AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 1.0	PAGE [PAGE] of [

Author: Todd Martin, Mangesh Nijasure

ISSUED TO:	COPY NO.

WD/IA/VGT

Micro-Architecture Specification

Rev 1.0 – Last Edit: [SAVEDATE \@ "d-MMM-yy" * MERGEFORMAT]5

THIS DOCUMENT CONTAINS (INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF AMD THROUGH UNLICENSED USE OR UNAUTHORIZED DISCLOSURE.

1. **Preserve this document's integrity:**

   [SYMBOL 222 \f "Symbol" \s 8 \h] Do not reproduce any portions of it.

   [SYMBOL 222 \f "Symbol" \s 8 \h] Do not separate any pages from this cover.

2. **This document is issued to you alone. Do not transfer it to or share it with another person, even within your organization.**
3. **Store this document in a locked cabinet accessible only by authorized users. Do not leave it unattended.**
4. **When you no longer need this document, return it to AMD. Please do not discard it.**

Revision History

Date	Revision	Description

Table of Contents[TOC \O "1-6" \T "FIGCAPTION,3,FIGCAPTIONTOP,3,FIGCAPTIONBOTTOM,3,CAPTION,3"]

1 Introduction

This document contains a description of the features and hardware implementation of the WD, IA, and the VGT blocks and how they fit into the overall graphics architecture.

1.1 Open Issues

There are no known open issues for the WD, IA, or VGT.

1.2 Scope

This document details the feature requirements and the hardware implementation for the WD, IA, and VGT blocks.

1.3 Reference

No external documents other than those explicitly linked in this specification are necessary to understand the material presented here. This micro architecture specification is self sufficient in describing the design and features of the WD, IA, and VGT blocks.

1.4 Definitions / glossary of terms

- ***WD*** – Work Distributer, receives all the draw commands and breaks them up into work groups which are sent to one or more IA units
- ***IA*** – Input Assembler, receives work groups and breaks them up into prim groups for the VGT. Fetches indices from memory.
- ***VGT*** – Vertex Geometry Tessellator, this is the main block responsible for supporting all DX and OGL draw packets
- ***THREAD*** – A thread is a single entity in a wavefront, this can be vertices, primitives etc
- ***WAVEFRONT*** – A group of threads that execute in SIMD fashion.
- ***SE*** – Shader Engine
- ***PA*** – Primitive Assembler
- ***LS*** – Local data Shader
- ***HS*** – Hull Shader
- ***DS*** – Domain Shader
- ***ES*** – Export Shader
- ***GS*** – Geometry Shader
- ***VS*** – Vertex Shader
- ***CS*** – Compute Shader
- ***EOP*** – End Of Packet
- ***EOPG*** – End Of Primgroup

1.5 Top Level Diagram

This diagram shows a 4 shader engine configuration.

[EMBED Visio.Drawing.11]

2 Delta Requirements

All delta features have been folded into this spec.

3 Features / Functionality

3.1 Overview

The WD, IA, and VGT are responsible for creating thread data and wavefront assignment for nearly all the graphics shader stages in use by the graphics core. These are Vertex Shader (VS), Export Shader (ES), Geometry Shader (GS), Local Data Shader (LS), and Hull Shader (HS). The remaining shader type, Pixel Shader (PS), is not controlled directly by the VGT, but the VGT does provide primitive information to the Primitive Assembly (PA) block which begins a process of clipping and sends clipped primitives to a Scan Converter (SC) that produces Pixel Shader (PS) data and wavefronts.

By providing these workloads in a deterministic, state-controlled order, the WD/IA/VGT enables and controls the various graphics pipelines (from a simple VS->PS pipeline, to a complicated pipeline such as LS->HS->TESS_FIXED_FUNCTION->ES(as DS)->GS->VS). Primarily the WD/IA/VGT accomplishes this by decomposing input packets which yield all of the shader types. In addition to the controlling shader stages, the VGT in particular, also provides a fixed function tessellation stage of the graphics pipeline. The register VGT_SHADER_STAGES_EN indicates which shader stages are enabled for a given Draw Initiator.

There is one WD block per chip and one VGT per Shader Engine. A single IA is paired with two VGT blocks so a two SE chip will have one IA while a four SE chip has 2 IAs. A single VGT has a throughput of one primitive per cycle so the number of VGTs present directly controls the maximum geometry throughput of the system.

The WD/IA/VGT is responsible for:

COMPUTE:

- Compute support has completely moved to the CP.

GRAPHICS:

- Receiving graphics state, draw requests, and synchronization events, from the GRBM bus.
- Fetching, from memory, the individual vertex indices (16 or 32 bit pointers to vertex data) requested by a draw call.
- Grouping the indices into primitives such as lines, triangles, or patches.
- Determining index reuse within a fixed window of indices. This avoids redundant vertex shading.
- Providing primitive information to the Primitive Assembler (PA) block.
- Providing statistics and synchronizing events to the Command Processor (CP).
- Alternate graphics workloads to shader engines.
- Support legacy tessellation mode (does not use the LS/HS/DS shader stages, and has a different fixed function tessellation algorithm)
- Arbitrate (at a packet boundary) between high and normal priority draw calls
- Independent pipeline reset such that work assigned a given VMID is stopped as quick as possible.
- Front end harvesting including deactivation of individual VGTs and associated IA.
- Order data for streamout.

