**	2

The 3DSTATE_POLY_STIPPLE_OFFSET command is used to specify the origin of the repeated screen-space Polygon Stipple Pattern as an X,Y offset from the Color Buffer origin.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 06h 3DSTATE_POLY_STIPPLE_OFFSET Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:13	**Reserved** Project: All Format: MBZ
	12:8	**Polygon Stipple X Offset** Project: All Format: U5 FormatDesc Range [0,31] Specifies a 5 bit x address offset in the poly stipple pattern
	7:5	**Reserved** Project: All Format: MBZ
	4:0	**Polygon Stipple Y Offset** Project: All Format: U5 FormatDesc Range [0,31] Specifies a 5 bit y address offset in the poly stipple pattern

8.3.3.3 3DST ATE_POLY_STIPPLE_PATTERN

3DSTATE_POLY_STIPPLE_PATTERN			
Project:	All	**Length Bias:**	2

The 3DSTATE_POLY_STIPPLE_PATTERN command is used to specify the 32x32 Polygon Stipple Pattern used in the Polygon Stipple function of the WM unit.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 07h 3DSTATE_POLY_STIPPLE_PATTERN Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 1Fh Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:0	**Polygon Stipple Pattern Row 1 (top most)** Project: All Format: 32 bit mask. Bit 31 = upper left corner, Bit 0 = upper right corner of first row. FormatDesc Specifies a pattern used by Polygon Stipple to mask out specific pixels of every 32x32 area rendered.
2..32	31:0	**Polygon Stipple Pattern Rows 2-32 (bottom most)** Project: All Format: 32 bit mask. Bit 31 = upper left corner, Bit 0 = upper right corner of first row. FormatDesc Specifies a pattern used by Polygon Stipple to mask out specific pixels of every 32x32 area rendered.

8.3.3.4 3DST ATE_GLOBAL_DEPTH_OFFSET_CLAMP [Pre-DevSNB]

3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP			
Project:	[Pre-DevSNB]	**Length Bias:**	2

The 3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP command is used to specify the clamp used in the depth bias function of the WM unit.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 09h 3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:0	**Global Depth Offset Clamp** Project: All Format: IEEE_FP FormatDesc This field specifies the *GlobalDepthOffsetClamp* term used in the Global Depth Offset Function

8.3.4 Multisample Modes/State

A number of state variables control the operation of the multisampling function. The following list indicates the state and their location. Refer to the state definition for more details.

- **Multisample Rasterization Mode** (3DSTATE_SF and 3DSTATE_WM): controls whether rasterization of non-lines is performed on a pixel or sample basis (PIXEL vs. PATTERN), and whether rasterization of lines is performed on a pixel or sample basis (OFF vs. ON). The table below details the possible values of this state:

Multisample Rasterization Mode	Description
MSRASTMODE_ OFF_PIXEL	**All object types:** Rasterization is performed on a pixel (vs. sample) basis. The number of pixel sample points is determined by **Number of Multisamples**, but the location(s) are all fixed at either the pixel center or UL corner, as defined by **Pixel Location** (3DSTATE_MULTISAMPLE). The programmed values **Sample Offset** states are ignored. **Lines:** Multisampling rasterization of lines is turned off, allowing 0-width lines, french-cut wide/stippled lines, and AA lines.
MSRASTMODE_ OFF_PATTERN	This mode is only valid when **Number of Multisamples** = NUMSAMPLES_4. **Non-Lines:** Rasterization is performed on a 4X sample basis. The four pixel sample points are completely defined by state variables programmed via 3DSTATE_MULTISAMPLE. **Lines:** Rasterization is performed on a pixel (vs. sample) basis. The number of pixel sample points is determined by **Number of Multisamples**, but the location(s) are all fixed at either the pixel center or UL corner, as defined by **Pixel Location** (3DSTATE_MULTISAMPLE). The programmed values **Sample Offset** states are ignored. Multisampling rasterization of lines is turned off, allowing 0-width lines, french-cut wide/stippled lines, and AA lines.
MSRASTMODE_ ON_PIXEL	**All object types:** Rasterization is performed on a pixel (vs. sample) basis. The number of pixel sample points is determined by **Number of Multisamples**, but the location(s) are all fixed at either the pixel center or UL corner, as defined by **Pixel Location** (3DSTATE_MULTISAMPLE). The programmed values **Sample Offset** states are ignored. **Lines:** Multisampling rasterization of lines is turned on, where all lines are drawn as rectangles using **Line Width**.
MSRASTMODE_ ON_PATTERN	This mode is only valid when **Number of Multisamples** = NUMSAMPLES_4. **All object types:** Rasterization is performed on a 4X sample basis. The four pixel sample points are completely defined by state variables programmed via 3DSTATE_MULTISAMPLE. **Lines:** Multisampling rasterization of lines is turned on, where lines are drawn as rectangles using **Line Width**.

- **Multisample Dispatch Mode** (3DSTATE_WM): controls whether the pixel shader is executed per pixel or per sample.
- **Number of Multisamples** (3DSTATE_MULTISAMPLE and SURFACE_STATE): indicates the number of samples per pixel contained on the surface. This field in 3DSTATE_MULTISAMPLE must match the corresponding field in SURFACE_STATE for each render target. The depth, hierarchical depth, and stencil buffers inherit this field from 3DSTATE_MULTISAMPLE.
- **Pixel Location** (3DSTATE_MULTISAMPLE): indicates the subpixel location where values specified as "pixel" are sampled. This is either the upper left corner or the center.
- **Sample Offsets** (3DSTATE_MULTISAMPLE): for each of the four samples, specifies the subpixel location of each sample.

APIs define a "Multisample" render state boolean which controls how objects are rasterized (sample level vs. pixel level). The binding of MSRTs also affects the rasterization process. The various permutations of multisample operation are listed below, along with the HW state settings required.

		HW Mode		
MSRT	**Sampling Pattern**	**Multisample Enable**	**PerSample PS?**	
1X	n/a	Disabled	-	Legacy Non-MSAA Mode
		Enabled	-	1X Multisampling Mode
4X	UL or Center	Disabled	No	MSRT Only, PerPixel PS
			Yes	MSRT Only, PerSample PS
		Enabled	No	Multibuffering MSAA, PerPixel PS
			Yes	Multibuffering MSAA, PerSample PS
	Pattern	Disabled	No	MSRT Only, PerPixel PS
			Yes	n/a
			No	Mixed Mode, PerPixel PS
			Yes	Mixed Mode, PerSample PS
		Enabled	No	Pattern MSAA, PerPixel PS
			Yes	Pattern MSAA, PerSample PS

HW State			HW Mode
Num Samples	**MSRAST MODE**	**DISP MODE**	
1X	OFF_PIXEL	PERSAMPLE	**Legacy Non-MSAA Mode** 1X rasterization, using Pixel Location Legacy lines/aa-line rasterization 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	ON_PIXEL	PERSAMPLE	**1X Multisampling Mode** 1X rasterization, using Pixel Location MSAA lines only, using Pixel Location 1X PS, sample at Pixel Location 1X output merge, eval Depth at Pixel Location
	-	PERPIXEL	Invalid
	ON_PATTERN	-	Invalid
	OFF_PATTERN	-	Invalid
4X	OFF_PIXEL	PERPIXEL	**MSRT Only, PerPixel PS** 1X rasterization, using Pixel Location Legacy lines/aa-line rasterization

			1X PS, sample at Pixel Location 4X output merge, eval Depth at Pixel Location
		PERSAMPLE	**MSRT Only, PerSample PS** 1X rasterization, using Pixel Location Legacy lines/aa-line rasterization 4X PS, all samples at Pixel Location 4X output merge, eval Depth at Pixel Location
	ON_PIXEL	PERPIXEL	**Multibuffering MSAA, PerPixel PS** 1X rasterization, using Pixel Location MSAA lines only 1X PS, sample at Pixel Location 4X output merge, eval Depth at Pixel Location
		PERSAMPLE	**Multibuffering MSAA, PerSample PS** 1X rasterization, using Pixel Location MSAA lines only 4X PS, all samples at Pixel Location 4X output merge, eval Depth at Pixel Location
	OFF_PATTERN	PERPIXEL	**Mixed Mode, PerPixel PS** Lines: Legacy lines/aa-line rasterization using Pixel Location Non-Lines: 4X rasterization, using Sample Offsets 1X PS, sample at Pixel Location 4X output merge, eval depth at Sample Offsets
		PERSAMPLE	**Mixed Mode, PerSample PS** Lines: Legacy lines/aa-line rasterization using Pixel Location Non-Lines: 4X rasterization, using Sample Offsets 4X PS, sample at Pixel Location or Sample Offsets 4X output merge, eval depth at Sample Offsets
	ON_PATTERN	PERPIXEL	**Pattern MSAA, PerPixel PS** 4X rasterization, using Sample Offsets MSAA lines only 1X PS, sample at Pixel Location 4X output merge, eval depth at Sample Offsets
		PERSAMPLE	**Pattern MSAA, PerSample PS** 4X rasterization, using Sample Offsets MSAA lines only 4X PS, sample at Pixel Location or Sample Offsets 4X output merge, eval depth at Sample Offsets

8.4 Early Depth/Stencil Processing

The Windower/IZ unit provides the Early Depth Test function, a major performance-optimization feature where an attempt is made to remove pixels that fail the Depth and Stencil Tests prior to pixel shading. This requires the WM unit to perform the interpolation of pixel (“source”) depth values, read the current (“destination”) depth values from the cached depth buffer, and perform the Depth and Stencil Tests. As the WM unit has per-pixel source and destination Z values, these values are passed in the PS thread payload, if required.

8.4.1 Depth Coefficient Read-Back [Pre-DevSNB]

The WM unit must read back the depth coefficients from the URB entry containing the output of the Setup kernel. The value to program into the **Depth Coefficient URB Read Offset** state variable (in WM_STATE) should be computed as follows:

Depth Coefficient URB Read Offset = element_entry * 2 + 1

where element_entry is the location of the position element in the vertex data (ignoring the vertex header). For most applications, the position element will be in element 0.

8.4.2 Depth Offset

There are occasions where the Z position of some objects need to be slightly offset in order to reduce artifacts due to coplanar or near-coplanar primitives. A typical example is drawing the edges of triangles as wireframes – the lines need to be drawn slightly closer to the viewer to ensure they will not be occluded by the underlying polygon. Another example is drawing objects on a wall – without a bias on the z positions, they might be fully or partially occluded by the wall.

The device supports *global* depth offset, applied only to triangles, that bases the offset on the object's z slope. Note that there is no clamping applied at this stage after the Z position is offset – clamping to [0,1] can be performed later after the Z position is interpolated to the pixel. This is preferable to clamping prior to interpolation, as the clamping would change the Z slope of the entire object.

The Global Depth Offset function is controlled by the **Global Depth Offset Enable** state variable in WM_STATE. Global Depth Offset is only applied to 3DOBJ_TRIANGLE objects.

When Global Depth Offset Enable is ENABLED, the pipeline will compute:
MaxDepthSlope = max(abs(dZ/dX),abs(dz/dy)) // approximation of max depth slope for polygon
When UNORM Depth Buffer is at Output Merger (or no Depth Buffer):

```
Bias = GlobalDepthOffsetConstant * r + GlobalDepthOffsetScale *
       MaxDepthSlope
```

Where r is the minimum representable value > 0 in the depth buffer format, converted to float32. (note: If state bit **Legacy Global Depth Bias Enable** is set, the r term will be forced to 1.0)

When Floating Point Depth Buffer at Output Merger:

```
Bias = GlobalDepthOffsetConstant * 2^(exponent(max z in primitive) -
       r) + GlobalDepthOffsetScale * MaxDepthSlope
```

Where r is the # of mantissa bits in the floating point representation (excluding the hidden bit), e.g. 23 for float32. (note: If state bit Legacy Global Depth Bias Enable is set, no scaling is applied to the GobalDepthOffsetConstant).

Adding Bias to z:

```
if (GlobalDepthOffsetClamp > 0)
      Bias = min(DepthBiasClamp, Bias)
else if(GlobalDepthOffsetClamp < 0)
      Bias = max(DepthBiasClamp, Bias)
// else if GlobalDepthOffsetClamp == 0, no clamping occurs
z = z + Bias
```

Biasing is constant for a given primitive. The biasing formulas are performed with float32 arithmetic. Global Depth Bias is not applied to any point or line primitives

8.4.3 Early Depth Test / Stencil Test/Write

When **Early Depth Test Enable** is ENABLED, the WM unit will attempt to discard depth-occluded pixels during scan conversion (before processing them in the Pixel Shader). Pixels are only discarded when the WM unit can ensure that they would have no impact to the ColorBuffer or DepthBuffer. This function is therefore only a performance feature. If some pixels within a subspan are discarded, only the pixel mask is affected indicating that the discarded pixels are not active. If all pixels within a subspan are discarded, that subspan will not even be dispatched.

8.4.3.1 Software-Provided PS Kernel Info

In order for the WM unit to properly perform Early Depth Test and supply the proper information in the PS thread payload (and even determine if a PS thread needs to be dispatched), it requires information regarding the PS kernel operation. This information is provided by a number of state bits in WM_STATE, as summarized in the following table.

State Bit	Description
Pixel Shader Kill Pixel	This must be set when there is a chance that valid pixels passed to a PS thread may be discarded. This includes the discard of pixels by the PS thread resulting from a “killpixel” or “alphatest” function or as dictated by the results of the sampling of a “chroma-keyed” texture. The WM unit needs this information to prevent early depth/stencil writes for pixels which might be killed by the PS thread, etc. See WM_STATE/3DSTATE_WM for more information.
Pixel Shader Computed Depth	This must be set when the PS thread computes the “source” depth value (i.e., from the API POV, writes to the “oDepth” output). In this case the WM unit can’t make any decisions based on the WM-interpolated depth value. See WM_STATE/3DSTATE_WM for more information.
Pixel Shader Uses Source Depth	Must be set if the PS thread requires the WM-interpolated source depth value. This will force the source depth to be passed in the thread payload where otherwise the WM unit would not have seen it as required. See WM_STATE/3DSTATE_WM for more information.

8.4.3.2 Early Depth Test Cases [Pre-DevSNB]

There are cases, however, where the early depth test cannot be completed without information that will be generated by the pixel shader thread. The cases of depth test are divided as follows:

- **Computed depth (C)** is active whenever depth test *and* depth write (if enabled) needs to be performed post pixel shader. Most commonly, this includes cases where the pixel shader program writes to oDepth, emitting a "source depth" value which overrides the interpolated depth value. For these cases, the depth test cannot be done early, as the source depth is not available. Stencil test could be done early, but because the depth test cannot be done, the stencil write cannot be completed. Therefore, there is no advantage to doing the stencil test early. This includes cases where the pixel shader can kill pixels, including via sampler chroma key, as well as cases where the alpha test function is enabled, which kills pixels based on a programmable alpha test. In this case, even if the depth test fails, the pixel cannot be killed if a stencil write is indicated. Whether or not the stencil write happens depends on whether or not the pixel is killed later.
- **Non-promoted depth (N)** is active whenever the depth test can be done early but it cannot determine whether or not to write source depth to the depth buffer, therefore the depth write must be performed post pixel shader. This includes cases where the pixel shader can kill pixels, including via sampler chroma key, as well as cases where the alpha test function is enabled, which kills pixels based on a programmable alpha test. In this case, even if the depth test fails, the pixel cannot be killed if a stencil write is indicated. Whether or not the stencil write happens depends on whether or not the pixel is killed later. In these cases if stencil test fails and stencil writes are off, the pixels can also be killed early. If stencil writes are enabled, the pixels must be treated as Computed depth (described above).
- **Promoted depth (P)** is active whenever both the depth test and the conditional depth write can be preformed before the pixel shader is executed. In this case, the entire depth/stencil operation is completed pre pixel shader. This includes all cases where depth test is disabled and stencil test is either disabled or no write is indicated.
-

The following logic equations define the test signals used by the following table. Also defined are the read enables that control reading of the depth/stencil buffer. Note that the **depth_test_en, stencil_test_en** and **depth_write_en** signals are qualified with a non-null depth buffer surface type (as specified in 3DSTATE_DEPTH_BUFFER).

```
depth_test_en = state_depth_test_en && !depth_surface_type_null

depth_read_en = depth_test_en

stencil_test_en = state_stencil_test_en && !depth_surface_type_null

stencil_read_en = state_stencil_test_en

depth_buffer_read_en = depth_read_en || stencil_read_en

depth_buffer_write_enable = state_depth_buffer_write_enable &&
      !depth_surface_type_null
```

```
stencil_buffer_write_en =  state_stencil_buffer_write_enable &&
      stencil_test_en
```

The following table indicates how the hardware determines which of the three above modes is active based on the above inputs. Note that cases where the stencil buffer write enable is active without the stencil test enable are not possible based on the equation above.

If statistics are enabled, windower (and Jitter) will need to detect when alpha test or killpix is on and the IZ Table output is Promoted (early depth test enabled or disabled). If these conditions are met, windower must force a write only depth allocation. In addition the windower / Jitter will force the result to be NONPROMOTED and force **Source Depth to Render Target** signal to be set. If **Pixel Shader Computed Depth** is not set, windower / Jitter must force the **Source Depth Present To EU** signal to be set and include the source depth data in the dispatch payload.