- If Geometry Shading is enabled
    - Generate ES wavefronts/vertices/GS primitives and send them to the SPI
    - Upon completion of ES, generate GS wavefronts and send them to the SPI
    - Receive Geometry information output from the GS

- Upon completion of GS, generate VS wavefronts/vertices (including streamout data) and send them to the SPI. In this situation the VS is also known as the Copy Shader since it is copying data from the GSVS ring buffer to the position buffer and parameter cache.
- Generate Primitive information and send it to the PA
- ES/GS output data is either all off chip or all on chip

- If Tessellation is enabled
  - Generate LS wavefronts/vertices and send them to the SPI
  - Upon completion of LS, generate HS wavefronts and send them to the SPI
  - Upon completion of HS, retrieve tessellation factors from memory, and execute the fixed function tessellator stage.
  - Create DS wavefronts/data from output of tessellator. Send the DS wavefronts to the SPI as either ES wavefronts (Geometry Shader enabled) or as Vertex Shader wavefronts (Geometry shader disabled)
  - Optionally (if Geometry Shader enabled) create GS wavefronts/data and send them to the SPI
  - Generate Primitive information and send it to the PA

3.2 Data Flow based on SHADER_STAGES_EN programming

Shader_en mode	VS	HS	DS	GS	Hardware data flow
F	on	on	on	on	LS->HS->TE->ES->GS->VS
E	on	on	on	off	API VS runs as LS. LS->HS->TE->VS
D	on	on	off	on	Not valid
C	on	on	off	off	Not valid
B	on	off	on	on	Not valid
A	on	off	on	off	Not valid
9	on	off	off	on	API VS runs as the ES. ES->GS->VS
8	on	off	off	off	VS->PS
7	off	on	on	on	Not possible VS has to be on
6	off	on	on	off	Not possible VS has to be on
5	off	on	off	on	Not possible VS has to be on
4	off	on	off	off	Not possible VS has to be on
3	off	off	on	on	Not possible VS has to be on
2	off	off	on	off	Not possible VS has to be on
1	off	off	off	on	Not possible VS has to be on
0	off	off	off	off	Not possible VS has to be on

3.3 Distribution of work amongst shader engines.

If there is one IA the WD passes through draw calls, however if there are two IA's the WD breaks draws into work groups which are twice the size of a prim group. The IA sends an entire prim group to a VGT before switching to the next VGT. A end of prim group (eopg) signal follows each prim group and the SC looks for these to know when to switch input FIFOs.

The WD will discard any draw call that that contains 0 indices or a primitive type of DI_PT_NONE. Beginning in gfx8, the WD will also discard any draw call that sets the number of instances to 0. If there is one IA, the draw will be dropped and nothing will be sent down the pipe, but if there are two IA's, the WD will send a null eop down the pipe and toggle the IA that will receive work next.

Below is an example of processing primitive groups from IA to SCs.

The Input Assembler (IA) block processes a serial stream of primitives. The bolded P's represent primitives that will be sent to SE0.

p, p, p (eopg), p, p, p (eopg), **p** (eop/eopg), event, p, p (eop/eopg), event, eop, event, **p, p** (eopg), p, eop/eopg, event

Four packets processed by VGT modules are shown below with different colors. This represents data sent from the IA to VGT's 0 and 1. PA_0 will receive the same sequence as VGT_0 and PA_1 will receive the same sequence as VGT_1. After reset processing starts with VGT module 0.

Eopg marks the end of the prim group and this is what tells the IA block to switch SE's. Eopg is only sent to the active SE.

Eop is sent to both SE's in unordered mode. Otherwise the PA drops the eop that isn't accompanied by eopg.

Each line represents a point in time so items on the same line occur simultaneously.
Read the data flow from top to bottom.

VGT_0	VGT_1
P	
P	
P/eopg	
	P
	P
	P/eopg
P/eopg/eop	eop
event	event
------------------------------	------------------------------
	P
eop	P/eopg/eop
event	event
------------------------------	------------------------------
event	event
------------------------------	------------------------------
P	
P/eopg	
	P
eop	eopg/eop
event	event

The table below shows how PA outputs are loaded into SCs. Some of the primitives which can span scan windows of two SCs are loaded into FIFOs of both SCs.

The processing of FIFOs of each SCs are done independently for each SC. The diagram below shows order of processing within FIFOs for a given SC.

The SC switches the fifo it's reading from after reading out a eopg. The SC synchronizes FIFOs when processing an event or eop though when ordered mode (default) is used the PA will only send eop if it is accompanied by a eopg.

Read the data flow from top to bottom. Each line shows what is being read out by the SC during a clock cycle.

SC_0		**SC_1**	
FIFO_0	**FIFO_1**	**FIFO_0**	**FIFO_1**
P		eopg	
P			P
P/eopg			P
	eopg		P/eopg
P/eopg/eop	eop	eopg/eop	eop
event	event	event	event

---

eop	P'/eopg/eop		P
event	event	eop	P''/eopg/eop
		event	event

---

event	event	event	event

P		eopg	
P/eopg			P
eop	eopg/eop	eop	eopg/eop
event	event	event	event

The following table shows what each SC sees after reading from the input FIFOs.
Read the data flow from top to bottom. Packets are separated for clarity.

SC_0	SC_1
P	P
P	P
P	P
P/eop	eop
event	event
-------------------------	-------------------------
P'/eop	P

event	P"/eop event
event	event
P P eop event	P eop event

3.4 Vertex Reuse

The intent of the vertex reuse determination is to efficiently use the Vertex Shader by preventing the Vertex Shader from processing the same vertex multiple times if that vertex is used in multiple primitives that occur relatively close together in the input stream. The VGT must detect vertex reuse within the previous 30 (or less) vertex indices. In other words, vertex reuse is determined by the redundant occurrence of an external vertex index within a limited scope of the external vertex index list. If a "hit" is detected for a given vertex index, that vertex index is not resubmitted for vertex processing.

Reuse checks are performed in multiple sub-blocks in the VGT. Here is a list of shader stages and where vertex reuse occurs.

VS -> PS: Vertex Reuse Block performs the reuse. If Streamout is enabled reuse is automatically disabled.

ES -> GS -> VS -> PS: The GS Reuse Check Module performs reuse checks to remove redundant vertices from ES wavefronts. GS prims output strips, but there is no reuse between the strips. If Streamout is enabled reuse is automatically disabled prior to the VS and the strips are converted to lists.

LS -> HS -> VS -> PS: The tessellator performs reuse prior to the VS stage. There is no reuse between patches, only between primitives output by a single patch. If Streamout is enabled reuse is automatically disabled.

LS -> HS -> ES -> GS -> VS -> PS: The tessellator performs reuse prior to the ES stage. GS prims output strips, but there is no reuse between the strips. If Streamout is enabled reuse is automatically disabled prior to the VS and the strips are converted to lists.

3.4.1 Bank Conflict Detection

With the increase to a reuse depth from 16 to 30 in gfx8, it became possible for there to be bank conflicts in the parameter cache. In order to simplify logic in the SPI and the Parameter Cache, there will be bank conflict prevention code added to some of the VGT reuse checkers. Any intra-prim relative indices that would cause bank conflicts in the parameter cache will not be allowed.

The diagram below shows a conflict in the parameter cache. If the indices so far have been 0, 1, ... ,21 they are stored in the parameter cache as follows. Each of the columns shown is a bank, there are 16 banks in the PC.

									22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

If the new incoming primitive (a triangle) has the indices 22, 3 and 6. The indices 22 and 6 will be fetched from the same bank and will cause a conflict.

The new conflict detection code will eliminate this condition by replicating the index 6 as follows

								6	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

This produces more unique vertices than ideal but it eliminates the need for complex bank conflict detection in the SPI and the PC.

Indices from **different** primitives are allowed to request indices from the same bank, this conflict checking is handled downstream. The VGT will only be responsible for eliminating any intra-prim conflicts.

This conflict detection step is not necessary in the GS RCM or in the TE11 when the GS is enabled. This is because the primitives at these pipeline stages do not go to the parameter cache and will not cause bank conflicts when fetched

The VGT_GS_VERTEX_REUSE register is now deprecated. The VGT_VERTEX_REUSE_BLOCK_CNTL register now controls reuse depth for all the reuse buffers (DX9, GS and TE11)

Setting VGT_REUSE_OFF.REUSE_OFF turns off reuse in all blocks. This did not turn reuse off in the GS block earlier.

To support this increased reuse depth, reuse is now turned off for any degenerate primitives (any primitive with repeated indices). This is an implementation level detail to save schedule.

3.5 Dealloc Distance and Reuse Depth

The shader always writes 16 vertex parameters. Therefore dealloc_distance and reuse depth is always set to 16 (points) or 14 (triangles). This also shows that for legacy (non-DX11) tessellation we can setup HOS_REUSE_DEPTH to 16. The following changes were done to remove dealloc_slot issue independent of quad_pipes.

1. The VGT submits 64 vertices per wave unless half pack is switched on.
```
if (half_packed flag)
    create 32 verts per vector
else
    create 64 verts per vector
```
2. De-allocate distance will always be 16 and reuse can always be 14 unless driver wants to limit it to be less
3. The VGT change to create
    a. 1 New Flag per vertex vector attached to the first primitive containing the first vert of a new vertex vector (Same as today)
    b. 1 De-allocate signal per Vertex Vector submitted instead of 4.

   i. This will be attached to the primitive holding a vertex that moves past the reuse window of the previous vector
   ii. It will be variable base on the resulting number of vertices per vector set up in number 1.
      1. vert 80 and every 64 thereafter
c. 1 last signal that is sent in the msb of de-allocate signal
   i. This will be attached to the primitive containing the first reference to a new specific vert for each vector depending on the vector size
      1. vert 61 and every 64 thereafter
   ii. This signal is only for SC usage
4. The largest actual de-allocate count the VGT will send now will be 3
5. The scan converter change his partial vector submit circuit to emit a partial vector when he has a de-allocate count of non zero and gets a last flag. This will prevent hang conditions when there is a lot of culling going on. The Scan converter will also remove the previous partial submit for this reason and add asserts for the occurrence of the partial submit and a fail assert if de-allocate is every larger than 2.
6. The PA changes to pack parameters into the parameter cache. It will use the bad pipe flags and number of parameters to determine next offsets.
7. SQ allocates and de-allocate based on num_quad_pipes and number of parameters. It will do a special case limiting when num_quad+pipes > 2 and num_parameters >= 16, it will actually act like two quad pipes. So

```
if (num_quad_pipes >2  && half_packed flag)
      alloc_amount = num_paramters * 2
else
      alloc_amount num_quad_pipes * num_parameters
```

3.6 VertexID

VertexID is a 32 bit unsigned integer value created by the VGT and loaded into a VGPR for the API vertex shader. It is unique per vertex though each VGT maintains its own count. This feature is not required by DX or OpenGL. The count is reset by a RESET_VTX_CNT event.

3.7 PrimitiveID

PrimitiveID is a 32 bit unsigned integer value created by the VGT and loaded into a VGPR. The register VGT_PRIMITIVEID_RESET specifies the reset value to be used at the beginning of each instance. Typically it's programmed to 0. For special GS modes like scenario A and B the VGT_PRIMITIVEID_EN register specifies that the primitive ID value should be loaded into a VGPR at the expense of an instance step rate value.

PrimitiveID is automatically available to the HS, DS and GS. If it's needed in the PS it must be passed as a parameter. It is expected that the situations where primitiveID is used by the PS but there is no GS instantiated are rare. To avoid having the hardware have to pipe the full 32-bit primitiveID through hundreds of clocks of pipeline, the driver will be expected to change the VS into a GS_A, which is basically a VS which gets primitiveID on the input, and output the primitiveID on the expected vector/component where the PS expects it. The only other unique processing associated with a GS_A is that the VGT must guarantee that the leading vertex is unique (i.e. does not hit in the vertex reuse cache). This is required so that unique data for the primitive (i.e. primitiveID) is available for constant interpolation for the primitive.