If statistics are enabled, depth-test/write disabled, stencil test/write disabled and kill-pix/alpha-test enabled. Hiz needs to be disabled.

[DevILK]: If the above condition occurs with stencil test enabled then in addition to setting Source Depth to Render Target, jitter and IZ also need to set destination stencil present. Jitter should expect stencil data into the pixel shader for these cases.

Behavior for Early Depth Test enabled:

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
0	0	0	0	0	0	P	0	0	0	0
0	0	0	0	0	1	P	0	0	0	0
0	0	0	0	1	0	P	0	0(1)[1]	0	0
0	0	0	0	1	1	P	0	0(1)[1]	0	0
0	0	0	1	0	0	P	0	0	0	0
0	0	0	1	0	1	N	1	1	0	0
0	0	0	1	1	0	N	0	1	0	0
0	0	0	1	1	1	N	0	1	0	0
0	0	1	0	0	0	P	0	0	0	0
0	0	1	0	0	1	P	0	0	0	0
0	0	1	0	1	0	C	0	1	1	0
0	0	1	0	1	1	C	0	1	1	0
0	0	1	1	0	0	P	0	0	0	0
0	0	1	1	0	1	N	1	1	0	0
0	0	1	1	1	0	C	0	1	1	0
0	0	1	1	1	1	C	0	1	1	0
1	0	0	0	0	0	P	0	0	0	0
1	0	0	0	0	1	P	0	0	0	0
1	0	0	0	1	0	P	0	0(1)[1]	0	0
1	0	0	0	1	1	P	0	0(1)[1]	0	0
1	0	0	1	0	0	P	0	0	0	0
1	0	0	1	0	1	N	1	1	0	1
1	0	0	1	1	0	N	0	1	0	1
1	0	0	1	1	1	N	0	1	0	1
1	0	1	0	0	0	P	0	0	0	0
1	0	1	0	0	1	P	0	0	0	0
1	0	1	0	1	0	C	0	1	1	1
1	0	1	0	1	1	C	0	1	1	1
1	0	1	1	0	0	P	0	0	0	0
1	0	1	1	0	1	N	1	1	0	1

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
1	0	1	1	1	0	C	0	1	1	1
1	0	1	1	1	1	C	0	1	1	1
1	1	0	0	0	0	P	0	0	0	0
1	1	0	0	0	1	C	0	0	0	1
1	1	0	0	1	0	P	0	0(1)[1]	0	0
1	1	0	0	1	1	C	0	1	0	1
1	1	0	1	0	0	P	0	0	0	0
1	1	0	1	0	1	C	1	1	0	1
1	1	0	1	1	0	C(N)[1]	0	1	0	1
1	1	0	1	1	1	C	0	1	0	1
1	1	1	0	0	0	P	0	0	0	0
1	1	1	0	0	1	C	1	1	1	1
1	1	1	0	1	0	C	0	1	1	1
1	1	1	0	1	1	C	0	1	1	1
1	1	1	1	0	0	P	0	0	0	0
1	1	1	1	0	1	C	1	1	1	1
1	1	1	1	1	0	C	0	1	1	1
1	1	1	1	1	1	C	0	1	1	1

NOTES:

1. The value in parenthesis is for [DevBW-A] only.

Behavior for Early Depth Test disabled:

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
0	0	0	0	0	0	P	0	0	0	0
0	0	0	0	0	1	P	0	0	0	0
0	0	0	0	1	0	C	0	1	0	0
0	0	0	0	1	1	C	0	1	0	0
0	0	0	1	0	0	C	1	1	0	0
0	0	0	1	0	1	C	1	1	0	0
0	0	0	1	1	0	C	0	1	0	0
0	0	0	1	1	1	C	0	1	0	0
0	0	1	0	0	0	C	1	1	1	0
0	0	1	0	0	1	C	1	1	1	0
0	0	1	0	1	0	C	0	1	1	0
0	0	1	0	1	1	C	0	1	1	0
0	0	1	1	0	0	C	1	1	1	0
0	0	1	1	0	1	C	1	1	1	0
0	0	1	1	1	0	C	0	1	1	0
0	0	1	1	1	1	C	0	1	1	0
1	0	0	0	0	0	C	0	0	0	1
1	0	0	0	0(1)[1]	1(0)[1]	C	0	0	0	1
1	0	0	0	1(0)[1]	0(1)[1]	C	0	1	0	1
1	0	0	0	1	1	C	0	1	0	1
1	0	0	1	0	0	C	1	1	0	1
1	0	0	1	0	1	C	1	1	0	1
1	0	0	1	1	0	C	0	1	0	1
1	0	0	1	1	1	C	0	1	0	1
1	0	1	0	0	0	C	1	1	1	1
1	0	1	0	0	1	C	1	1	1	1
1	0	1	0	1	0	C	0	1	1	1
1	0	1	0	1	1	C	0	1	1	1
1	0	1	1	0	0	C	1	1	1	1
1	0	1	1	0	1	C	1	1	1	1

Stencil Test Enable	Stencil Buffer Write Enable	Depth Test Enable	Depth Buffer Write Enable	Pixel Shader Computed Depth	Pixel Shader Kill Pixel OR Alpha Test Enable	Early Depth Mode	Source Depth Present (to EU)	Source Depth to Render Target	Destination Depth Present (to EU and RT)	Destination Stencil Present (to EU and RT)
1	0	1	1	1	0	C	0	1	1	1
1	0	1	1	1	1	C	0	1	1	1
1	1	0	0	0	0	C	0	0	0	1
1	1	0	0	0	1	C	0	0	0	1
1	1	0	0	1	0	C	0	1	0	1
1	1	0	0	1	1	C	0	1	0	1
1	1	0	1	0	0	C	1	1	0	1
1	1	0	1	0	1	C	1	1	0	1
1	1	0	1	1	0	C	0	1	0	1
1	1	0	1	1	1	C	0	1	0	1
1	1	1	0	0	0	C	1	1	1	1
1	1	1	0	0	1	C	1	1	1	1
1	1	1	0	1	0	C	0	1	1	1
1	1	1	0	1	1	C	0	1	1	1
1	1	1	1	0	0	C	1	1	1	1
1	1	1	1	0	1	C	1	1	1	1
1	1	1	1	1	0	C	0	1	1	1
1	1	1	1	1	1	C	0	1	1	1

NOTE:

1. The value in parenthesis is for [DevBW-A] only.

Note: source depth present (to EU) will also be set in cases in which the pixel shader uses source depth (vPos.z) regardless of any other condition.

The specific actions for each case are as follows.

Early Depth Mode	Pixel	Depth	Stencil	Depth sent to Pixel Shader	Depth sent to Render Target	Stencil sent to PS/RT
Computed Depth	conditionally killed based on depth/stencil test post-shader	tested and written post-shader	tested and written post-shader	source depth for vPos.z if used dest depth passed through	source depth from oDepth dest depth passed through	dest stencil passed through if stencil test enabled
Non-promoted	pixel killed pre-shader if depth test fails and no stencil write indicated	test pre-and post-shader, written post-shader	tested and written post-shader	source depth for vPos.z if used source depth always	source depth from vPos.z	dest stencil passed through if stencil test enabled
Promoted	pixel killed pre-shader on fail	tested and written pre-shader	tested and written pre-shader	source depth for vPos.z if used	none	none

The following psuedocode describes the logic that determines whether color, depth, and stencil are written depending on results of alpha, depth, and stencil tests.

```
alpha_test_pass = TRUE
depth_test_pass = TRUE
stencil_test_pass = TRUE

if (alpha_test_enable) alpha_test_pass = TestAlpha();
if (depth_test_enable) depth_test_pass = TestDepth();
if (stencil_test_enable) stencil_test_pass = TestStencil();

stencil_update = (new_stencil_value != dst_stencil_value) &&
      (stencil_test_enable == TRUE)

pass_color_depth = (alpha_test_pass == TRUE) && (depth_test_pass == TRUE)
      && (stencil_test_pass == TRUE) && (pixel_enabled == TRUE)
pass_stencil = (alpha_test_pass == TRUE) && (stencil_update == TRUE) &&
      (pixel_enabled == TRUE)

pixel_color_write = pass_color_depth && (color_component_write_disables !=
      0xf)
pixel_depth_write = pass_color_depth && (depth_buffer_write_enable ==
      TRUE)
pixel_stencil_write = pass_stencil && (stencil_buffer_write_enable ==
      TRUE)
```

8.4.4 Hierarchical Depth Buffer [DevILK+]

A hierarchical depth buffer is supported beginning with [DevILK] to reduce memory traffic due to depth buffer accesses. This buffer is supported only in Tile Y memory.

The **Surface Type**, **Height**, **Width**, **Depth**, **Minimum Array Element**, **Render Target View Extent**, and **Depth Coordinate Offset X/Y** of the hierarchical depth buffer are inherited from the depth buffer. The height and width of the hierarchical depth buffer that must be allocated are computed by the following formulas, where HZ is the hierarchical depth buffer and Z is the depth buffer. The Z_Height, Z_Width, and Z_Depth values given in these formulas are those present in 3DSTATE_DEPTH_BUFFER incremented by one.

Surface Type	HZ_Width (bytes)	HZ_Height (rows)
SURFTYPE_1D	`ceiling(Z_Width / 16) * 16`	`2 * Z_Depth`
SURFTYPE_2D	`ceiling(Z_Width / 16) * 16`	`ceiling(Z_Height / 8) * 4 * Z_Depth`
SURFTYPE_3D	`ceiling(Z_Width / 16) * 16`	`ceiling(Z_Height / 8) * 4 * Z_Depth`
SURFTYPE_CUBE	`ceiling(Z_Width / 16) * 16`	`ceiling(Z_Height / 8) * 24 * Z_Depth`

where, QPitch is computed using vertical alignment j=8, please refer to the GPU overview volume for QPitch definition.

The format of the data in the heirarchical depth buffer is not documented here, as this surface needs only to be allocated by software. Hardware will read and write this surface during operation and its contents are discarded once the last primitive is rendered that uses the hierarchical depth buffer.

The hierarchical depth buffer can be enabled whenever a depth buffer is defined, with its effect being invisible other than generally higher performance. The only exception is the hierarchical depth buffer must be disabled when using software tiled rendering.

If Hierarchical Depth Buffer is enabled, then the flush needs to be sent after clear-prim. [ILK+:]

If HiZ is enabled, you must initialize the clear value by either

a. Perform a depth clear pass to initialize the clear value.

b. Send a 3dstate_clear_params packet with valid = 1

Without one of these events, context switching will fail, as it will try to save off a clear value even though no valid clear value has been set. When context restore happens, HW will restore an uninitialized clear value.

8.4.4.1 Depth Buffer Clear

With the hierarchical depth buffer enabled, performance is generally improved by using the special clear mechanism described here to clear the hierarchical depth buffer and the depth buffer. This is enabled though the **Depth Buffer Clear** field in WM_STATE or 3DSTATE_WM. This bit can be used to clear the depth buffer in the following situations:

- Complete depth buffer clear
- Partial depth buffer clear with the clear value the same as the one used on the previous clear
- Partial depth buffer clear with the clear value different than the one used on the previous clear can use this mechanism if a depth buffer resolve is performed first.

The following is required when performing a depth buffer clear with this field:

- If other rendering operations have preceded this clear, a PIPE_CONTROL with write cache flush enabled and Z-inhibit disabled must be issued before the rectangle primitive used for the depth buffer clear operation.
- The fields in 3DSTATE_CLEAR_PARAMS are set to indicate the source of the clear value and (if source is in this command) the clear value itself.
- A rectangle primitive representing the clear area is delivered.
- **Depth Test Enable** must be disabled and **Depth Buffer Write Enable** must be enabled.
- **Errata: [DevILK]: For D16_UNORM depth buffer format, fast clear optimization and resolve operations are not supported.**
- Stencil buffer clear can be performed at the same time by enabling Stencil Buffer Write Enable. Stencil Test Enable must be disabled and Stencil Pass Depth Pass Op set to REPLACE.
- **Pixel Shader Dispatch, Alpha Test, Pixel Shader Kill Pixel** and **Pixel Shader Computed Depth** must all be disabled.

[DevILK]: Several cases exist where **Depth Buffer Clear** with Fast Clear Optimization enabled (Cache Mode Register offset 0x**2120, bit 2**) cannot be enabled:

- If the depth buffer format is D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT.
- If the depth buffer format is D16_UNORM (**[DevILK]** only).
- If the separate stencil buffer is disabled.
- If the depth buffer format is D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT.
- If stencil test is enabled but the separate stencil buffer is disabled.
- [DevILK]: When depth buffer format is D16_UNORM and the width of the map (LOD0) is not multiple of 16, fast clear optimization must be disabled.

8.4.4.2 Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the depth buffer may contain incorrect results after rendering is complete. If the depth buffer is retained and used for another purpose (i.e as input to the sampling engine as a shadow map), it must first be "resolved". This is done by setting the **Depth Buffer Resolve Enable** field in WM_STATE or 3DSTATE_WM and rendering a full render target sized rectangle. Once this is complete, the depth buffer will contain the same contents as it would have had the rendering been performed with the hierarchical depth buffer disabled. In a typical usage model, depth buffer needs to be resolved after rendering on it and before using a depth buffer as a source for any consecutive operation. Depth buffer can be used as a

source in three different cases: using it as a texture for the nest rendering sequence, honoring a lock on the depth buffer to the host OR using the depth buffer as a blit source.

The following is required when performing a depth buffer resolve:

- A rectangle primitive of the same size as the previous depth buffer clear operation must be delivered, and depth buffer state cannot have changed since the previous depth buffer clear operation.
- **Depth Test Enable** must be enabled with the **Depth Test Function** set to NEVER. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch, Alpha Test, Pixel Shader Kill Pixel** and **Pixel Shader Computed Depth** must all be disabled.

8.4.4.3 Hierarchical Depth Buffer Resolve

If the hierarchical depth buffer is enabled, the hierarchical depth buffer may contain incorrect results if the depth buffer is written to outside of the 3D rendering operation. If this occurs, the hierarchical depth buffer must be "resolved" to avoid incorrect device behavior. This is done by setting the **Hierarchical Depth Buffer Resolve Enable** field in WM_STATE or 3DSTATE_WM and rendering a full render target sized rectangle. Once this is complete, the hierarchical depth buffer will contain contents such that rendering will give the same results as it would have had the rendering been performed with the hierarchical depth buffer disabled.

The following is required when performing a hierarchical depth buffer resolve:

- A rectangle primitive covering the full render target must be delivered.
- **Depth Test Enable** must be disabled. **Depth Buffer Write Enable** must be enabled. **Stencil Test Enable** and **Stencil Buffer Write Enable** must be disabled.
- **Pixel Shader Dispatch, Alpha Test, Pixel Shader Kill Pixel** and **Pixel Shader Computed Depth** must all be disabled.

Errata: [DevILK]:

- If alpha-test or kill-pix enabled and Hierarchical Depth Buffer is enabled and depth test function is GREATER/GREATEREQ/LESS/LESSEQ, Hierarchical Depth Buffer must be disabled. After such rendering, in order to enable Hierarchical Depth Buffer for rendering without alpha-test or kill-pix, Hierarchical Depth Buffer Resolve pass is required.

- Hiz buffer should be disabled when the depth-buffer-format is 16bpp.

8.4.5 Separate Stencil Buffer

8.4.5.1 [Pre-DevGT]

A separate stencil buffer is supported beginning with **[DevILK],** which improves performance when using the hierarchical depth buffer with stencil test enabled. This buffer is supported only in Tile Y memory. If the separate stencil buffer is enabled, it always has the format S8_UINT. The **Surface Type**, **Height**, **Width**, and **Depth**, **Minimum Array Element**, **Render Target View Extent**, and **Depth Coordinate Offset X/Y** of the stencil buffer are inherited from the depth buffer.

The stencil depth buffer does not support the **LOD** field, it is assumed by hardware to be zero. A separate stencil depth buffer is required for each LOD used, and the corresponding buffer's state delivered to hardware each time a new depth buffer state with modified LOD is delivered.

The stencil channel in the depth buffer is still supported, however if this is used with the hierarchical depth buffer, performance will generally be lower than using the separate stencil buffer.

Errata:

- [**DevILK**]: Separate Stencil Buffer must be disabled if rendering with alpha-test or kill-pix is present in an application.
- **[DevILK]:** When Separate Stencil Buffer is enabled and the surface type is 2D array, each array element would require twice the height derived by the normal computation based on 2D array layout.
- **[DevILK]** When Separate Stencil Buffer is enabled, fast clear optimization must be disabled if stencil buffer is cleared with value 0.

8.4.6 Depth/Stencil Buffer State

8.4.6.1.1 [Pre-Dev GT]

3DSTATE_DEPTH_BUFFER			
Project:	All	**Length Bias:**	2
The depth buffer surface state is delivered as a non-pipelined state packet.			
DWord	**Bit**	**Description**	
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode	
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode	
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode	

3DSTATE_DEPTH_BUFFER

	23:16	**3D Command Sub Opcode** Default Value: 05h 3DSTATE_DEPTH_BUFFER Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 3h Excludes DWord (0,1) Format: =n Total Length - 2 Project: [Pre-DevCTG]
	7:0	**DWord Length** Default Value: 4h Excludes DWord (0,1) Format: =n Total Length - 2 Project: [DevCTG], DevILK]
1	31:29	**Surface Type** Project: All Format: U3 Enumerated Type This field defines the type of the surface.

Value Na	me	Description	Project
0h	SURFTYPE_1D	Defines a 1-dimensional map or array of maps	All
1h	SURFTYPE_2D	Defines a 2-dimensional map or array of maps	All
2h	SURFTYPE_3D	Defines a 3-dimensional (volumetric) map	All
3h	SURFTYPE_CUBE	Defines a cube map	All
4h-6h	Reserved		All
7h	SURFTYPE_NULL	Defines a null surface	All

Programming Notes
The **Surface Type** of the depth buffer must be the same as the **Surface Type** of the render target(s) (defined in SURFACE_STATE), unless either the depth buffer or render targets are SURFTYPE_NULL.

	28	**Reserved** Project: All Format: MBZ

3DSTATE_DEPTH_BUFFER

27

Tiled Surface

Project: All

Format: U1 enumerated type FormatDesc

Specifies if the surface is tiled.

Value Name	Description	Project
0h FALSE	Linear surface	All
1h TRUE	Tiled surface	All

Programming Notes	Project
Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory. [DevILK] : When Hierarchical Depth Buffer is enabled, this bit must be set.	All
The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.	All

Errata	Description	Project
BWT014	The Depth Buffer Must be Tiled, it cannot be linear. This field must be set to 1 on DevBW-A.	[DevBW

26

Tile Walk

Project: All

Format: U1 enumerated type FormatDesc

This field specifies the type of memory tiling (XMajor or YMajor) employed to tile this surface. The Depth Buffer, if tiled, must use Y-Major tiling. See *Memory Interface Functions* for details on memory tiling and restrictions.

This field is ignored when the surface is linear.

Value Name	Description	Project
0h Reserved		All
1h TILEWALK_YMAJOR	Y major tiled	All

3DSTATE_DEPTH_BUFFER		
	25	**Depth Buffer Coordinate Offset Disable** Project: [Pre-DevCTG] Format: Disable FormatDesc Disables the application (addition) of the “upper bits” of the Drawing Rectangle Origin to Depth Buffer coordinates. (This does not affect the application of the Drawing Rectangle Origin to the Color Buffer coordinates). This control is provided to better support “Front Buffer Rendering”. By disabling the Draw Rectangle adjustment of Depth Buffer coordinates, software can utilize a “window-sized” Depth Buffer while rendering to a window within the Color Buffer. Without this control, use of the Draw Rectangle adjustment would require the Depth Buffer to be dimensioned to match the Color Buffer (screen) vs. the target window.
	25	**Reserved** Project: [DevCTG+] Format: MBZ

Programming Notes	Project
The device still applies some small coordinate offset in order to provide the required alignment of color and depth memory/cache accesses. Software needs to consider this alignment when allocating depth buffers.	All
This bit must not be set when rendering to field-mode (interlaced) Color Buffers (i.e., when Surface State’s VerticalLineStride==1).	All
This bit can only be set when rendering to surfaces of type SURFTYPE_1D and SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped)	All

3DSTATE_DEPTH_BUFFER

	24:23	**Software Tiled Rendering Mode** Project: All Format: U2 enumerated type FormatDesc This field is intended to enable *software tiled rendering (STR)*. If certain restrictions are met, performance can be improved by reducing memory bandwidth to the render target and depth buffer. **Normal mode:** Rendering behaves normally. **STR1 mode:** Only pixels within a particular 64x32 block (aligned relative to the upper left corner of the render target) are rendered between pixel shader serializations. Generally the alignment is guaranteed via a scissor rectangle. A write to a given pixel in the render target must occur before a read from the same pixel. **STR2 mode:** The restrictions of STR1 mode applies, and in addition each pixel must be rendered with depth write enabled and depth test disabled before it can be rendered with depth test enabled. The depth buffer in memory is not updated, even on a render cache flush. Depth buffer data is contained only within the render cache during rendering.

Value Name	Description	Project
0h	NORMAL	Normal mode	All
1h	STR1	STR1 mode	[DevCTG+]
2h	Reserved		All
3h	STR2	STR2 mode	[DevCTG+]

Programming Notes	Project
[Pre-DevCTG]]: Only mode is supported	[Pre-DevCTG]
The render cache must be flushed when this field is modified from its previous state	All
For both STR modes, the depth buffer (if used) must be tiled Y with D16_UNORM format, and the render target surface must be tiled X or Y	All
For both STR modes, the only data port messages allowed that use the render cache are the Render Target UNORM Read and Write messages.	All
Performance considerations: Both STR modes eliminate all memory read traffic from the render target. The STR2 mode additionally eliminates all memory traffic to the depth buffer.	All
This field must be set to NORMAL if the **Render Cache Operational Flush Enable** bit is enabled in the Cache_Mode_0 register.	All

3DSTATE_DEPTH_BUFFER

22 Hierarchical Depth Buffer Enable

Project: [DevILK+]

Format: Enable FormatDesc

If enabled, indicates that a hierarchical depth buffer is defined.

Programming Notes
If this field is enabled, the **Software Tiled Rendering Mode** must be NORMAL.
This field must be disabled if **Early Depth Test Enable** is disabled.

21 Separate Stencil Buffer Enable

Project: [DevILK+]

Format: Enable FormatDesc

If enabled, indicates that a separate stencil buffer is defined.

Programming Notes
If this field is enabled, the **Surface Format** of the depth buffer must be D32_FLOAT or D24_UNORM_X8_UINT.
If this field is disabled, the **Surface Format** of the depth buffer cannot be D24_UNORM_X8_UINT.
If this field is enabled, the **Software Tiled Rendering Mode** must be NORMAL.
If this field is enabled, **Hierarchical Depth Buffer Enable** must also be enabled.

20:18 Surface Format

Project: All

Format: U3 enumerated type FormatDesc

Specifies the format of the depth buffer. See the **Separate Stencil Buffer Enable** and **Hierarchical Depth Buffer Enable** fields for restrictions on the use of some of these formats.

Value Name	Description	Project
0h	D32_FLOAT_S8X24_UINT	D32_FLOAT_S8X24_UINT	All
1h	D32_FLOAT	D32_FLOAT	All
2h	D24_UNORM_S8_UINT	D24_UNORM_S8_UINT	All
3h	D24_UNORM_X8_UINT	D24_UNORM_X8_UINT	DevILK
4h	Reserved	Reserved	All
5h	D16_UNORM	D16_UNORM	All
6h-7h	Reserved	Reserved	All

17 Reserved Project: All Format: MBZ

3DSTATE_DEPTH_BUFFER

	16:0	**Surface Pitch** Project: All Format: U17 pitch in (Bytes – 1) FormatDesc Range if linear: [63, 128K-1] corresponding to [64B, 128KB] also restricted to a multiple of 64B if tiled: [127, 128K-1] corresponding to [128B, 128KB] also restricted to a multiple of 128B This field specifies the pitch of the depth buffer in (#Bytes – 1). **Programming Notes** / **Project** If this surface is tiled, the pitch specified must be a multiple of the tile pitch, in the range [128B, 128KB]. / All If the surface is linear, the pitch can be any multiple of 64 bytes up to 128KB. / All
2	31:0	**Surface Base Address** Project: All Address: GraphicsAddress[31:0] Surface Type: Depth Buffer This field specifies the starting DWord address of the buffer in mapped Graphics Memory. **Programming Notes** The Depth Buffer can only be mapped to Main Memory (uncached). If the surface is tiled, the base address must conform to the Per-Surface Tiling Alignment If the buffer is linear, the surface must be 64-byte aligned.
3	31:19	**Height** Project: All Format: U13 FormatDesc Range SURFTYPE_1D: must be zero SURFTYPE_2D: height of surface – 1 (y/v dimension) [0,8191] SURFTYPE_3D: height of surface – 1 (y/v dimension) [0,2047] SURFTYPE_CUBE: height of surface – 1 (y/v dimension) [0,8191] This field specifies the height of the surface. If the surface is MIP-mapped, this field contains the height of the base MIP level. **Programming Notes** The **Height** of the depth buffer must be the same as the **Height** of the render target(s) (defined in SURFACE_STATE), unless **Surface Type** is SURFTYPE_1D or SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped).

3DSTATE_DEPTH_BUFFER

	18:6	**Width** Project: All Format: U13 FormatDesc Range SURFTYPE_1D: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_2D: width of surface – 1 (x/u dimension) [0,8191] SURFTYPE_3D: width of surface – 1 (x/u dimension) [0,2047] SURFTYPE_CUBE: width of surface – 1 (x/u dimension) [0,8191] This field specifies the width of the surface. If the surface is MIP-mapped, this field specifies the width of the base MIP level. The width is specified in units of pixels.

Programming Notes	Project
The Width specified by this field must be less than or equal to the surface pitch (specified in bytes via the **Surface Pitch** field).	All
For cube maps, **Width** must be set equal to **Height**.	All
The **Width** of the depth buffer must be the same as the **Width** of the render target(s) (defined in SURFACE_STATE), unless **Surface Type** is SURFTYPE_1D or SURFTYPE_2D with **Depth** = 0 (non-array) and **LOD** = 0 (non-mip mapped).	All

	5:2	**LOD** Project: All Format: U4 in LOD units FormatDesc Range [0, 13] This field defines the MIP level that is currently being rendered into.

Programming Notes	Project
The **LOD** of the depth buffer must be the same as the **LOD** of the render target(s) (defined in SURFACE_STATE).	All

3DSTATE_DEPTH_BUFFER

	1	**MIP Map Layout Mode** Project: All Format: U1 enumerated type FormatDesc **For 1D and 2D Surfaces:** This field specifies which MIP map layout mode is used, whether the map for LOD 1 is stored to the right of the LOD 0 map, or stored below it. See Memory Data Formats for details on the specifics of each layout mode. **For Other Surfaces:** This field is reserved : MBZ

Value Name	Description	Project
0h	MIPLAYOUT_BELOW	MIPLAYOUT_BELOW	All
1h	MIPLAYOUT_RIGHT	MIPLAYOUT_RIGHT	All

Programming Notes	Project
MIPLAYOUT_RIGHT is legal only for 2D non-array surfaces	All
The **MIP Map Layout Mode** of the depth buffer must be the same as the **MIP Map Layout Mode** of the render target(s) (defined in SURFACE_STATE).	[Pre-DevGT]

	0	**Reserved** Project: All Format: MBZ
4	31:21	**Depth** Project: All Format: U11 FormatDesc Range SURFTYPE_1D: number of array elements – 1 [0,511] SURFTYPE_2D: number of array elements – 1 [0,511] SURFTYPE_3D: depth of surface – 1 (r/z dimension) [0,2047] SURFTYPE_CUBE: must be zero This field specifies the total number of levels for a volume texture or the number of array elements allowed to be accessed starting at the **Minimum Array Element** for arrayed surfaces. If the volume texture is MIP-mapped, this field specifies the depth of the base MIP level.

Programming Notes
The **Depth** of the depth buffer must be the same as the **Depth** of the render target(s) (defined in SURFACE_STATE).

3DSTATE_DEPTH_BUFFER

	20:10	**Minimum Array Element** Project: All Format: U11 FormatDesc Range SURFTYPE_1D/2D: [0,511] SURFTYPE_3D: [0,2047] **For 1D and 2D Surfaces:** This field indicates the minimum array element that can be accessed as part of this surface. The delivered array index is added to this field before being used to address the surface. **For 3D Surfaces:** This field indicates the minimum 'R' coordinate on the LOD currently being rendered to. This field is added to the delivered array index before it is used to address the surface. **For Other Surfaces:** This field is ignored. **Programming Notes** / **Project** **[DevBW-A]:** this field must be zero. / [DevBW-A] The **Minimum Array Element** of the depth buffer must be the same as the **Minimum Array Element** of the render target(s) (defined in SURFACE_STATE). / [Pre-DevGT]
	9:1	**Render Target View Extent** Project: All Format: U9 FormatDesc Range SURFTYPE_1D/2D: same value as Depth field SURFTYPE_3D: [0,511] to indicate extent of [1,512] **For 3D Surfaces:** This field indicates the extent of the accessible 'R' coordinates minus 1 on the LOD currently being rendered to. **For 1D and 2D Surfaces:** This field must be set to the same value as the **Depth** field. **For Other Surfaces:** This field is ignored. **Programming Notes** / **Project** **[DevBW-A]:** this field must be zero / [DevBW-A] The **Render Target View Extent** of the depth buffer must be the same as the **Render Target View Extent** of the render target(s) (defined in SURFACE_STATE). / [Pre-DevGT]
	0	**Reserved** Project: All Format: MBZ

3DSTATE_DEPTH_BUFFER		
5	31:16	**Depth Coordinate Offset Y** Project: [DevCTG+] Format: S15 in Screen Space (pixels) FormatDesc (3 LSBs MBZ) Range [-8192,8191] (Bits 31:30 should be a sign extension) Specifies a signed pixel offset to be added to the RenderTarget Y coordinate in order to generate a DepthBuffer Y coordinate. (See Depth Coordinate in Windower). **Programming Notes** The 3 LSBs of both offsets must be zero to ensure correct alignment Software must ensure that the resulting (sum) coordinate value is non-negative. This field must be zero when rendering to field-mode (interlaced) Color Buffers (i.e., when Surface State's VerticalLineStride==1). This field can only be nonzero when rendering to surfaces of type SURFTYPE_1D and SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped) [DevILK]: This field must be zero when separate stencil buffer is enabled.
	15:0	**Depth Coordinate Offset X** Project: [DevCTG+] Format: S15 in Screen Space (pixels) FormatDesc (3 LSBs MBZ) Range [-8192,8191] (Bits 15:14 should be a sign extension) Specifies a signed pixel offset to be added to the RenderTarget X coordinate in order to generate a DepthBuffer X coordinate. (See Depth Coordinate in Windower). **Programming Notes** The 3 LSBs of both offsets must be zero to ensure correct alignment Software must ensure that the resulting (sum) coordinate value is non-negative. This field must be zero when rendering to field-mode (interlaced) Color Buffers (i.e., when Surface State's VerticalLineStride==1). This field can only be nonzero when rendering to surfaces of type SURFTYPE_1D and SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped) [DevILK]: This field must be zero when separate stencil buffer is enabled.

3DSTATE_DEPTH_BUFFER		
6	31:21	**Render Target View Extent** Project: All Format: U11 FormatDesc Range SURFTYPE_1D/2D: same value as Depth field. SURFTYPE_3D: [0, 2047} to indicate the extent of [1, 2048] **FOR 3D Surfaces**: This field indicates the extent of the accessible 'R' coordinates minus 1 on the LOD currently being rendered to. **For 1D and 2D Surfaces**: This field must be set to the same value as the **Depth** field. **For Other Surfaces**: This field is ignored.
	20:0	**Reserved** Project: All Format: MBZ

8.4.6.2 3DST ATE_STENCIL_BUFFER

8.4.6.2.1 3DSTATE_STENCIL_BUFFER [DevILK]

3DSTATE_STENCIL_BUFFER			
Project:	[DevILK]]	**Length Bias:**	2
This command sets the surface state of the separate stencil buffer, delivered as a non-pipelined state command..			

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 0Eh 3DSTATE_STENCIL_BUFFER Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:29	**Reserved** Project: All Format: MBZ
	28:25	**Reserved** Project: [DevILK] Format: MBZ
	24:17	**Reserved** Project: All Format: MBZ
	16:0	**Surface Pitch** Project: All Format: U17 pitch in (Bytes – 1) FormatDesc Range [127, 128K-1] corresponding to [128B, 128KB] also restricted to a multiple of 128B This field specifies the pitch of the stencil buffer in (#Bytes – 1).

Programming Notes	Project
Since this surface is tiled, the pitch specified must be a multiple of the tile pitch, in the range [128B, 128KB].	All
The pitch must be set to 2x the value computed based on width, as the stencil buffer is stored with two rows interleaved. Refer to “Memory Data Formats” chapter for details on the separate stencil buffer storage format in memory.	[HVN/]

3DSTATE_STENCIL_BUFFER		
2	31:0	**Surface Base Address** Project: All Address: GraphicsAddress[31:0] Surface Type: Stencil Buffer This field specifies the starting DWord address of the buffer in mapped Graphics Memory. **Programming Notes** The Stencil Buffer can only be mapped to Main Memory (uncached). Since this surface is tiled, the base address must conform to the Per-Surface Tiling Alignment Rules as documented in TBD.