3.8 InstanceID

InstanceID is a 32 bit unsigned integer value created by the VGT and loaded into a VGPR. It starts at 0 for all of the vertices of the first instance, and increments thereafter for each instance. It should also be 0 for non-instance draw calls.

The value will be supplied to the API VS and is available to the GS and PS. The path to the VS is the only hardware-dedicated path for instanceID as the driver is expected to create a VS (which would pass it along if necessary) if there is no VS instantiated.

The VGT will also supply up to two step-rate divide values to assist the fetch shader for cases with a small number of unique step-rates. In case the required number step rates exceeds what is supported by VGT, the remaining instanceID/step-rate will be calculated by the fetch shader. A vertex wavefront may consist of vertices with different instanceID's.

In a multiple VGT system, the end of instance flag needs to be propagated to all VGTs in order to correctly increment the instanceID and reset the reuse buffer. In variants with more than one IA, the WD sends a null_eoi to the 'other' IA which propagates the flag to the VGTs connected to it. When the entire instance fits on one IA, the null_eois sent to the other side can add up as dead cycles and show up as performance glitches.

If the entire draw is smaller than a primgroup, the null_eois are suppressed and instance_id is still handled correctly. This is done by adding an interface bit from the WD to the IA indicating that the draw was a candidate for optimization. Any null_eois that are not eops will be discarded gracefully and will not exit the IA

3.9 Reset Index

A reset index is a special index value that signifies the end of a primitive. It's typically used to break strips, but may be enabled for lists. Reset index is not supported with patches. Reset index checking occurs in the IA and it's enabled by setting the VGT_MULTI_PRIM_IB_RESET_EN register. The index value to check for is specified in VGT_MULTI_PRIM_IB_RESET_INDX.

Enabling reset index limits performance for designs that have greater than 2x prim rate (2 or more IAs) as it requires WD_SWITCH_ON_EOP to be set. For this reason we recommend our developer relations personnel evangelize using lists instead of strips with reset indices.

Partial primitives that result from a reset index or at the end of a packet are silently discarded.

Prior to gfx8, drivers modified the value of the reset index check register VGT_MULTI_PRIM_IB_RESET_INDX based on the index type (8, 16 or 32 bit). In order to alleviate the software validation that is performed, in gfx8 and later projects the hardware masks out the register bits depending on the number of bits in the current index.

For 16 bit indices, earlier the driver needed to program 0x0000VVVV, where VVVV is the reset index.

Now the driver can use 0xXXXXVVVV, where XXXX are don't care

Besides the performance implications other caveats to using reset index are:
1. Line stipple will not produce the correct visual result with this mode. The line stipple pattern will not reset between strips (which it would if the strips were sent with separate VGT_DRAW_INITIATOR commands).
2. Edge flags will not be correct for the prim order VGT_GRP_POLYGON. This will have a visual impact in OpenGL for this primitive order if POLY_MODE is set to LINES or POINTS. (This applies mostly to the OpenGL polygon primitive.)

3.10 Provoking Vertex

If flat shading is enabled for a primitive, then the provoking vertex is the vertex whose color is used to shade the entire primitive. OpenGL and Direct3D differ (for most primitive types) in their respective selections of the provoking vertex. The VGT will be designed so that the OpenGL primitives will always program the provoking vertex select to "last vertex" and the Direct3D primitives will always program the provoking vertex select to "first vertex".

OpenGL Specification
The following table is based directly on table 4-2 from OpenGL Programming Guide, Second Edition. (The version in the OpenGL spec counts vertices and primitives starting at 1, whereas this version counts vertices and primitives starting at 0. After swapping for specified vertex order within the primitive, the provoking vertex is the last vertex in the primitive with the exception of the polygon primitive where the first vertex is the provoking vertex.

Table 1. OpenGL Provoking Vertex.

Type of Polygon	OpenGL Vertex Used to Select the Color for the ith Polygon	Direct3D Vertex Used to Select the Color for the ith Polygon
single polygon	0	N/A
triangle strip	i+2 (last vtx in tri)	i (first vtx in tri)
triangle fan	i+2 (last vtx in tri)	i (first vtx in tri)
independent triangle (triangle list)	3i+2 (last vtx in tri)	3i (first vtx in tri)
quad strip[1]	2i+3 (next-to-last vtx in quad)	N/A — 2i (first vtx in quad)
independent quad	4i+3 (last vtx in quad)	N/A — 4i (first vtx in quad)

[1] For OpenGL quad strips, the provoking vertex is the last vertex in the vertex buffer that forms the primitive; however, it is the next-to-last vertex the primitive using the primitive-relative vertex order. For example, if the vertex buffer contains V0, V1, V2, and V3 in that order, then the first quad primitive from that strip will have the vertex order V0, V1, V3, V2. The provoking vertex for the quad is V3. See [REF _Ref687713 \r \h] for more detail.