8.4.6.3 3DST ATE_HIER_DEPTH_BUFFER

8.4.6.3.1 3DST ATE_HIER_DEPTH_BUFFER [DevILK],

3DSTATE_HIER_DEPTH_BUFFER						
Project:	DevILK			**Length Bias:**	2	
This command sets the surface state of the hierarchical depth buffer, delivered as a non-pipelined state command.						
DWord	**Bit**	**Description**				
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode				
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode				
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode				
	23:16	**3D Command Sub Opcode** Default Value: 0Fh 3DSTATE_HIER_DEPTH_BUFFER Format: OpCode				
	15:8	**Reserved** Project: All Format: MBZ				
	7:0	**DWord Length** Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All				
1	31:29	**Reserved** Project: All Format: MBZ				
	28:25	**Reserved** Project: [DevILK] Format: MBZ				
	24:17	**Reserved** Project: All Format: MBZ				

3DSTATE_HIER_DEPTH_BUFFER		
	16:0	**Surface Pitch** Project: All Format: U17 pitch in (Bytes – 1) FormatDesc Range [127, 128K-1] corresponding to [128B, 128KB] also restricted to a multiple of 128B This field specifies the pitch of the hierarchical depth buffer in (#Bytes – 1). **Programming Notes** (Project: All) Since this surface is tiled, the pitch specified must be a multiple of the tile pitch, in the range [128B, 128KB].
2	31:0	**Surface Base Address** Project: All Address: GraphicsAddress[31:0] Surface Type: Hierarchical Depth Buffer This field specifies the starting DWord address of the buffer in mapped Graphics Memory. **Programming Notes** The Hierarchical Depth Buffer can only be mapped to Main Memory (uncached). Since this surface is tiled, the base address must conform to the Per-Surface Tiling Alignment Rules as documented in TBD.

8.4.6.4 3DST ATE_CLEAR_PARAMS

8.4.6.4.1 3DSTATE_CLEAR_PARAMS [DevILK]

3DSTATE_CLEAR_PARAMS packet must follow the DEPTH_BUFFER_STATE packet when HiZ is enabled and the DEPTH_BUFFER_STATE changes.

If HiZ is enabled, you must initialize the clear value by either

a. Perform a depth clear pass to initialize the clear value.

b. Send a 3dstate_clear_params packet with valid = 1

Without one of these events, context switching will fail, as it will try to save off a clear value even though no valid clear value has been set. When context restore happens, HW will restore an uninitialized clear value.

3DSTATE_CLEAR_PARAMS			
Project:	[DevILK]	**Length Bias:**	2
This command defines the depth and stencil clear values, delivered as a non-pipelioned state.			

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 10h 3DSTATE_CLEAR_PARAMS Format: OpCode
	14:8	**Reserved** Project: All Format: MBZ

3DSTATE_CLEAR_PARAMS		
1	31:0	**Depth Clear Value** Project: All Format: IEEE_Float for **Surface Format** of depth buffer: D32_FLOAT_S8X24_UINT: IEEE_Float D32_FLOAT: IEEE_Float D24_UNORM_S8_UINT: U24 UNORM in bits [23:0] D24_UNORM_X8_UINT: U24 UNORM in bits [23:0] D16_UNORM: U16 UNORM in bits [15:0] This field defines the clear value that will be applied to the depth buffer if the **Depth Buffer Clear** field is enabled. It is valid only if **Depth Buffer Clear Value Valid** is set.

d2dms and *sample2dms* messages.

8.5 Pixel Shader Thread Generation

After a group of object pixels have been rasterized, the Pixel Shader function is invoked to further compute pixel color/depth information and cause results to be written to rendertargets and/or depth buffers. For each pixel, the Pixel Shader calculates the values of the various vertex attributes that are to be interpolated across the object using the interpolation coefficients. It then executes an API-supplied Pixel Shader Program. Instructions in this program permit the accessing of texture map data, where Texture Samplers are employed to sample and filter texture maps (see the *Shared Functions* chapter). Arithmetic operations can be performed on the texture data, input pixel information and Pixel Shader Constants in order to compute the resultant pixel color/depth. The Pixel Shader program also allows the pixel to be discarded from further processing. For pixels that are not discarded, the pixel shader must send messages to update one or more render targets with the pixel results.

8.5.1 Pixel Grouping (Dispatch Size) Control

The WM unit can pass a grouping of 2 subspans (8 pixels), 4 subspans (16 pixels) or 8 subspans (32 pixels) to a Pixel Shader thread. Software should take into account the following considerations when determining which groupings to support/enable during operation. This determination involves a tradeoff of these likely conflicting issues. Note that the size of the dispatch has significant impact on the kernel program (it is certainly not transparent to the kernel). Also note that there is no implied spatial relationship between the subspans passed to a PS thread, other than the fact that they come from the same object.

1. **Thread Efficiency**: In general, there is some amount of overhead involved with PS thread dispatch, and if this can be amortized over a larger number of pixels, efficiency will likely increase. This is especially true for very short PS kernels, as may be used for desktop composition, etc.

2. **GRF Consumption**: Processing more pixels per thread will require a larger thread payload and likely more temporary register usage, both of which translate into a requirement for a larger GRF register allocation for the threads. If this increased GRF usage could lead to increased use of scratch space (for spill/fill, etc.) and possibly less efficient use of the EUs (as it would be less likely to find an EU with enough free physical GRF registers to service the thread).

3. **Object Size**: If the number of very small objects (e.g., covering 2 subspans or fewer) is expected to comprise a significant portion of the workload, supporting the 8-pixel dispatch mode may be advantageous. Otherwise there could be a large number of 16-pixel dispatches with only 1 or 2 valid subspans, resulting in low efficiency for those threads.

4. **Intangibles**: Kernel footprint & Instruction Cache impact; Complexity; ….

The groupings of subspans that the WM unit is allowed to include in a PS thread payload is controlled by the **32,16,8 Pixel Dispatch Enable** state variables programmed in WM_STATE. Using these state variables, the WM unit will attempt to dispatch the largest allowed grouping of subspans. The following table lists the possible combinations of these state variables.

Note: in the table below, the Valid column indicates which products that combination is supported on. Combinations of dispatch enables not listed in the table are not available on any product.

A: Valid on all products

B: Valid only on **[DevCTG+]**.

C: Valid only on **[DevCTG]**, and **[**DevILK**]**

E: Valid on all products

For **[Pre-DevILK]**, there is only one kernel start pointer (KSP) specified in WM_STATE, with other kernels being entered via an offset from the single KSP as follows:

KSP[0] = KSP

KSP[1] = KSP+1

KSP[2] = KSP+2

KSP[3] = KSP+3

For **[DevILK+]**, each of the four KSP values is separately specified. In addition, each kernel has a separately-specified GRF register count, whereas on **[Pre-DevILK]**, all kernels share the same GRF register count field, with the one with the maximum register count required applying to all.

Table 8-1. Variable Pixel Dispatch

Contiguous 64 Pixel Dispatch Enable	Contiguous 32 Pixel Dispatch Enable	32 Pixel Dispatch Enable	16 Pixel Dispatch Enable	8 Pixel Dispatch Enable	Valid	IP for n-pixel Contiguous Dispatch		IP for n-pixel Dispatch (KSP offsets are in 128-bit instruction units)		
						n=64 n	=32 n	=32	n=16	n=8
0	0	0	0	1	A					KSP[0]
0	0	0	1	0	F				KSP[0]	
0	0	0	1	1	D				KSP[2]	KSP[0]
0	0	1	0	0	B			KSP[0]		
0	0	1	1	0	E			KSP[1]	KSP[2]	
0	0	1	1	1	D			KSP[1]	KSP[2]	KSP[0]
0	1	0	0	0	C		KSP[0]			
0	1	1	0	0	C		KSP[1]	KSP[0]		
0	1	1	1	0	C		KSP[2]	KSP[1]	KSP[0]	
1	0	0	0	0	C	KSP[0]				
1	0	1	0	0	C	KSP[1]		KSP[0]		
1	0	1	1	0	C	KSP[2]		KSP[1]	KSP[0]	
1	1	0	0	0	C	KSP[1]	KSP[0]			
1	1	1	0	0	C	KSP[2]	KSP[1]	KSP[0]		

[DevBW] and [DevCL] only:

The WM unit will select the optimal dispatch size given the enabled modes and the number of subspans remaining in the object (n), via the following algorithm: (note: This algorithm assumes a valid set of state variables, as listed in the table below, the Valid column indicates which products that the combinationis supported on. Combinations of dispatch enables not listed in the table are not available on any product.

A: Valid on all products

B: Valid only on [DevCTG+]. C: Valid only on [DevCTG], and [DevILK]

D: Valid on all products.

E: Valid on all products.

F: Valid on all products

For [DevILK], there is only one kernel start pointer (KSP) specified in WM_STATE, with other kernels being entered via an offset from the single KSP as fiollows:

KSP[0] = KSP

KSP[1] = KSP + 1

KSP[2] = KSP + 2

KSP[3] = KSP + 3

For {DevILK+], each of the four KSP values is separately specified. IN addition, each kernel has a separately-specified GRF register count, whereas on [Pre-DevILK], all kernels share the same GRF register cont field, with the one with the maximum register count required applying to all

Table 8-2

```
      if (32PixelDispatchEnable && n>7)
            Dispatch 32 Pixels
      else if (16PixelDispatchEnable && (n>2 || !8PixelDispatchEnable))
            Dispatch 16 Pixels
      else
            Dispatch 8 Pixels
```

Depending on the subspan grouping selected, the WM unit will modify the starting PS Instruction Pointer (derived from the Kernel Start Pointer in WM_STATE) as a means to inform the PS kernel of the number of subspans included in the payload. The modified IP is a function of the enabled modes and the dispatch size, as shown in the table.

The driver must ensure that the PS kernel begins with a corresponding jump table to properly handle the number of subspans dispatched. The WM unit will “OR” in the two lsbs of the Kernel Pointer (bits 5:4) to create an instruction level address (note that the pointer from WM_STATE is 64 byte aligned which correpsonds to four instructions).

If only one dispatch mode is enabled, the Jitter should not include any jump table entries at the beginning of the PS kernel. If multiple dispatch modes are enabled, a two entry jump table should always be inserted, regardless of which modes are enabled (jump table entry for 8 pixel dispatch, followed by jump table entry for 32 pixel dispatch).

Note that for a 32 pixel dispatch, the Windower will mulitply the **Dispatch GRF Start Register for URB Data** state by 2 to account for the extra payload data required. The Pixel Shader kernel needs to comprehend this modification for the 32 pixel kernel code.

8.5.1.1 Contiguous Dispatch Modes

[DevCTG] to [DevILK] only

Two contiguous dispatch modes are also available, where the pixels dispatched are guaranteed to be contiguous and aligned as follows:

- Contiguous 64 pixel dispatch: 8 wide by 8 high pixel block aligned to 8x8 grid relative to the render target origin.
- Contiguous 32 pixel dispatch: 8 wide by 4 high pixel block aligned to 8x4 grid relative to the render target origin.

These dispatch modes have a different payload than the normal dispatch modes as documented in the section titled *PS Thread Payload for Contiguous Dispatch*. Only a single X and Y pair is provided representing the upper left pixel within the block. There is also no provision for sending depth, stencil, or antialias alpha data into the thread, and the thread must be terminated with a *Render Target UNORM Write* message rather than with a *Render Target Write* message.

There are three cases to consider depending on which dispatch modes are enabled based on the legal combinations in the table above:

Only normal dispatch modes are enabled. This is the normal operating mode in which all features are supported.

Only contiguous dispatch modes are enabled. In this case, software must ensure that the fast composite restrictions are met.

Both normal and contiguous dispatch modes are enabled. In this case, a combination of software and the setup kernel must check all of the restrictions required by the contiguous dispatch pixel shader code. The result of the check in the setup kernel is indicated in the message descriptor of the URB write message. The windower then chooses a dispatch mode from either the normal category or the contiguous category depending on whether the restriction check fails or passes, respectively.

If both the 32- and 64-pixel contiguous dispatch modes are enabled together, the windower will choose which one to use based on whether at least one pixel from the upper and lower 8x4 halves of the 8x8 block is active. If one half has no pixel active, the half that does have pixels active will be dispatched as a 32-pixel thread.

The following logic describes how the windower chooses the dispatch mode based on which modes are enabled:

```
d32 = normal 32-pixel dispatch mode enabled
d16 = normal 16-pixel dispatch mode enabled
d8 = normal 8-pixel dispatch mode enabled
c64 = contiguous 64-pixel dispatch mode enabled
c32 = contiguous 32-pixel dispatch mode enabled

ContiguousSelect = (c64 || c32) &&
                        [!(d32 || d16 || d8) || RestrictionCheckPass]
```

For ContiguousSelect true:

contiguous area available	first priority	second priority
both superspan halves	c64	c32
one superspan half	c32	c64

For ContiguousSelect false:

subspans available	first priority	second priority	third priority
s >= 4	d32	d16	d8
4 > s >= 2	d16	d8	d32
2 > s >= 1	d8	d16	d32

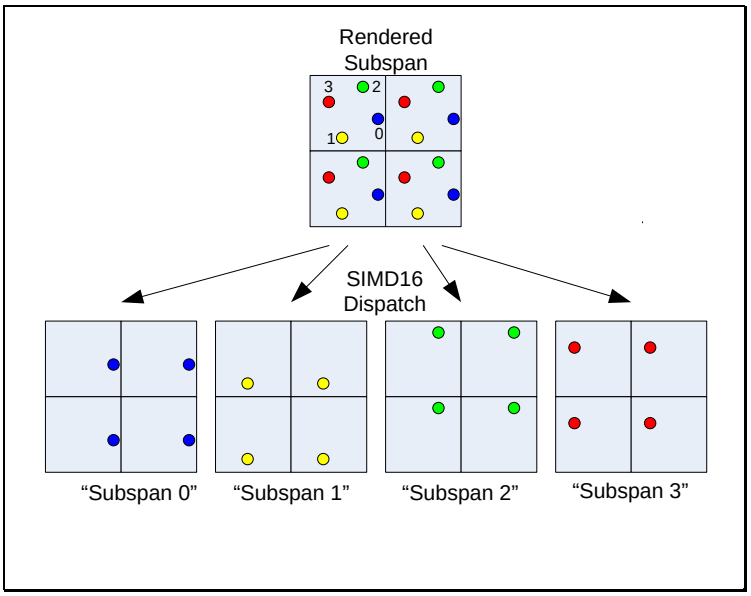
8.5.1.2 MSDISPMODE_PERP IXEL Thread Dispatch

In PERPIXEL mode, the pixel shader kernel still works on 2/4/8 separate subspans, depending on dispatch mode. The fact that rasterization and the depth/stencil tests are being performed on a per-sample (not per-pixel) basis is transparent to the pixel shader kernel.

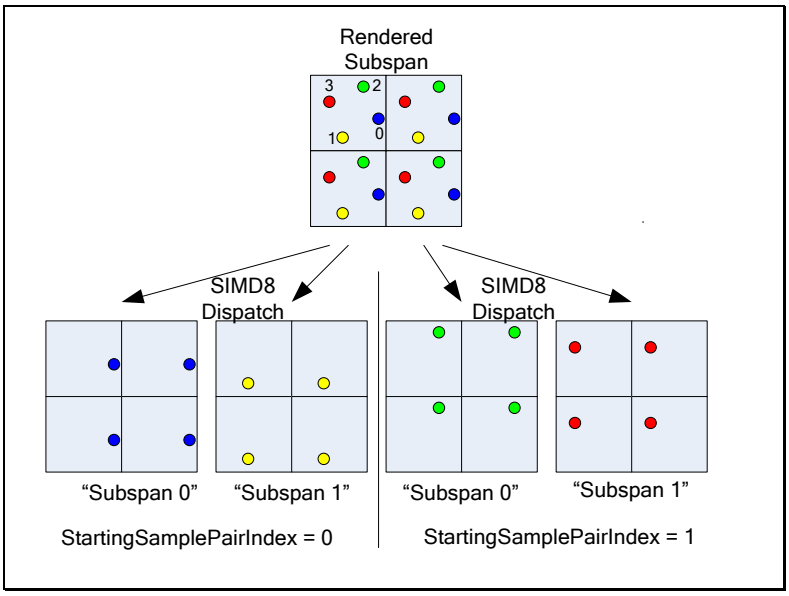
8.5.1.3 MSDISPMODE_PERSAM PLE Thread Dispatch

In PERSAMPLE mode, the pixel shader needs to operate on a sample vs. pixel basis (although this collapses in NUMSAMPLES_1 mode). Instead of processing strictly different subspans in parallel , the PS kernel processes different sample indices of one or more subspans in parallel. For example, a SIMD16 dispatch in PERSAMPLE/NUMSAMPLES_4 mode would operate on a single subspan, with the usual “4 Subspan0 pixel slots” used for the “4 Sample0 locations of the (single) subspan”. Subspan1 slots would be used for the Sample1 locations, and so on. This layout allows the pixel shader to compute derivatives/LOD based on deltas between corresponding sample locations in the subspan in the same fashion as LEGACY pixel shader execution.

Depending on the dispatch mode (8/16/32 pixels) and multisampling mode (1X/4X), there are different mappings of subspans/samples onto dispatches and slots-within-dispatch. In some cases, more than one subspan may be included in a dispatch, while in other cases multiple dispatches are be required to process all samples for a single subspan. In the latter case, the **StartingSamplePairIndex** value is included in the payload header so the Render Target Write message will access the correct samples with each message.



PERSAMPLE SIMD16 4X Dispatch



PERSAMPLE SIMD8 4X Dispatch

The following table provides the complete dispatch/slot mappings for all the Dispatch combinations.

Dispatch Size	Num Samples	Slot Mapping (SSPI = Starting Sample Pair Index)
SIMD32	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0] Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0] Slot[19:16] = Subspan[4].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[5].Pixel[3:0].Sample[0] Slot[27:24] = Subspan[6].Pixel[3:0].Sample[0] Slot[31:28] = Subspan[7].Pixel[3:0].Sample[0]
	4X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3] Slot[19:16] = Subspan[1].Pixel[3:0].Sample[0] Slot[23:20] = Subspan[1].Pixel[3:0].Sample[1] Slot[27:24] = Subspan[1].Pixel[3:0].Sample[2] Slot[31:28] = Subspan[1].Pixel[3:0].Sample[3]
SIMD16	8X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3] Slot[19:16] = Subspan[0].Pixel[3:0].Sample[4] Slot[23:20] = Subspan[0].Pixel[3:0].Sample[5] Slot[27:24] = Subspan[0].Pixel[3:0].Sample[6] Slot[31:28] = Subspan[0].Pixel[3:0].Sample[7]
	1X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0] Slot[11:8] = Subspan[2].Pixel[3:0].Sample[0] Slot[15:12] = Subspan[3].Pixel[3:0].Sample[0]
	4X	Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0] Slot[7:4] = Subspan[0].Pixel[3:0].Sample[1] Slot[11:8] = Subspan[0].Pixel[3:0].Sample[2] Slot[15:12] = Subspan[0].Pixel[3:0].Sample[3]
	8X	Dispatch[i]: (i=0, 2)

		SSPI = i ***Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]*** ***Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]*** ***Slot[11:8] = Subspan[0].Pixel[3:0].Sample[SSPI*2+2]*** ***Slot[15:12] = Subspan[0].Pixel[3:0].Sample[SSPI*2+3]***
SIMD8	***1X***	***Slot[3:0] = Subspan[0].Pixel[3:0].Sample[0]*** ***Slot[7:4] = Subspan[1].Pixel[3:0].Sample[0]***
	4X	***Dispatch[i]: (i=0..1)*** ***SSPI = i*** ***Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]*** ***Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]***
	8X	***Dispatch[i]: (i=0, 1, 2, 3)*** ***SSPI = i*** ***Slot[3:0] = Subspan[0].Pixel[3:0].Sample[SSPI*2+0]*** ***Slot[7:4] = Subspan[0].Pixel[3:0].Sample[SSPI*2+1]***

8.5.2 PS Thread Payload for Normal Dispatch

The following table lists all possible contents included in a PS thread payload, in the order they are provided. Certain portions of the payload are optional, in which case the corresponding phase is skipped.

This payload does not apply to the contiguous dispatch modes on **[DevCTG+]**. The payload for these modes are documented in the section titled *PS Thread Payload for Contiguous Dispatch*.

8.5.2.1 [Pre-DevSNB]

The following payload applies only to [Pre-DevSNB] devices. All registers are numbered starting at 0, but many registers are skipped depending on configuration. This causes all registers below to be renumbered to fill in the skipped locations. The only case where actual registers may be skipped is immediately before the CURBE data and again before the setup URB data.

DWord	Bit	Description
R0.7	31	Reserved
	30:24	Reserved
	23:0	**Primitive Thread ID:** This field contains the primitive thread count passed to the Windower from the Strips Fans Unit. Format: Reserved for HW Implementation Use.
R0.6	31:24	Reserved

DWord	Bit	Description
	23:0	**Thread ID:** This field contains the thread count which is incremented by the Windower for every thread that is dispatched. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer:** Specifies the 1K-byte aligned pointer to the scratch space available for this PS thread. This is specified as an offset to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	**Color Code:** This ID is assigned by the Windower unit and is used to track synchronizng events. Format: Reserved for HW Implementation Use.
	7:0	**FFTID:** This ID is assigned by the WM unit and is a identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the Sampler State table. It is specified as an offset from the **General State Base Address**. Format = GeneralStateOffset[31:5]
	4	Reserved
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. **Programming Notes:** • **[DevBW-A] A0 Erratum BWT005**: The range [0,11] for this register indicates [1KB, 12KB] in 1K byte increments. If MMIO register 21D0h bit 3 is set, then value 11 is an exception and indicates a 256KB space instead of 12KB. Note that Scratch Space Base Pointer must be 8MB-aligned in order to set the 256KB scratch space. • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two
R0.2	31:0	Reserved : delivered as zeros (reserved for message header fields)
R0.1	31:6	**Color Calculator State Pointer:**Specifies the 64-byte aligned pointer to the Color Calculator state (CC_STATE structure in memory). It is specified as an offset from the **General State Base Address**. This value is eventually passed to the ColorCalc function in the DataPort and is used to fetch the corresponding CC_STATE data. Format = GeneralStateOffset[31:6]
	5:0	Reserved

DWord	Bit	Description
R0.0	31:16	**Pixel Mask (SubSpan[3:0]) :** Indicates which pixels within the four subspans are lit. If 32 pixel dispatch is enabled, this field contains the pixel mask for the first four subspans. Note: This is not a duplicate of the Dispatch Mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel Mask Copy (SubSpan[3:0]) :** This is a duplicate copy of the pixel mask. This copy can be modified as the pixel shader thread executesin order to turn off pixels based on kill instructions.
R1.7	31	Reserved
	30:27	**Viewport Index:** Specifies the index of the viewport currently being used. Format = U4 Range = [0,15]
	26:16	**Render Target Array Index:**Specifies the array index to be used for the following surface types: SURFTYPE_1D: specifies the array index. Range = [0,511] SURFTYPE_2D: specifies the array index. Range = [0,511] SURFTYPE_3D: specifies the “r” coordinate. Range = [0,2047] SURFTYPE_CUBE: specifies the face identifier. Range = [0,5] face Render Target Array Index +x 0 -x 1 +y 2 -y 3 +z 4 -z 5 Format = U11
	15:0	Reserved
R1.6	31	**Front/Back Facing Polygon:**Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use. 0: Front Facing 1: Back Facing
	30	**Source Depth Present:**Indicates that source depth is included in the dispatch
	29	**Source Depth to Render Target:** Indicates that source depth will be sent to the render target
	28	**Destination Depth Present:** Indicates that destination depth is included in the dispatch and sent to the render target
	27	**Destination Stencil Present:** Indicates that destination stencil is included in the dispatch and sent to the render target

DWord	Bit	Description
	26	**Antialias Alpha to Render Target:** Indicates to the PS thread that antialias alpha data must be included in render target writes (i.e., included in the DataPort RT Write message payload). The WM unit generates this control bit based on object type and state settings. This indication is required as the PS kernel is likely shared between anti-aliased and non-anti-aliased objects. This bit applies to all subspans (i.e., both sets of 4 subspans for 32-pixel dispatches). By definition, **Antialias Alpha Present** will also be set. Format: Enable
	25	**Antialias Alpha Present:** Indicates that antialias alpha data is included in this PS thread payload. This bit applies to all subspans (i.e., both sets of 4 subspans for 32-pixel dispatches). Format: Enable
	24:5	Reserved
	4:0	**Primitive Topology Type:**This field identifies the Primitive Topology Type associated with the primitive spawning this object. The WM unit does not modify this value (e.g., objects within POINTLIST topologies see POINTLIST). Format: (See 3DPRIMITIVE command in *3D Pipeline*)
R1.5	31:16	**Y3:** Y coordinate (screen space) for upper-left pixel of subspan 3 Format = U16
	15:0	**X3:** X coordinate (screen space) for upper-left pixel of subspan 3 Format = U16
R1.4	31:16	**Y2 :** Y coordinate (screen space) for upper-left pixel of subspan 2 Format = U16
	15:0	**X2 :** X coordinate (screen space) for upper-left pixel of subspan 2 Format = U16
R1.3	31:16	**Y1 :** Y coordinate (screen space) for upper-left pixel of subspan 1 Format = U16
	15:0	**X1 :** X coordinate (screen space) for upper-left pixel of subspan 1 Format = U16
R1.2	31:16	**Y0 :** Y coordinate (screen space) for upper-left pixel of subspan 0 Format = U16
	15:0	**X0 :** X coordinate (screen space) for upper-left pixel of subspan 0 Format = U16
R1.1	31:0	**Ystart :** Y coordinate (screen space) for the start vertex (V0, upper left vertex of the object, as selected by the SF unit) Format = IEEE_Float
R1.0	31:0	**Xstart:** X coordinate (screen space) for the start vertex (V0, upper left vertex of the object, as selected by the SF unit) Format = IEEE_Float

DWord Bit		Description
		The following data is optional depending on the state relating to depth / stencil / alpha present flags above. Phases including only data for subspans 2 and 3 *are* included for 8-pixel dispatches, even though they do not contain valid data. Following the optional data is the attribute interpolation coefficient data
		R2-R3: delivered only if **Source Depth Present** is set.
R2.7	31:0	Interpolated Depth for Subspan 1, Pixel 3 (lower right) Format = IEEE_Float
R2.6	31:0	Interpolated Depth for Subspan 1, Pixel 2 (lower left)
R2.5	31:0	Interpolated Depth for Subspan 1, Pixel 1 (upper right)
R2.4	31:0	Interpolated Depth for Subspan 1, Pixel 0 (upper left)
R2.3	31:0	Interpolated Depth for Subspan 0, Pixel 3 (lower right)
R2.2	31:0	Interpolated Depth for Subspan 0, Pixel 2 (lower left)
R2.1	31:0	Interpolated Depth for Subspan 0, Pixel 1 (upper right)
R2.0	31:0	Interpolated Depth for Subspan 0, Pixel 0 (upper left)
R3.7	31:0	Interpolated Depth for Subspan 3, Pixel 3 (lower right)
R3.6	31:0	Interpolated Depth for Subspan 3, Pixel 2 (lower left)
R3.5	31:0	Interpolated Depth for Subspan 3, Pixel 1 (upper right)
R3.4	31:0	Interpolated Depth for Subspan 3, Pixel 0 (upper left)
R3.3	31:0	Interpolated Depth for Subspan 2, Pixel 3 (lower right)
R3.2	31:0	Interpolated Depth for Subspan 2, Pixel 2 (lower left)
R3.1	31:0	Interpolated Depth for Subspan 2, Pixel 1 (upper right)
R3.0	31:0	Interpolated Depth for Subspan 2, Pixel 0 (upper left)
		R4: delivered only if **Antialias Alpha Present** or **Destination Stencil Present** is set. The Antialias Alpha data is only valid if **Antialias Alpha Present** is set, and likewise the Destination Stencil data is only valid if **Destination Stencil Present** is set.
		[DevCTG]
R4.7	31:24	Antialias Alpha for Subspan 3, Pixel 3 (lower right) This field contains the coverage value associated with Pixel 3 of Subspan 7. Format = U0.8
	23:16	Antialias Alpha for Subspan 3, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 3, Pixel 1 (upper right)
R4.6	7:0	Antialias Alpha for Subspan 3, Pixel 0 (upper left)
	31:24	Antialias Alpha for Subspan 2, Pixel 3 (lower right)
	23:16	Antialias Alpha for Subspan 2, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 2, Pixel 1 (upper right)
	7:0	Antialias Alpha for Subspan 2, Pixel 0 (upper left)

DWord	Bit	Description
R4.5	31:24	Antialias Alpha for Subspan 1, Pixel 3 (lower right)
	23:16	Antialias Alpha for Subspan 1, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 1, Pixel 1 (upper right)
	7:0	Antialias Alpha for Subspan 1, Pixel 0 (upper left)
R4.4	31:24	Antialias Alpha for Subspan 0, Pixel 3 (lower right)
	23:16	Antialias Alpha for Subspan 0, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 0, Pixel 1 (upper right)
	7:0	Antialias Alpha for Subspan 0, Pixel 0 (upper left)
		[DevBW, DevCL]
R4.7	31:28	Antialias Alpha for Subspan 3, Pixel 3 (lower right) This field contains the coverage value associated with Pixel 3 of Subspan 7. Format = U0.4
	27:24	Antialias Alpha for Subspan 3, Pixel 2 (lower left)
	23:20	Antialias Alpha for Subspan 3, Pixel 1 (upper right)
	19:16	Antialias Alpha for Subspan 3, Pixel 0 (upper left)
	15:12	Antialias Alpha for Subspan 2, Pixel 3 (lower right)
	11:8	Antialias Alpha for Subspan 2, Pixel 2 (lower left)
	7:4	Antialias Alpha for Subspan 2, Pixel 1 (upper right)
	3:0	Antialias Alpha for Subspan 2, Pixel 0 (upper left)
R4.6	31:28	Antialias Alpha for Subspan 1, Pixel 3 (lower right)
	27:24	Antialias Alpha for Subspan 1, Pixel 2 (lower left)
	23:20	Antialias Alpha for Subspan 1, Pixel 1 (upper right)
	19:16	Antialias Alpha for Subspan 1, Pixel 0 (upper left)
	15:12	Antialias Alpha for Subspan 0, Pixel 3 (lower right)
	11:8	Antialias Alpha for Subspan 0, Pixel 2 (lower left)
	7:4	Antialias Alpha for Subspan 0, Pixel 1 (upper right)
	3:0	Antialias Alpha for Subspan 0, Pixel 0 (upper left)
R4.5:4		Reserved
R4.3	31:24	Destination Stencil for Subspan 3, Pixel 3 (lower right) Format = U8
	23:16	Destination Stencil for Subspan 3, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 3, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 3, Pixel 0 (upper left)

DWord	Bit	Description
R4.2	31:24	Destination Stencil for Subspan 2, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 2, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 2, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 2, Pixel 0 (upper left)
R4.1	31:24	Destination Stencil for Subspan 1, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 1, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 1, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 1, Pixel 0 (upper left)
R4.0	31:24	Destination Stencil for Subspan 0, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 0, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 0, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 0, Pixel 0 (upper left)
		R5-R6: delivered only if **Destination Depth Present** is set.
R5.7	31:0	Destination Depth for Subspan 1, Pixel 3 (lower right) Format depends on depth buffer surface format, and is intended to be passed through to the render target without modification by software.
R5.6	31:0	Destination Depth for Subspan 1, Pixel 2 (lower left)
R5.5	31:0	Destination Depth for Subspan 1, Pixel 1 (upper right)
R5.4	31:0	Destination Depth for Subspan 1, Pixel 0 (upper left)
R5.3	31:0	Destination Depth for Subspan 0, Pixel 3 (lower right)
R5.2	31:0	Destination Depth for Subspan 0, Pixel 2 (lower left)
R5.1	31:0	Destination Depth for Subspan 0, Pixel 1 (upper right)
R5.0	31:0	Destination Depth for Subspan 0, Pixel 0 (upper left)
R6.7	31:0	Destination Depth for Subspan 3, Pixel 3 (lower right)
R6.6	31:0	Destination Depth for Subspan 3, Pixel 2 (lower left)
R6.5	31:0	Destination Depth for Subspan 3, Pixel 1 (upper right)
R6.4	31:0	Destination Depth for Subspan 3, Pixel 0 (upper left)
R6.3	31:0	Destination Depth for Subspan 2, Pixel 3 (lower right)
R6.2	31:0	Destination Depth for Subspan 2, Pixel 2 (lower left)
R6.1	31:0	Destination Depth for Subspan 2, Pixel 1 (upper right)
R6.0	31:0	Destination Depth for Subspan 2, Pixel 0 (upper left)
		R7: delivered only if this is a *32-pixel dispatch*.
R7.7	31:0	Reserved
R7.6	31:0	Reserved
R7.5	31:0	Reserved

DWord	Bit	Description
R7.5	31:16	**Y7:** Y coordinate (screen space) for upper-left pixel of subspan 7 Format = U16
	15:0	**X7:** X coordinate (screen space) for upper-left pixel of subspan 7 Format = U16
R7.4	31:16	**Y6**
	15:0	**X6**
R7.3	31:16	**Y5**
	15:0	**X5**
R7.2	31:16	**Y4**
	15:0	**X4**
R7.1	31:0	Reserved
R7.0	31:16	**Pixel Mask (SubSpan[7:4]) :** Indicates which pixels within the upper four subspans are lit. This field is valid only when the 32 pixel dispatch state is enabled. This field must not be modified by the pixel shader thread. Note: This is not a duplicate of the dispatch mask that is delivered to the thread. The dispatch mask has all pixels within a subspan as active if any of them are lit to enable LOD calculations to occur correctly. This field must not be modified by the Pixel Shader kernel.
	15:0	**Pixel Mask Copy (SubSpan[7:4]) :** This is a duplicate copy of pixel mask for the upper 16 pixels. This copy will be modified as the pixel shader thread executes to turn off pixels based on kill instructions.
		R8-R9: delivered only if **Source Depth Present** is set and this is a *32-pixel dispatch*.
R8.7	31:0	Interpolated Depth for Subspan 5, Pixel 3 (lower right) Format = IEEE_Float
R8.6	31:0	Interpolated Depth for Subspan 5, Pixel 2 (lower left)
R8.5	31:0	Interpolated Depth for Subspan 5, Pixel 1 (upper right)
R8.4	31:0	Interpolated Depth for Subspan 5, Pixel 0 (upper left)
R8.3	31:0	Interpolated Depth for Subspan 4, Pixel 3 (lower right)
R8.2	31:0	Interpolated Depth for Subspan 4, Pixel 2 (lower left)
R8.1	31:0	Interpolated Depth for Subspan 4, Pixel 1 (upper right)
R8.0	31:0	Interpolated Depth for Subspan 4, Pixel 0 (upper left)
R9.7	31:0	Interpolated Depth for Subspan 7, Pixel 3 (lower right)
R9.6	31:0	Interpolated Depth for Subspan 7, Pixel 2 (lower left)
R9.5	31:0	Interpolated Depth for Subspan 7, Pixel 1 (upper right)
R9.4	31:0	Interpolated Depth for Subspan 7, Pixel 0 (upper left)
R9.3	31:0	Interpolated Depth for Subspan 6, Pixel 3 (lower right)
R9.2	31:0	Interpolated Depth for Subspan 6, Pixel 2 (lower left)