3.10.1 Primitive Vertex Ordering and Provoking Vertex Summary

Table 2. Primitive Vertex Order and Provoking Vertex Summary

Underlined Vertex is the Provoking Vertex for Flat Shading		
Primitive Type	Direct3D	OpenGL
Point	V0 V1	V0 V1
Line List	V0, V1 V2, V3	V0, V1 V2, V3
Line Strip	V0, V1 V1, V2	V0, V1 V1, V2
Line Loop	V0, V1 V1, V2 V2, V0 <= created by VGT	V0, V1 V1, V2 V2, V0 <= created by VGT
Tri List	V0, V1, V2 V3, V4, V5	V0, V1, V2 V3, V4, V5
Tri Strip	V0, V1, V2 V1, V3, V2 <= VGT swaps last two	V0, V1, V2 V2, V1, V3 <= VGT swaps first two
Tri Fan	V1, V2, V0 <= VGT rotates first to last V2, V3, V0 <= VGT rotates first to last	V0, V1, V2 V0, V2, V3
Quad List (Native)	Does not exist — assumed V0, V1, V2, V3 V4, V5, V6, V7	V0, V1, V2, V3 V4, V5, V6, V7
Quad List (Decomposed)	Does not exist — assumed V0, V1, V2 and V0, V2, V3 V4, V5, V6 and V4, V6, V7	V0, V1, V3 and V1, V2, V3 V4, V5, V7 and V5, V6, V7
Quad Strip (Native)	Does not exist — assumed V0, V1, V3, V2 <= VGT swaps last two V2, V3, V5, V4 <= VGT swaps last two V4, V5, V7, V6 <= VGT swaps last two	V0, V1, V3, V2 <= VGT swaps last two V2, V3, V5, V4 <= VGT swaps last two V4, V5, V7, V6 <= VGT swaps last two
Quad Strip (Decomposed)	Does not exist — assumed V0, V1, V3 and V0, V3, V2 V2, V3, V5 and V2, V5, V4 V4, V5, V7 and V4, V7, V6	V0, V1, V3 and V1, V2, V3 V2, V3, V5 and V3, V4, V5 V4, V5, V7 and V5, V6, V7
Polygon (Decomposed)	Does not exist — assumed V0, V1, V2, and V0, V2, V3 and V0, V3, V4 etc…	V1, V2, V0 <= VGT rotates first to last V2, V3, V0 <= VGT rotates first to last V3, V4, V0 <= VGT rotates first to last

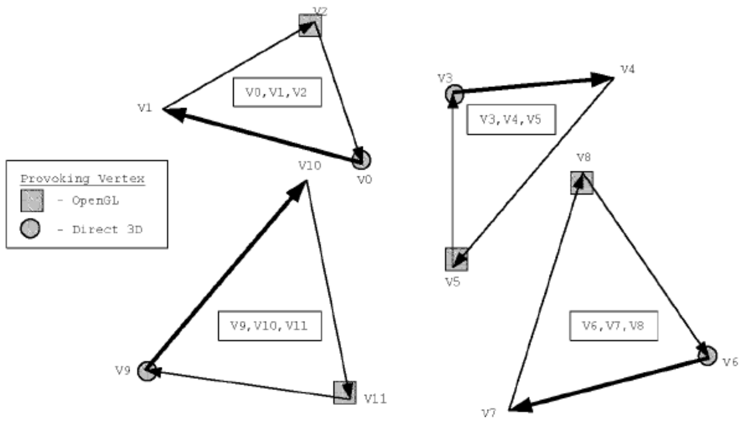
Direct X Specification
The DirectX 8.0 documentation states "When flat shading is enabled, the system shades the triangle with the color from its first vertex." There is no direct mention of flat shading lines, but the VGT design assumes that lines also use the first vertex in each line segment as the provoking vertex.

3.11 Primitive Types

3.11.1 Triangle List

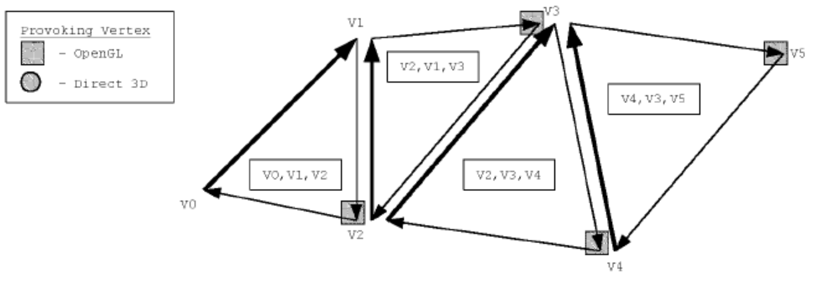
The first edge in each triangle is a bold line. For OpenGL, the last vertex (shown with a square box in [REF _Ref685804 \r \h * MERGEFORMAT]) is used as the provoking vertex. For Direct3D, the first vertex (shown in a circle in [REF _Ref685804 \r \h * MERGEFORMAT]) in each triangle in a triangle list is the provoking vertex.

OpenGL and D3D order.



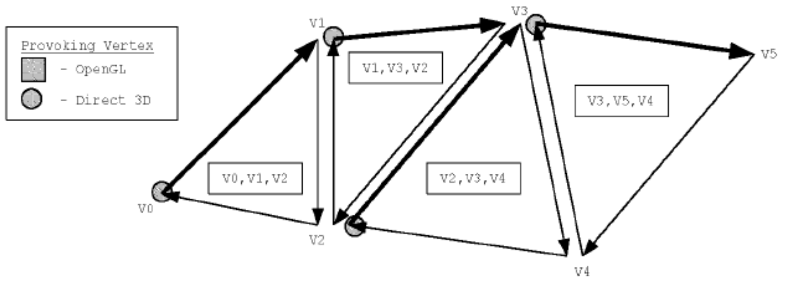
3.11.2 Triangle Strip

The first edge in each triangle is a bold line. Note for OpenGL, only the last vertex (shown with a square box in [REF _Ref685843 \r \h * MERGEFORMAT]) in each triangle progresses in a series (V2, V3, V4, etc). For OpenGL, the last vertex is used as the provoking vertex.