DWord	Bit	Description
R9.1	31:0	Interpolated Depth for Subspan 6, Pixel 1 (upper right)
R9.0	31:0	Interpolated Depth for Subspan 6, Pixel 0 (upper left)
		R10: delivered only if **Antialias Alpha Present** or **Destination Stencil Present** is set and this is a 32-pixel dispatch. The Antialias Alpha data is only valid if **Antialias Alpha Present** is set, and likewise the Destination Stencil data is only valid if **Destination Stencil Present** is set.
		[DevCTG]
R10.7	31:24	Antialias Alpha for Subspan 7, Pixel 3 (lower right) This field contains the coverage value associated with Pixel 3 of Subspan 7. Format = U0.8
	23:16	Antialias Alpha for Subspan 7, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 7, Pixel 1 (upper right)
R10.6	7:0	Antialias Alpha for Subspan 7, Pixel 0 (upper left)
	31:24	Antialias Alpha for Subspan 6, Pixel 3 (lower right)
	23:16	Antialias Alpha for Subspan 6, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 6, Pixel 1 (upper right)
	7:0	Antialias Alpha for Subspan 6, Pixel 0 (upper left)
R10.5	31:24	Antialias Alpha for Subspan 5, Pixel 3 (lower right)
	23:16	Antialias Alpha for Subspan 5, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 5, Pixel 1 (upper right)
	7:0	Antialias Alpha for Subspan 5, Pixel 0 (upper left)
R10.4	31:24	Antialias Alpha for Subspan 4, Pixel 3 (lower right)
	23:16	Antialias Alpha for Subspan 4, Pixel 2 (lower left)
	15:8	Antialias Alpha for Subspan 4, Pixel 1 (upper right)
	7:0	Antialias Alpha for Subspan 4, Pixel 0 (upper left)
		[DevBW, DevCL]
R10.7	31:28	Antialias Alpha for Subspan 7, Pixel 3 (lower right) This field contains the coverage value associated with Pixel 3 of Subspan 7. Format = U0.4
	27:24	Antialias Alpha for Subspan 7, Pixel 2 (lower left)
	23:20	Antialias Alpha for Subspan 7, Pixel 1 (upper right)
	19:16	Antialias Alpha for Subspan 7, Pixel 0 (upper left)
	15:12	Antialias Alpha for Subspan 6, Pixel 3 (lower right)
	11:8	Antialias Alpha for Subspan 6, Pixel 2 (lower left)
	7:4	Antialias Alpha for Subspan 6, Pixel 1 (upper right)
	3:0	Antialias Alpha for Subspan 6, Pixel 0 (upper left)

DWord	Bit	Description
R10.6	31:28	Antialias Alpha for Subspan 5, Pixel 3 (lower right)
	27:24	Antialias Alpha for Subspan 5, Pixel 2 (lower left)
	23:20	Antialias Alpha for Subspan 5, Pixel 1 (upper right)
	19:16	Antialias Alpha for Subspan 5, Pixel 0 (upper left)
	15:12	Antialias Alpha for Subspan 4, Pixel 3 (lower right)
	11:8	Antialias Alpha for Subspan 4, Pixel 2 (lower left)
	7:4	Antialias Alpha for Subspan 4, Pixel 1 (upper right)
	3:0	Antialias Alpha for Subspan 4, Pixel 0 (upper left)
R10.5:4		Reserved
R10.3	31:24	**Destination Stencil for Subspan 7, Pixel 3 (lower right) :** This field contains the destination stencil value associated with Pixel 3 of Subspan 7. Format = U8
	23:16	Destination Stencil for Subspan 7, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 7, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 7, Pixel 0 (upper left)
R10.2	31:24	Destination Stencil for Subspan 6, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 6, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 6, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 6, Pixel 0 (upper left)
R10.1	31:24	Destination Stencil for Subspan 5, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 5, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 5, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 5, Pixel 0 (upper left)
R10.0	31:24	Destination Stencil for Subspan 4, Pixel 3 (lower right)
	23:16	Destination Stencil for Subspan 4, Pixel 2 (lower left)
	15:8	Destination Stencil for Subspan 4, Pixel 1 (upper right)
	7:0	Destination Stencil for Subspan 4, Pixel 0 (upper left)
		R11-R12: delivered only if **Destination Depth Present** is set and this is a *32-pixel dispatch*.
R11.7	31:0	Destination Depth for Subspan 5, Pixel 3 (lower right) Format = IEEE_Float
R11.6	31:0	Destination Depth for Subspan 5, Pixel 2 (lower left)
R11.5	31:0	Destination Depth for Subspan 5, Pixel 1 (upper right)
R11.4	31:0	Destination Depth for Subspan 5, Pixel 0 (upper left)
R11.3	31:0	Destination Depth for Subspan 4, Pixel 3 (lower right)

DWord	Bit	Description
R11.2	31:0	Destination Depth for Subspan 4, Pixel 2 (lower left)
R11.1	31:0	Destination Depth for Subspan 4, Pixel 1 (upper right)
R11.0	31:0	Destination Depth for Subspan 4, Pixel 0 (upper left)
R12.7	31:0	Destination Depth for Subspan 7, Pixel 3 (lower right)
R12.6	31:0	Destination Depth for Subspan 7, Pixel 2 (lower left)
R12.5	31:0	Destination Depth for Subspan 7, Pixel 1 (upper right)
R12.4	31:0	Destination Depth for Subspan 7, Pixel 0 (upper left)
R12.3	31:0	Destination Depth for Subspan 6, Pixel 3 (lower right)
R12.2	31:0	Destination Depth for Subspan 6, Pixel 2 (lower left)
R12.1	31:0	Destination Depth for Subspan 6, Pixel 1 (upper right)
R12.0	31:0	Destination Depth for Subspan 6, Pixel 0 (upper left)
		Optional Padding before the Start of URB-Sourced Data The locations between the end of the Optional Payload Header and the location programmed via **Dispatch GRF Start Register for URB Data** (if any) are considered "padding" and Reserved. (see below)
optional, multiple of 8 DWs	31:0	Reserved
		URB DATA STARTS HERE The **Dispatch GRF Start Register for URB Data** state variable in WM_STATE is used to define the starting location of URB-sourced data within the PS thread payload. This control is provided to allow the URB-sourced data to be located at a fixed location within thread payloads, regardless of the amount of data in the Optional Payload Header. This permits the kernel to use direct GRF addressing to access the URB-sourced data, regardless of the optional parameters being passed (as these are determined on-the-fly by the WM unit).
		Constant URB Entry (CURBE) Data Optionally, some amount of data (multiples of 8 DWs) can be read from the CURBE URB entry and placed in the thread payload at this point (after the variable payload header and prior to the Setup URB data). The amount of CURBE data provided is specified by **Constant URB Entry Read Length** in WM_STATE, and the starting read offset in that URB entry is specified by **Constant URB Entry Read Offset** in WM_STATE.
optional, multiple of 8 DWs	31:0	Constant Data

DWord	Bit	Description
		Setup URB Data (Attribute Interpolation Coeffcients) Some amount of data (multiples of 8 DWs) can be read from the Setup URB entry and placed in the thread payload at this point (after the variable payload header and any CURBE data – i.e., the end of the payload). This data is read from the Setup URB entry based on the URB Handle associated with the object being rendered (as received from the SF unit). The amount of Setup URB data provided is specified by **Setup URB Entry Read Length** in WM_STATE, and the starting read offset in that URB entry is specified by **Setup URB Entry Read Offset** in WM_STATE. The order/content/format of this data is actually determined by the Setup kernel which is executed from the Strips Fans Unit. The following DWords are labelled assuming the typical/expected definition.
Rp.7	31:0	Co[1] – Co Coefficient for Attribute [1] (optional)
Rp.6	31:0	Reserved
Rp.5	31:0	Cy[1] – Cy Coefficient for Attribute [1] (optional)
Rp.4	31:0	Cx[1] – Cx Coefficient for Attribute [1] (optional)
Rp.3	31:0	Co[0] – Co Coefficient for Attribute [0]
Rp.2	31:0	Reserved
Rp.1	31:0	Cy[0] – Cy Coefficient for Attribute [0]
Rp.0	31:0	Cx[0] – Cx Coefficient for Attribute [0]
R(p+1):Rq		Coefficients for additional attributes (optional) See definition of Rp for formats.

[DevILK]: Erratum: SW must assign R0.7 with R1.7 in order for separate stencil buffer to get correct RTAI

8.5.3 PS Thread Payload for Contiguous Dispatch [DevCTG] to [DevILK]

The contiguous dispatch modes have the following payload:

DWord	Bit	Description
	30:24	Reserved
	23:0	**Primitive Thread ID:** This field contains the primitive thread count passed to the Windower from the Strips Fans Unit. Format: Reserved for HW Implementation Use.
R0.6	31:24	Reserved
	23:0	**Thread ID:** This field contains the thread count which is incremented by the Windower for every thread that is dispatched. Format: Reserved for HW Implementation Use.
R0.5	31:10	**Scratch Space Pointer:** Specifies the 1K-byte aligned pointer to the scratch space available for this PS thread. This is specified as an offset to the **General State Base Address**. Format = GeneralStateOffset[31:10]
	9:8	**Color Code:** This ID is assigned by the Windower unit and is used to track synchronizng events. Format: Reserved for HW Implementation Use.
	7:0	**FFTID:** This ID is assigned by the WM unit and is a identifier for the thread. It is used to free up resources used by the thread upon thread completion. Format: Reserved for HW Implementation Use.
R0.4	31:5	**Binding Table Pointer:** Specifies the 32-byte aligned pointer to the Binding Table. It is specified as an offset from the **Surface State Base Address**. Format = SurfaceStateOffset[31:5]
	4:0	Reserved
R0.3	31:5	**Sampler State Pointer:** Specifies the 32-byte aligned pointer to the Sampler State table. It is specified as an offset from the **General State Base Address**. Format = GeneralStateOffset[31:5]
	4	Reserved
	3:0	**Per Thread Scratch Space:** Specifies the amount of scratch space allowed to be used by this thread. **Programming Notes:** • This amount is available to the kernel for information only. It will be passed verbatim (if not altered by the kernel) to the Data Port in any scratch space access messages, but the Data Port will ignore it. Format = U4 Range = [0,11] indicating [1k bytes, 2M bytes] in powers of two

DWord	Bit	Description
R0.2	31:0	**Pixel Mask 0** For Contiguous 32-Pixel Dispatch: Indicates which of the 32 pixels are lit For Contiguous 64-Pixel Dispatch: Indicates which of the 32 pixels in the upper half (8x4) are lit. The bits in this mask correspond to the pixels as follows: *0 1 4 5 16 17 20 21* *2 3 6 7 18 19 22 23* *8 9 12 13 24 25 28 29* *10 11 14 15 26 27 30 31*
R0.1	31:0	**Y :** Y coordinate (screen space) for upper-left pixel of the contiguous block Format = U32
R0.0	31:0	**X :** X coordinate (screen space) for upper-left pixel of the block Format = U32
R1.7:3		Reserved
R1.2	31:0	**Pixel Mask 1** For Contiguous 32-Pixel Dispatch: Reserved For Contiguous 64-Pixel Dispatch: Indicates which of the 32 pixels in the lower half (8x4) are lit. Refer to the bit numberings in **Pixel Mask 0** above for bit positions in this mask.
R1.1	31:0	**Ystart :** Y coordinate (screen space) for the start vertex (V0, upper left vertex of the object, as selected by the SF unit) Format = IEEE_Float
R1.0	31:0	**Xstart:** X coordinate (screen space) for the start vertex (V0, upper left vertex of the object, as selected by the SF unit) Format = IEEE_Float
		Optional Padding before the Start of URB-Sourced Data The locations between the end of the Optional Payload Header and the location programmed via **Dispatch GRF Start Register for URB Data** (if any) are considered "padding" and Reserved. (see below)
optional, multiple of 8 DWs	31:0	Reserved
		URB DATA STARTS HERE The **Dispatch GRF Start Register for URB Data** state variable in WM_STATE is used to define the starting location of URB-sourced data within the PS thread payload. This control is provided to allow the URB-sourced data to be located at a fixed location within thread payloads, regardless of the amount of data in the Optional Payload Header. This permits the kernel to use direct GRF addressing to access the URB-sourced data, regardless of the optional parameters being passed (as these are determined on-the-fly by the WM unit).

DWord	Bit	Description
		Constant Data (optional) : **[Pre-DevSNB]:** Some amount of constant data (possible none) can be extracted from the URB and passed to the thread following the R0 Header. The data is read from the Constant URB Entry at some offset (**Constant URB Entry Read Offset** state) from the handle. The amount of data provided is defined by the **Constant URB Entry Read Length** state. The Constant Data arrives in a non-interleaved format.
optional, multiple of 8 DWs	31:0	Constant Data
		Setup URB Data (Attribute Interpolation Coeffcients) Some amount of data (multiples of 8 DWs) can be read from the Setup URB entry and placed in the thread payload at this point (after the variable payload header and any CURBE data – i.e., the end of the payload). This data is read from the Setup URB entry based on the URB Handle associated with the object being rendered (as received from the SF unit). The amount of Setup URB data provided is specified by **Setup URB Entry Read Length** in WM_STATE, and the starting read offset in that URB entry is specified by **Setup URB Entry Read Offset** in WM_STATE. The order/content/format of this data is actually determined by the Setup kernel which is executed from the Strips Fans Unit. The following DWords are labelled assuming the typical/expected definition.
Rp.7	31:0	Co[1] – Co Coefficient for Attribute [1] (optional)
Rp.6	31:0	Reserved
Rp.5	31:0	Cy[1] – Cy Coefficient for Attribute [1] (optional)
Rp.4	31:0	Cx[1] – Cx Coefficient for Attribute [1] (optional)
Rp.3	31:0	Co[0] – Co Coefficient for Attribute [0]
Rp.2	31:0	Reserved
Rp.1	31:0	Cy[0] – Cy Coefficient for Attribute [0]
Rp.0	31:0	Cx[0] – Cx Coefficient for Attribute [0]
R(p+1):Rq		Coefficients for additional attributes (optional) See definition of Rp for formats.

8.6 Other WM Functions

8.6.1 Statistics Gathering

If **Statistics Enable** is set in WM_STATE or 3DSTATE_WM, the Windower increments the PS_INVOCATIONS_COUNT register once for each unmasked pixel (or sample) that is *dispatched* to a Pixel Shader thread. If **Early Depth Test Enable** is set it is possible for pixels or samples to be discarded prior to reaching the Pixel Shader due to failing the depth or stencil test. PS_INVOCATIONS_COUNT will still be incremented for these pixels or samples since the depth test occurs after the pixel shader from the point of view of SW.

[DevBW] A0 Erratum BWT004 states that there is no way to indicate a true "null" pixel shader (in the sense that the pixel shader dispatch will be skipped.) The "dummy" PS thread required for a "null" pixel shader will still cause PS_INVOCATIONS_COUNT to increment on pixel dispatches; if the "null" pixel dispatches are not to be counted, **Statistics Enable** must be *cleared* when changing to a "null" pixel shader. Clearing **Statistics Enable** may also prevent PS_DEPTH_COUNT from incrementing properly. Therefore, in certain pipeline configurations, it may be *impossible* to maintain both PS_INVOCATIONS_COUNT and PS_DEPTH_COUNT accurately.

9. Color Calculator (Output Merger)

Note: The Color Calculator logic resides in the Render Cache backing Data Port (DAP) shared function. It is described in this chapter as the Color Calc functions are naturally an extension of the 3D pipeline past the WM stage. See the DataPort chapter for details on the messages used by the Pixel Shader to invoke Color Calculator functionality.

The *Color Calculator* function within the Data Port shared function completes the processing of rasterized pixels after the pixel color and depth have been computed by the Pixel Shader. This processing is initiated when the pixel shader thread sends a Render Target Write message (see *Shared Functions*) to the Render Cache. (Note that a single pixel shader thread may send multiple Render Target Write messages, with the result that multiple render targets get updated). The pixel variables pass through a pipeline of fixed (yet programmable) functions, and the results are conditionally written into the appropriate buffers.

Pipeline Stage	Description
Alpha Test	Compare pixel alpha with reference alpha and conditionally discard pixel
Stencil Test	Compare pixel stencil value with reference and forward result to Buffer Update stage
Depth Test	Compare pix.Z with corresponding Z value in the Depth Buffer and forward result to Buffer Update stage
Color Blending	Combine pixel color with corresponding color in color buffer according to programmable function
Gamma Correction	Adjust pixel's color according to gamma function for SRGB destination surfaces.
Color Quantization	Convert "full precision" pixel color values to fixed precision of the color buffer format
Logic Ops	Combine pixel color logically with existing color buffer color (mutually exclusive with Color Blending)
Buffer Update	Write final pixel values to color and depth buffers or discard pixel without update

9.1.1 Alpha Test

The Alpha Test function can be used to discard pixels based on a comparison between the incoming pixel's alpha value and the **Alpha Test Reference** state variable in COLOR_CALC_STATE. This operation can be used to remove transparent or nearly-transparent pixels, though other uses for the alpha channel and alpha test are certainly possible.

This function is enabled by the **Alpha Test Enable** state variable in COLOR_CALC_STATE. If ENABLED, this function compares the incoming pixel's alpha value (*pixColor.Alpha*) and the reference alpha value specified by via the **Alpha Test Reference** state variable in COLOR_CALC_STATE. The comparison performed is specified by the **Alpha Test Function** state variable in COLOR_CALC_STATE.

The **Alpha Test Format** state variable is used to specify whether Alpha Test is performed using fixed-point (UNORM8) or FLOAT32 values. Accordingly, it determines whether the **Alpha Reference Value** is passed in a UNORM8 or FLOAT32 format. If UNORM8 is selected, the pixel's alpha value will be converted from floating-point to UNORM8 before the comparison.

Pixels that pass the Alpha Test proceed for further processing. Those that fail are discarded at this point in the pipeline.

If **Alpha Test Enable** is DISABLED, this pipeline stage has no effect.

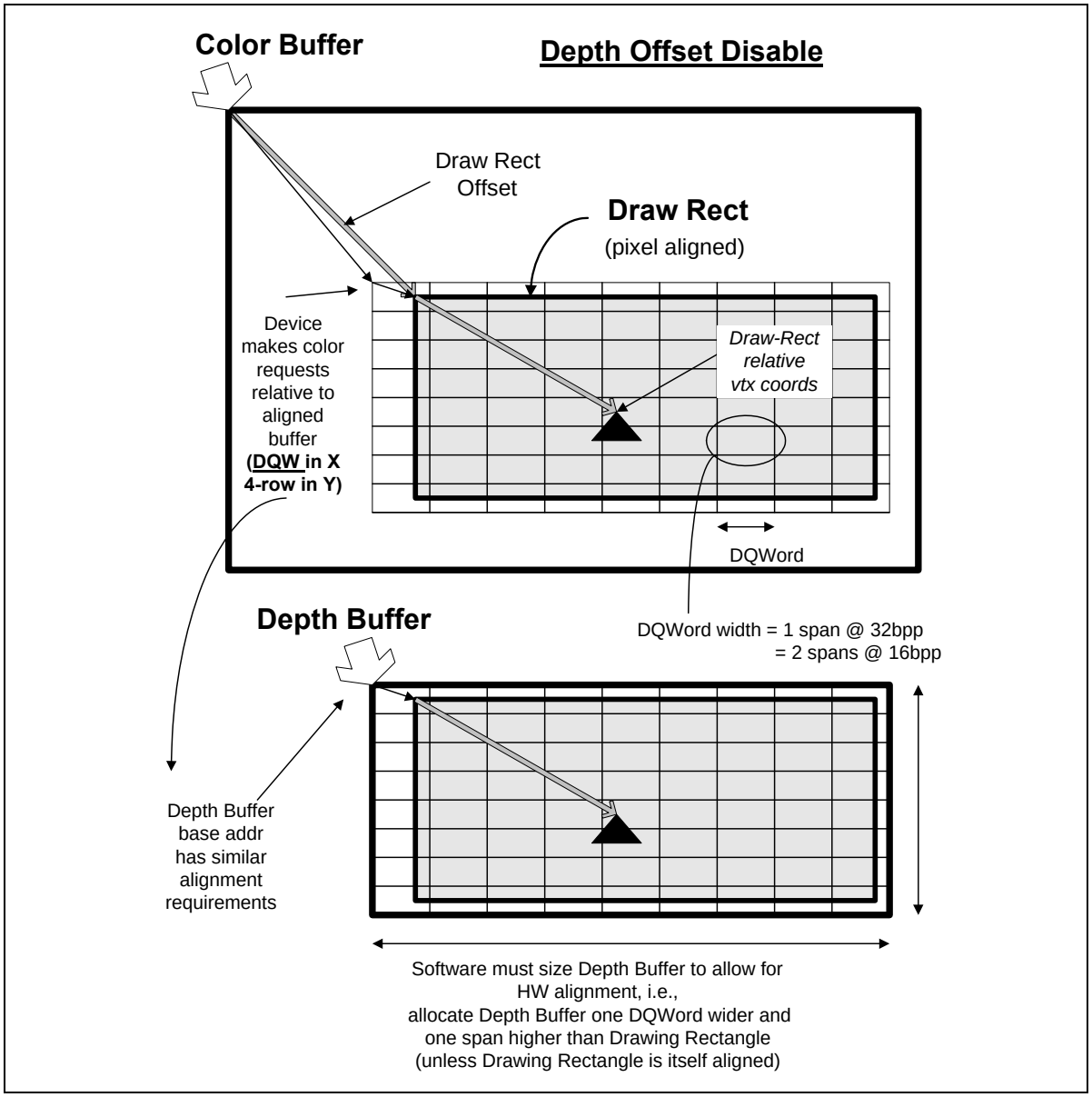
9.1.2 Depth Buffer Coordinate Offset Disable [DevBW, DevCL]

There is a capability to effectively disable the application of the Drawing Rectangle coordinate offset for accesses to the Depth Buffer. This is controlled via the **Depth Buffer Coordinate Offset Disable** state variable in the 3DSTATE_DEPTH_BUFFER command. This capability exists in order to better support "front buffer rendering" where the Color Buffer is screen-sized (by definition) while the Depth Buffer does not have to be (i.e., it may be desired to have window-sized Depth Buffer to match a window-sized back buffer). Therefore the ability to offset only the Color (front) Buffer coordinate – and not the Depth Buffer coordinate – by the **Drawing Rectangle Origin X,Y** is desired. However, due to Color/Depth Buffer access alignment issues, the offset of the Depth Buffer X,Y coordinates can not be completely disabled – a few low-order bits of the **Drawing Rectangle Origin** must still be applied to provide some alignment of Color/Depth Buffer accesses.

The alignment restrictions require:

- 2 LSBs (when rendering 32-bit color) or 3 LSBs (when rendering 16-bit color) of the **Drawing Rectangle Origin X** are unconditionally applied to the Depth Buffer X coordinate. This corresponds to one 4x4 span or two 4x4 span alignment, respectively.
- 2 LSBs of **Drawing Rectangle Origin Y** are unconditionally applied to the Depth Buffer Y coordinate (i.e., 4-row co-alignment in Y)

Figure 9-1. Drawing Rectangle Offset



9.1.3 Depth Coordinate Offset [DevCTG+]

[This is replaces the DevBW,DevCL Depth Coordinate Offset Disable (DBCOD) feature.]

The Depth Coordinate Offset function applies a programmable constant offset to the RenderTarget X,Y screen space coordinates in order to generate DepthBuffer coordinates.

The function has been specifically added to allow the OpenGL driver to deal with a RenderTarget and DepthBuffer of differing sizes. OpenGL defines a lower-left screen coordinate origin. This requires the driver to incorporate a "Y coordinate flipping" transformation into the viewport mapping function. The Y extent of the RT is used in this flipping transformation. If the DepthBuffer extent is different, the wrong pixel Y locations within the DepthBuffer will be accessed.

The least expensive solution is to provide a translation offset to be applied to the post-viewport-mapped DepthBuffer Y pixel coordinate, effectively allowing the alignment of the lower-left origins of the RT and DepthBuffer. [Note that the previous DBCOD feature performed an optional translation of post-viewport-mapping RT pixel (screen) coordinates to generate DepthBuffer pixel (window) coordinates. Specifically, the Draw Rect Origin X,Y state could be subtracted from the RT pixel coordinates.]

This function uses **Depth Coordinate Offset X,Y** state (signed 16-bit values in 3DSTATE_DEPTH_RECTANGLE) that is unconditionally added to the RT pixel coordinates to generate DepthBuffer pixel coordinates.

The previous DBCOB feature can be supported by having the driver program Depth Coordinate X,Y Offset to the two's complement of the the Draw Rect Origin. By programming Depth Coordinate X,Y Offset to zeros, the current "normal" operation (DBCOD disabled) can be achieved.

Programming Restrictions:

- Only simple 2D RTs are supported (no mipmaps)
- Software must ensure that the resultant DepthBuffer Coordinate X,Y values are non-negative.
- There are alignment restrictions – see 3DSTATE_DEPTH_BUFFER command.

9.1.4 Stencil Test

The Stencil Test function can be used to discard pixels based on a comparison between the [**Backface**] **Stencil Test Reference** state variable and the pixel's stencil value. This is a general purpose function used for such effects as shadow volumes, per-pixel clipping, etc. The result of this comparison is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Stencil Test Enable** state variable. If ENABLED, the current stencil buffer value for this pixel is read.

Programming Notes:

- If the Depth Buffer is either undefined or does **not** have a surface format of D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT, **Stencil Test Enable** must be DISABLED.

A 2nd set of the stencil test state variables is provided so that pixels from back-facing objects, assuming they are not culled, can have a stencil test performed on them separate from the test for normal front-facing objects. The separate stencil test for back-facing objects can be enabled via the **Double Sided Stencil Enable** state variable. Otherwise, non-culled back-facing objects will use the same test function, mask and reference value as front-facing objects. The 2nd stencil state for back-facing objects is most commonly used to improve the performance of rendering shadow volumes which require a different stencil buffer operation depending on whether pixels rendered are from a front-facing or back-facing object. The backface stencil state removes the requirement to render the shadow volumes in 2 passes or sort the objects into front-facing and back-facing lists.

The remainder of this subsection describes the function in term of [**Backface**] <**state variable name**>. The Backface set of state variables are only used if Double Sided Stencil Enable is ENABLED and the object is considered back-facing. Otherwise the normal (front-facing) state variables are used.

This function then compares the [**Backface**] **Stencil Test Reference** value and the pixel's stencil value value after logically ANDing both values by [**Backface**] **Stencil Test Mask**. The comparison performed is specified by the [**Backface**] **Stencil Test Function** state variable. The result of the comparison is passed down the pipeline for use in the Stencil Buffer Update function. The Stencil Test function does not in itself discard pixels.

If **Stencil Test Enable** is DISABLED, a result of "stencil test passed" is propagated down the pipeline.

9.1.5 Depth Test

The Depth Test function can be used to discard pixels based on a comparison between the incoming pixel's depth value and the current depth buffer value associated with the pixel. This function is typically used to perform the "Z Buffer" hidden surface removal. The result of this pipeline function is used in the Stencil Buffer Update function later in the pipeline.

This function is enabled by the **Depth Test Enable** state variable. If enabled, the pixel's ("source") depth value is first computed. After computation the pixel's depth value is clamped to the range defined by **Minimum Depth** and **Maximum Depth** in the selected CC_VIEWPORT state. Then the current ("destination") depth buffer value for this pixel is read.

This function then compares the source and destination depth values. The comparison performed is specified by the **Depth Test Function** state variable.

The result of the comparison is propogated down the pipeline for use in the subsequent Depth Buffer Update function. The Depth Test function does not in itself discard pixels.

If **Depth Test Enable** is DISABLED, a result of "depth test passed" is propagated down the pipeline.

Programming Notes:

- Enabling the Depth Test function without defining a Depth Buffer is UNDEFINED.

9.1.6 Pre-Blend Color Clamping

Pre-Blend Color Clamping, controlled via **Pre-Blend Color Clamp Enable** and **Color Clamp Range** states in COLOR_CALC_STATE, is affected by the enabling of Color Buffer Blend as described below.

The following table summarizes the requirements involved with Pre-/Post-Blend Color Clamping.

Blending	RT Format	Pre-Blend Color Clamp	Post-Blend Color Clamp
Off	UNORM, UNORM_SRGB,YCRCB	Must be enabled with range = RT range or [0,1] (same function)	n/a, state ignored
	SNORM	Must be enabled with range = RT range or [-1,1] (same function)	n/a, state ignored
	FLOAT (except for R11G11B10_FLOAT)	Must be enabled (with any desired range)	n/a, state ignored
	R11G11B10_FLOAT	Must be enabled with either [0,1] or RT range	n/a, state ignored
	UINT, SINT	State ignored, implied clamp to RT range	n/a, state ignored
On (where permitted)	UNORM, UNORM_SRGB	Must be enabled with range = RT range or [0,1] (same function)	Must be enabled with range = RT range or [0,1] (same function)
	SNORM	Must be enabled with range = RT range or [-1,1] (same function)	Must be enabled with range = RT range or [-1,1] (same function)
	FLOAT (except for R11G11B10_FLOAT)	Can be disabled or enabled (with any desired range)	Must be enabled (with any desired range)
	R11G11B10_FLOAT	Can be disabled or enabled (with any desired range)	Must be enabled with either [0,1] or RT range

[Pre-DevSNB]: Note regarding Multiple RenderTargets (MRTs): There is only one set of Pre/Post-Blend Color Clamp state variables, and therefore they apply to all RTs (i.e., for each separate RT-Write DataPort message). If all RTs have the same format, then these controls can be programmed with the same flexibility as if there was only one RT. However, if the RTs can have differing formats, then software must ensure that the shared control settings make sense for each RT format. For example, specifying a pre-blend and post-blend clamp to RT-range will work for any combination of RT formats, while specifying a pre-blend clamp to [-1,1] when using a UNORM+SNORM MRT likely won't produce meaningful results in the UNORM RT.

9.1.6.1.1 Pre-Blend Color Clamping when Blending is Disabled

The clamping of source color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all source color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed.

Programming Notes:

- Given the possibility of writing UNPREDICTABLE values to the Color Buffer, it is expected and highly recommended that, when blending is disabled, software set **Pre-Blend Color Clamp Enable** to ENABLED and select an appropriate **Color Clamp Range**.
- When using SINT or UINT rendertarget surface formats, **Blending must** be DISABLED. The **Pre-Blend Color Clamp Enable** and **Color Clamp Range** fields are ignored, and an implied clamp to the rendertarget surface format is performed.

9.1.6.1.2 Pre-Blend Color Clamping when Blending is Enabled

The clamping of source, destination and constant color components is controlled by **Pre-Blend Color Clamp Enable**. If ENABLED, all these color components are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed on these color components prior to blending.

9.1.7 Color Buffer Blending

The Color Buffer Blending function is used to combine one or two incoming "source" pixel color+alpha values with the "destination" color+alpha read from the corresponding location in a RenderTarget.

Blending is enabled on a global basis by the **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE). If DISABLED, Blending and Post-Blend Clamp functions are disabled for all RenderTargets, and the pixel values (possibly subject to Pre-Blend Clamp) are passed through unchanged.

[Pre-DevSNB]: If the **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE) is ENABLED, then the RenderTarget's **Color Blend Enable** bit (in SURFACE_STATE) is used to determine if Blending is enabled or disabled. Note that each RenderTarget has its own "local" Color Blend Enable state, so in Multi-RenderTarget scenarios some RTs may have Blending enabled and other RTs may have Blending disabled.

DevBW-A,B Errata: The Color Blend Enable bit in SURFACE_STATE is not used, and acts as if it is ENABLED for each RenderTarget. Blending is enabled or disabled only a a global basis by **Color Buffer Blend Enable** state variable (in COLOR_CALC_STATE)

Programming Note:

- Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
- Dual source blending:
  - **[DevBW, DevCL-A]** Not supported
  - **[DevCL-B, DevCTG+]:** The DataPort only supports dual source blending with a SIMD8-style message.
- Only certain surface formats support Color Buffer Blending. Refer to the Surface Format tables in *Sampling Engine*. Blending must be disabled on a RenderTarget if blending is not supported.

The incoming "source" pixel values are modulated by a selected "source" blend factor, and the possibly gamma-decorrected "destination" values are modulated by a "destination" blend factor. These terms are then combined with a "blend function". In general:

src_term = src_blend_factor * src_color

dst_term = dst_blend_factor * dst_color

color output = blend_function(src_term, dst_term)

If there is no alpha value contained in the Color Buffer, a default value of 1.0 is used and, correspondingly, there is no alpha component computed by this function.

[DevCL-B, DevCTG+]: Dual Source Blending: When using "Dual Source" Render Target Write messages, the Source1 pixel color+alpha passed in the message can be selected as a src/dst blend factor.

Table 9-1. In single-source mode, those blend factor selections are invalid. If SRC1 is included in a src/dst blend factor and a DualSource RT Write message is not utilized, results are UNDEFINED. (This reflects the same restriction in APIs, where undefined results are produced if “o1” is not written by a PS – there are no default values defined). **[Pre-DevSNB]**: Also, it is UNDEFINED to utilize a DualSource RT Write message when Blending is disabled.

The blending of the color and alpha components is controlled with two separate (color and alpha) sets of state variables. However, if the **Independent Alpha Blend Enable** state variable in COLOR_CALC_STATE is DISABLED, then the “color” (rather than “alpha”) set of state variables is used for both color and alpha. Note that this is the only use of the **Independent Alpha Blend Enable** state – it does not control whether Blending occurs, only how.

The following table describes the color source and destination blend factors controlled by the **Source [Alpha] Blend Factor** and **Destination [Alpha] Blend Factor** state variables in COLOR_CALC_STATE. Note that the blend factors applied to the R,G,B channels are always controlled by the **Source/Destination Blend Factor**, while the blend factor applied to the alpha channel is controlled either by **Source/Destination Blend Factor** or **Source/Destination Alpha Blend Factor**.