OpenGL order

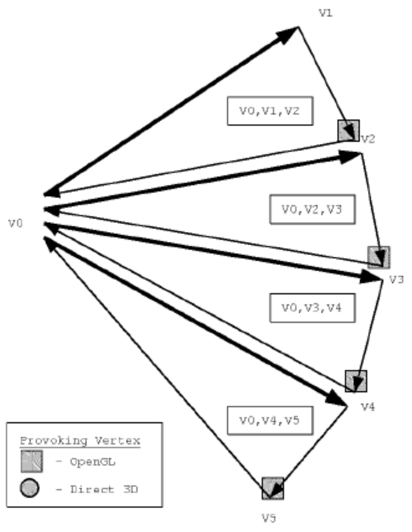
The first edge in each triangle is a bold line. Note for Direct3D, only the first vertex (shown in a circle in [REF _Ref685858 \r \h * MERGEFORMAT]) in each triangle progresses in a series (V0, V1, V2, etc). For Direct3D, the first vertex is used as the provoking vertex.



D3D order

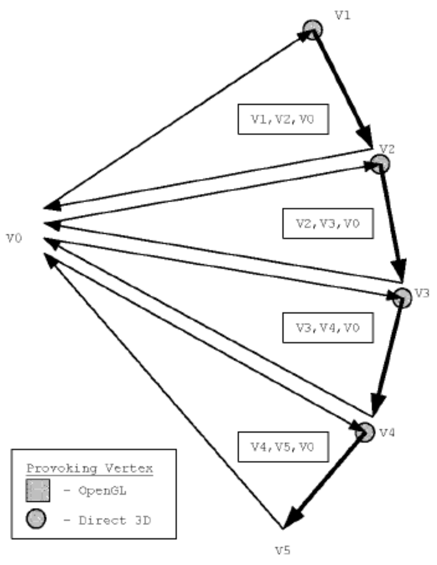
3.11.3 Triangle Fan

The first edge in each triangle is a bold line. Note for OpenGL, the last vertex (shown with a square box [REF _Ref685872 \r \h * MERGEFORMAT]) is used as the provoking vertex.



OpenGL order

The first edge in each triangle is a bold line. Note for Direct3D, the first vertex (shown in a circle in [REF _Ref685888 \r \h * MERGEFORMAT]) is used as the provoking vertex.



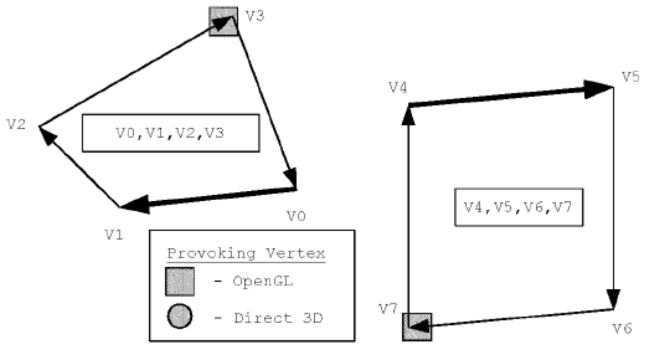
D3D triangle fan order

3.11.4 Quad List

The first edge in each quad is a bold line. Note for OpenGL, the last vertex (shown with a square box [REF _Ref687174 \r \h * MERGEFORMAT]) is used as the provoking vertex.

There is currently no definition of a quad list primitive type in Direct3D (as of DX9, Nov 2001). The VGT design assumes that if such a primitive type were to appear in Direct3D, it would use the first vertex in each quad as the provoking vertex.

The rasterizer hardware cannot render a quad primitive; however, the tessellation engine can process a quad primitive type. Therefore, when processing without the tessellation engine, quad list primitives will be decomposed (by the VGT) into two triangles which each contain the provoking vertex. The vertices will be ordered such that the provoking vertex is the correct for the current provoking vertex state settings. When processing with the tessellation engine, the quad primitives are sent intact to the tessellation engine.

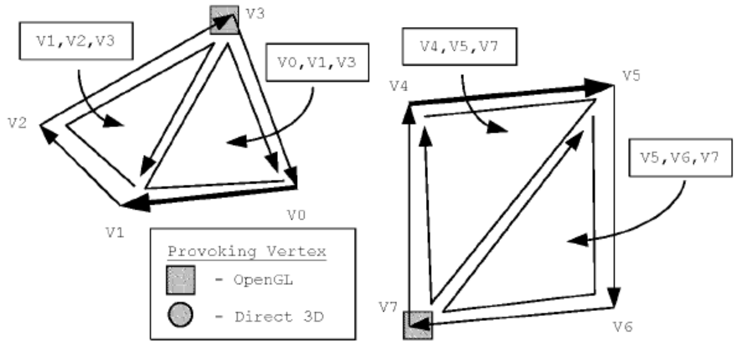


OpenGL order

[REF _Ref775496 \r \h * MERGEFORMAT] shows [REF _Ref687174 \r \h * MERGEFORMAT] with the individual quads decomposed into triangles. The decomposition scheme shown is required because the rasterizer cannot process quad primitives and the provoking vertex must be in each sub-triangle (and it must be the last vertex in each sub-triangle).

#case internal
The decomposition shown is consistent with R100, R200, and R300, but it is inconsistent with Microsoft and nVidia's GeForce3, which both seem to split the other way.
#endcase

The VGT will attach flags to the interior edges of the sub-triangles to prevent those edges from being drawn in wireframe mode. It is not possible, using the rasterizer, to split the quad in such a way that each sub-triangle contains the provoking vertex and that the edges of the original quad are draw in the correct order for line stippling. In this case, the line stipple is sacrificed. After much review, this compromise seems acceptable to anyone that at least had an opinion.