Table 9-1. Color Buffer Blend Color Factors

Blend Factor Selection	Blend Factor Applied for R,G,B,A channels (oN = output from PS to RT#N) (o1 = 2nd output from PS in Dual-Souce mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_ZERO	0.0, 0.0, 0.0, 0.0
BLENDFACTOR_ONE	1.0, 1.0, 1.0, 1.0
BLENDFACTOR_SRC_COLOR	oN.r, oN.g, oN.b, oN.a
BLENDFACTOR_INV_SRC_COLOR	1.0-oN.r, 1.0-oN.g, 1.0-oN.b, 1.0-oN.a
BLENDFACTOR_SRC_ALPHA	oN.a, oN.a, oN.a, oN.a
BLENDFACTOR_INV_SRC_ALPHA	1.0-oN.a, 1.0-oN.a, 1.0-oN.a, 1.0-oN.a
BLENDFACTOR_SRC1_COLOR	o1.r, o1.g, o1.b, o1.a
BLENDFACTOR_INV_SRC1_COLOR	1.0-o1.r, 1.0-o1.g, 1.0-o1.b, 1.0-o1.a
BLENDFACTOR_SRC1_ALPHA	o1.a, o1.a, o1.a, o1.a
BLENDFACTOR_INV_SRC1_ALPHA	1.0-o1.a, 1.0-o1.a, 1.0-o1.a, 1.0-o1.a
BLENDFACTOR_DST_COLOR	rtN.r, rtN.g, rtN.b, rtN.a
BLENDFACTOR_INV_DST_COLOR	1.0-rtN.r, 1.0-rtN.g, 1.0-rtN.b, 1.0-rtN.a
BLENDFACTOR_DST_ALPHA	rtN.a, rtN.a, rtN.a, rtN.a
BLENDFACTOR_INV_DST_ALPHA	1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a, 1.0-rtN.a
BLENDFACTOR_CONST_COLOR	CC.r, CC.g, CC.b, CC.a
BLENDFACTOR_INV_CONST_COLOR	1.0-CC.r, 1.0-CC.g, 1.0-CC.b, 1.0-CC.a
BLENDFACTOR_CONST_ALPHA	CC.a, CC.a, CC.a, CC.a
BLENDFACTOR_INV_CONST_ALPHA	1.0-CC.a, 1.0-CC.a, 1.0-CC.a, 1.0-CC.a
BLENDFACTOR_SRC_ALPHA_SATURATE	f,f,f,1.0 where f = min(1.0 – rtN.a, oN.a)

The following table lists the supported blending operations defined by the **Color Blend Function** state variable and the **Alpha Blend Function** state variable (when in independent alpha blend mode).

Table 9-2. Color Buffer Blend Functions

Blend Function	Operation (for each color component)
BLENDFUNCTION_ADD	SrcColor*SrcFactor + DstColor*DstFactor
BLENDFUNCTION_SUBTRACT	SrcColor*SrcFactor - DstColor*DstFactor
BLENDFUNCTION_REVERSE_SUBTRACT	DstColor*DstFactor - SrcColor*SrcFactor
BLENDFUNCTION_MIN	min (SrcColor*SrcFactor, DstColor*DstFactor) **Programming Note:** This is a superset of the OpenGL "min" function.
BLENDFUNCTION_MAX	max (SrcColor*SrcFactor, DstColor*DstFactor) **Programming Note:** This is a superset of the OpenGL "max" function.

9.1.7.1 3DST ATE_CONSTANT COLOR [Pre-DevSNB]

3DSTATE_CONSTANT_COLOR

Project:	[Pre-DevSNB]	Length Bias:	2

The 3DSTATE_CONSTANT_COLOR command is used to specify the Constant Color used in Color Buffer Blending. It is a non-pipelined command.

For **[DevSNB+]**, these paramaters are included in the COLOR_CALC_STATE structure and this command is not supported.

DWord	Bit	Description
0	31:29	**Command Type** Default Value: 3h GFXPIPE Format: OpCode
	28:27	**Command SubType** Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	**3D Command Opcode** Default Value: 1h 3DSTATE_NONPIPELINED Format: OpCode
	23:16	**3D Command Sub Opcode** Default Value: 01h 3DSTATE_CONSTANT_COLOR Format: OpCode
	15:8	**Reserved** Project: All Format: MBZ
	7:0	**DWord Length** Default Value: 3h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All

3DSTATE_CONSTANT_COLOR		
1	31:0	**Blend Constant Color Red** Project: All Format: IEEE_Float FormatDesc This field specifies the Red channel of the Constant Color used in Color Buffer Blending.
2	31:0	**Blend Constant Color Green** Project: All Format: IEEE_Float FormatDesc This field specifies the Green channel of the Constant Color used in Color Buffer Blending.
3	31:0	**Blend Constant Color Blue** Project: All Format: IEEE_Float FormatDesc This field specifies the Blue channel of the Constant Color used in Color Buffer Blending.
4	31:0	**Blend Constant Color Alpha** Project: All Format: IEEE_Float FormatDesc This field specifies the Alpha channel of the Constant Color used in Color Buffer Blending.

9.1.8 Post-Blend Color Clamping

(See *Pre-Blend Color Clamping* above for a summary table regarding clamping)

Post-Blend Color clamping is available only if Blending is enabled.

If Blending is enabled, the clamping of blending output color components is controlled by **Post-Blend Color Clamp Enable**. If ENABLED, the color components output from blending are clamped to the range specified by **Color Clamp Range**. If DISABLED, no clamping is performed at this point.

Regardless of the setting of **Post-Blend Color Clamp Enable**, when Blending is enabled color components will be automatically clamped to (at least) the rendertarget surface format range at this stage of the pipeline.

9.1.9 Color Quantization

[This is considered an implementation-specific topic, covered in the detailed hardware design documents]

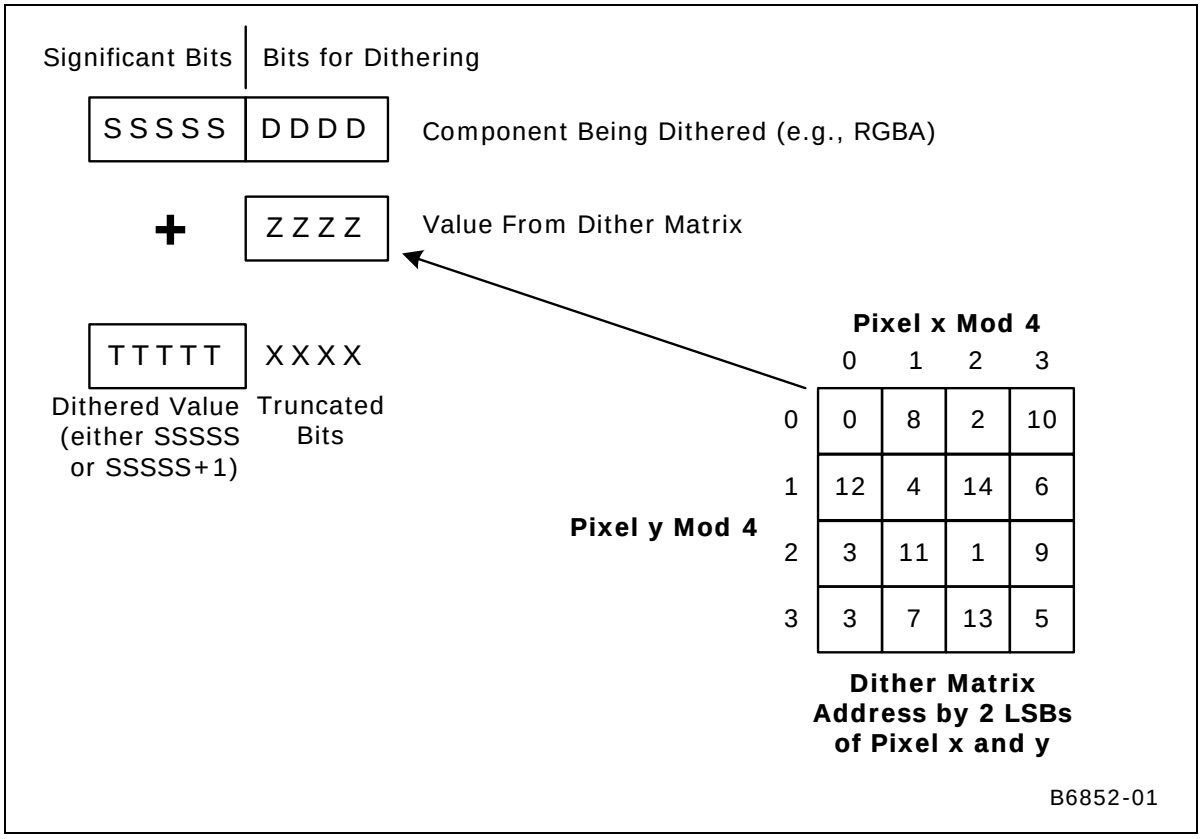
9.1.10 Dithering

Dithering is used to give the illusion of a higher resolution when using low-bpp channels in color buffers (e.g., with 16bpp color buffer). By carefully choosing an arrangement of lower resolution colors, colors otherwise not representable can be approximated, especially when seen at a distance where the viewer's eyes will average adjacent pixel colors. Color dithering tends to diffuse the sharp color bands seen on smooth-shaded objects.

A four-bit dither value is obtained from a 4x4 Dither Constant matrix depending on the pixel's X and Y screen coordinate. The pixel's X and Y screen coordinates are first offset by the **Dither Offset X** and **Dither Offset Y** state variables (these offsets are used to provide window-relative dithering). Then the two LSBs of the pixel's screen X coordinate are used to address a column in the dither matrix, and the two LSBs of the pixel's screen Y coordinate are used to address a row. This way, the matrix repeats every four pixels in both directions.

The value obtained is appropriately shifted to align with (what would be otherwise) truncated bits of the component being dithered. It is then added with the component and the result is truncated to the bit depth of the component given the color buffer format.

Figure 9-2. Dithering Process (5-Bit Example)



9.1.11 Logic Ops

The Logic Ops function is used to combine the incoming “source” pixel color/alpha values with the corresponding “destination” color/alpha contained in the ColorBuffer, using a logic function.

The Logic Op function is enabled by the **LogicOp Enable** state variable. If DISABLED, this function is ignored and the incoming pixel values are passed through unchanged.

Programming Note:

- Color Buffer Blending and Logic Ops must not be enabled simultaneously, or behavior is UNDEFINED.
- Logic Ops are only supported on *_UNORM surfaces (excluding _SRGB variants), otherwise Logic Ops must be DISABLED.
- **DevBW-A,B Errata:** Logic Ops are not supported on 16-bit per channel UNORM surfaces.

The following table lists the supported logic ops. The logic op is selected using the **Logic Op Function** field in COLOR_CALC_STATE.

Table 9-3. Logic Ops

LogicOp Function	Definition (S=Source, D=Destination)
LOGICOP_CLEAR	all 0's
LOGICOP_NOR	NOT (S OR D)
LOGICOP_AND_INVERTED	(NOT S) AND D
LOGICOP_COPY_INVERTED	NOT S
LOGICOP_AND_REVERSE	S AND NOT D
LOGICOP_INVERT	NOT D
LOGICOP_XOR	S XOR D
LOGICOP_NAND	NOT (S AND D)
LOGICOP_AND	S AND D
LOGICOP_EQUIV	NOT (S XOR D)
LOGICOP_NOOP	D
LOGICOP_OR_INVERTED	(NOT S) OR D
LOGICOP_COPY	S
LOGICOP_OR_REVERSE	S OR NOT D
LOGICOP_OR	S OR D
LOGICOP_SET	all 1's

9.1.12 Buffer Update

The Buffer Update function is responsible for updating the pixel's Stencil, Depth and Color Buffer contents based upon the results of the Stencil and Depth Test functions. Note that Kill Pixel and/or Alpha Test functions may have already discarded the pixel by this point.

9.1.12.1 Stencil Buffer Updates

If and only if stencil testing is enabled, the Stencil Buffer is updated according to the **Stencil Fail Op**, **Stencil Pass Depth Fail Op**, and **Stencil Pass Depth Pass Op** state (or their backface counterparts if **Double Sided Stencil Enable** is ENABLED and the pixel is from a back-facing object) and the results of the Stencil Test and Depth Test functions.

Stencil Fail Op and **Backface Stencil Fail Op** specify how/if the stencil buffer is modified if the stencil test fails. **Stencil Pass Depth Fail Op** and **Backface Stencil Pass Depth Fail Op** specify how/if the stencil buffer is modified if the stencil test passes but the depth test fails. **Stencil Pass Depth Pass Op** and **Backface Stencil Pass Depth Pass Op** specify how/if the stencil buffer is modified if both the stencil and depth tests pass. The operations (on the stencil buffer) that are to be performed under one of these (mutually exclusive) conditions is summarized in the following table.

Table 9-4. Stencil Buffer Operations

Stencil Operation	Description
STENCILOP_KEEP	Do not modify the stencil buffer
STENCILOP_ZERO	Store a 0
STENCILOP_REPLACE	Store the *StencilTestReference* reference value
STENCILOP_INCRSAT	Saturating increment (clamp to max value)
STENCILOP_DECRSAT	Saturating decrement (clamp to 0)
STENCILOP_INCR	Increment (possible wrap around to 0)
STENCILOP_DECR	Decrement (possible wrap to max value)
STENCILOP_INVERT	Logically invert the stencil value

Any and all writes to the stencil portion of the depth buffer are enabled by the **Stencil Buffer Write Enable** state variable.

When writes are enabled, the **Stencil Buffer Write Mask** and **Backface Stencil Buffer Write Mask** state variables provide an 8-bit mask that selects which bits of the stencil write value are modified. Masked-off bits (i.e., mask bit == 0) are left unmodified in the Stencil Buffer.

Programming Notes:

- If the Depth Buffer does **not** have a surface format of D32_FLOAT_S8X24_UINT or D24_UNORM_S8_UINT, **Stencil Buffer Write Enable** must be DISABLED.
- The Stencil Buffer can be written even if depth buffer writes are disabled via **Depth Buffer Write Enable**.

9.1.12.2 Depth Buffer Updates

Any and all writes to the Depth Buffer are enabled by the **Depth Buffer Write Enable** state variable. If there is no Depth Buffer, writes must be explicitly disabled with this state variable, or operation is UNDEFINED.

If depth testing is disabled or the depth test passed, the incoming pixel's depth value is written to the Depth Buffer. If depth testing is enabled and the depth test failed, the pixel is discarded – with no modification to the Depth or Color Buffers (though the Stencil Buffer may have been modified).

9.1.12.3 Color Gamma Correction

Computed RGB (not A) channels can be gamma-corrected prior to update of the Color Buffer.

This function is automatically invoked whenever the destination surface (render target) has an SRGB format (see surface formats in *Sampling Engine*). For these surfaces, the computed RGB values are converted from gamma=1.0 space to gamma=2.4 space by applying a ^(2.4) exponential function.

9.1.12.4 Color Buffer Updates

Finally, if the pixel has not been discarded by this point, the incoming pixel color is written into the Color Buffer. The **Surface Format** of the color buffer indicates which channel(s) are written (e.g., R8G8_UNORM are written with the Red and Green channels only). The **Color Buffer Component Write Disables** from the Color Buffer's SURFACE_STATE provide an independent write disable for each channel of the Color Buffer.

9.2 Pixel Pipeline State Summary

9.2.1 COLOR_ CALC_STATE

9.2.2 CC_VI EWPORT

CC_VIEWPORT		
Project: All		
The viewport state is stored as an array of up to 16 elements, each of which contains the DWords described here. The start of each element is spaced 2 DWords apart. The first element of the viewport state array is aligned to a 32-byte boundary.		
DWord	**Bit**	**Description**
0	31:0	**Minimum Depth** Project: All Format: IEEE_Float FormatDesc Indicates the minimum depth. The interpolated or computed depth is clamped to this value prior to the depth test.
1	31:0	**Maximum Depth** Project: All Format: IEEE_Float FormatDesc Indicates the maximum depth. The interpolated or computed depth is clamped to this value prior to the depth test.

9.3 Other Pixel Pipeline Functions

9.3.1 Statistics Gathering

[Pre-DevSNB]: If **Statistics Enable** is set in WM_STATE *and* in CC_STATE, the PS_DEPTH_COUNT register (see Memory Interface Registers in Volume Ia, *GPU*) will be incremented once for each pixel that passes the depth, stencil and alpha tests. Note that each of these tests is treated as passing if disabled. This count is accurate regardless of whether **Early Depth Test Enable** is set. In order to obtain the value from this register at a deterministic place in the primitive stream without flushing the pipeline, however, the PIPE_CONTROL command must be used. See the *3D Pipeline* chapter in this volume for details on PIPE_CONTROL.

command must be used. See the *3D Pipeline* chapter in this volume for details on PIPE_CONTROL.

[DevBW-A] Errata BWT008: PS_DEPTH_COUNT cannot be accurately read using PIPE_CONTROL. Attempting to do so will result in an UNDEFINED value being written out to the PIPE_CONTROL target address.