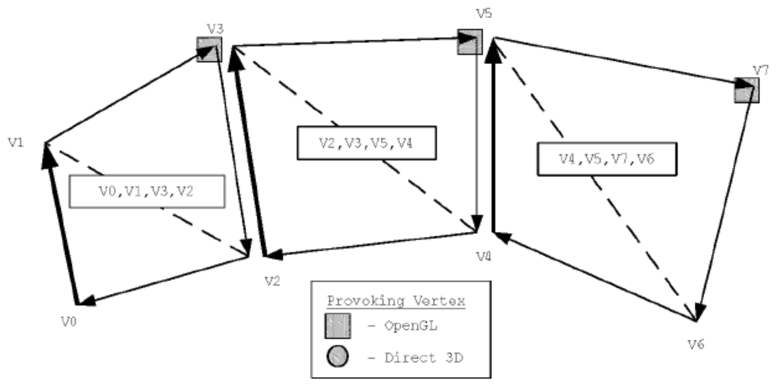
Triangular decomposition of OpenGL quads

3.11.5 Quad Strip

The first edge in each quad is a bold line. Note for OpenGL, the next-to-last vertex in the primitive vertex order (shown with a square box [REF _Ref687713 \r \h * MERGEFORMAT]) is used as the provoking vertex. This vertex is next-to-last in the primitive vertex order, however, it is the last vertex supplied to form the primitive. The dashed lines in [REF _Ref688323 \r \h * MERGEFORMAT] indicate where the vertex list is used out of order.

There is currently no definition of a quad strip primitive type in Direct3D (as of DX9, Nov 2001). The VGT design assumes that if such a primitive type were to appear in Direct3D, it would use the first vertex in each quad as the provoking vertex.

The rasterizer hardware cannot render a quad primitive; however, the tessellation engine can process a quad primitive type. Therefore, when processing without the tessellation engine, quad list primitives will be decomposed (by the VGT) into two triangles which each contain the provoking vertex. The vertices will be ordered such that the provoking vertex is the correct for the current provoking vertex state settings. When processing with the tessellation engine, the quad primitives are sent intact to the tessellation engine.

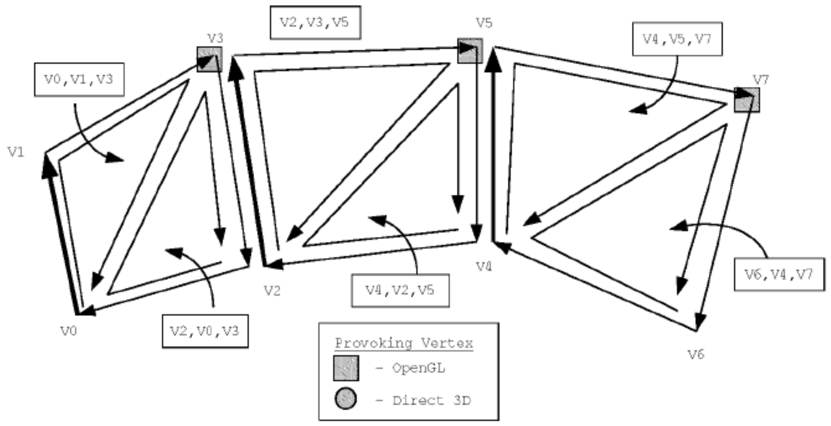


Quadstrip Order

[REF _Ref857238 \r \h * MERGEFORMAT] shows [REF _Ref687713 \r \h * MERGEFORMAT] with the individual quads decomposed into triangles. The decomposition scheme shown is required because the rasterizer cannot process quad primitives and the provoking vertex must be in each sub-triangle (and it must be the last vertex in each sub-triangle).

#case internal
The decomposition shown is consistent with R100, R200, and R300, but it is inconsistent with Microsoft and nVidia's GeForce3, which both seem to split the other way.
#endcase

The VGT will attach flags to the interior edges of the sub-triangles to prevent those edges from being drawn in wireframe mode. It is not possible, using the rasterizer, to split the quad in such a way that each sub-triangle contains the provoking vertex as the last vertex in the sub-triangle and that the edges of the original quad are draw in the correct order for line stippling. In this case, the line stipple is sacrificed. After much review, this compromise seems acceptable to anyone that at least had an opinion.

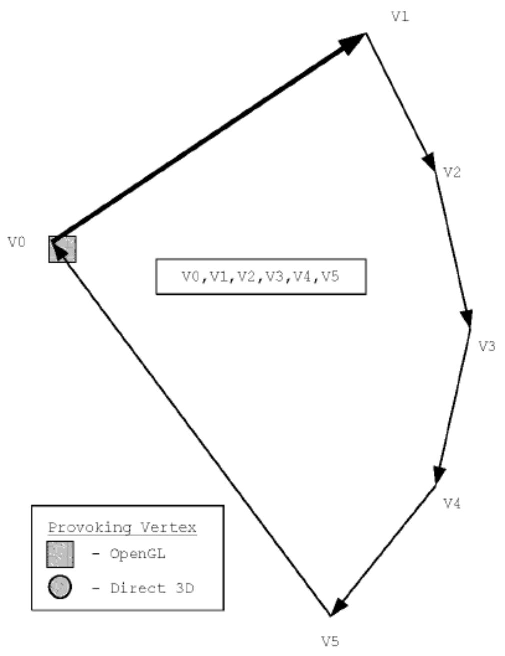


Triangular decomposition of OpenGL quadstrip

3.11.6 OpenGL Polygon Type

The first edge in the polygon is a bold line. Note for OpenGL, the first vertex (shown with a square box [REF _Ref688561 \r \h * MERGEFORMAT]) is used as the provoking vertex.

There is currently no definition of a polygon primitive type in Direct3D. The VGT design assumes that if such a primitive type were to appear in Direct3D, it would use the first vertex as the provoking vertex.

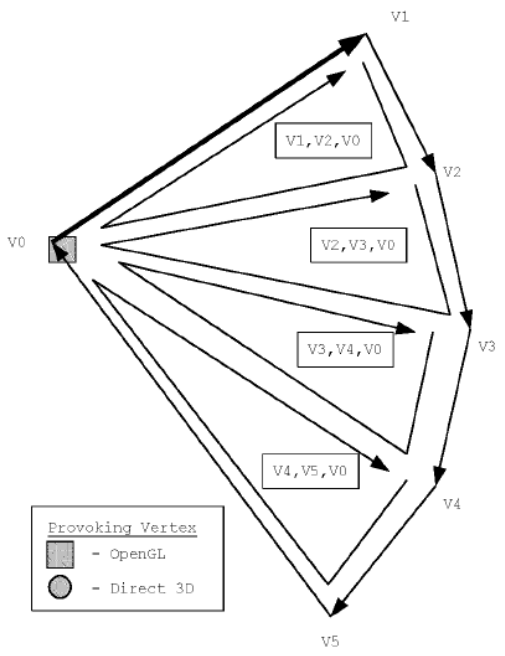


OpenGL polygon

According to OpenGL Programming Guide, Second Edition - page 45, the prim type polygon "draws a polygon using the points v0, …, vn-1 as vertices. n must be at least 3, or nothing is drawn. In addition, the polygon specified must not intersect itself and must not be convex. If the vertices don't satisfy these conditions, the results are unpredictable."

The rasterizer hardware cannot render a polygon primitive. Therefore, polygon primitives will be decomposed by the VGT into an OpenGL-ordered triangle list wherein the vertices of the sub-triangles of the list will be ordered so that the provoking vertex (the last vertex in each sub-triangle) will be the first vertex of the original polygon primitive.

[REF _Ref858551 \r \h * MERGEFORMAT] shows [REF _Ref688561 \r \h * MERGEFORMAT] with the polygon decomposed into triangles. The decomposition scheme shown is required because the hardware cannot process polygons and the provoking vertex must be in each sub-triangle (and it must be the last vertex in each sub-triangle). The VGT will attach flags to the interior edges of the sub-triangles to prevent those edges from being drawn in wireframe mode. It is not possible, using the rasterizer, to split the polygon in such a way that each sub-triangle contains the provoking vertex as the last vertex in the sub-triangle and that the edges of the original polygon are draw in the correct order for line stippling. In this case, the line stipple is sacrificed. After much review, this compromise seems acceptable to anyone that at least had an opinion.



Triangular decomposition of OpenGL polygon

3.11.7 Line Stipple Wireframe Fill Mode of Quads/Polygons

This section addresses line stipple for wireframe fill mode. It applies to the following primitive types: triangles lists, triangle strips, triangle fans, quad lists, quad strips, and polygons. This issue is mostly related to OpenGL.

Do not confuse this topic with line stipple for line lists and line strips which have very strict conformance tests and a very well known industry understanding of correctness.

There is very little consensus in the industry about the "correct answer" for line stipple with wireframe fill mode. This is not a required test mode for OpenGL conformance.

. The VGT achieves this by sending the vertices down the primitive path in the order specified by the API (for both D3D and OpenGL). For triangle lists, the vertex order is specified by the list. For triangle strips and triangle fans, the vertices will be re-ordered so that the primitives start on the correct (API specified) vertex and have the provoking vertex in the correct position within the primitive.

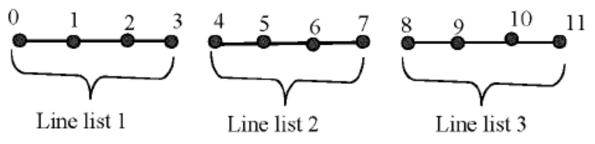
For OpenGL quad lists, the quads must be split into triangles a specific way so that the provoking vertex is in each triangle. This split makes it impossible to achieve the "correct" rendering of wireframe line stippling.

For OpenGL quad strips, it is possible to achieve the "correct" rendering of wireframe line stippling; however, the provoking vertex location would vary between the two sub-triangles of the quad. This requires additional storage in the VGT-Clipper primitive FIFO to communicate the location of the provoking vertex, and it would complicate the VGT-Clipper interface. Therefore, the VGT will not achieve the "correct" rendering of wireframe line stippling for quad strips.

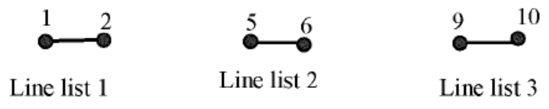
For OpenGL polygons, it is possible to achieve the "correct" rendering of wireframe line stippling; however, the provoking vertex location would differ from all the other OpenGL primitive types. This would require that the OpenGL driver do a check of the primitive type during the "Begin" call. The OpenGL driver team perceives this check to be expensive and very undesirable. . For this reason, the VGT will decompose OpenGL polygons into a triangle list that is consistent with the normal OpenGL triangle list (in terms of provoking vertex). Using this method, the VGT will not achieve the "correct" rendering of wireframe line stippling for polygons.

3.11.8 Line list w/ adj

The above index list will give following as line list w/ adj

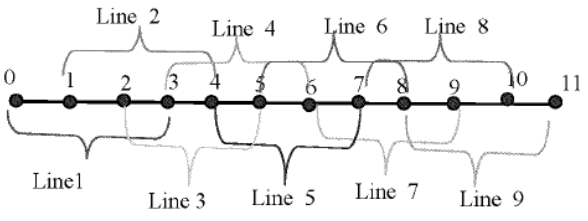


If GS is off, starting and ending indices are ignored and following line lists are generated

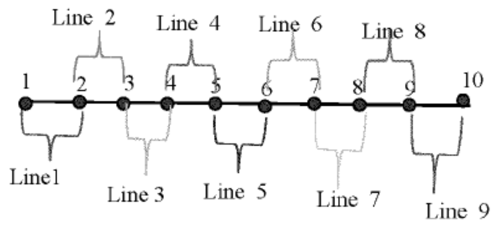


3.11.9 Line strip w/ adj

The above index list will give following as line list w/ adj

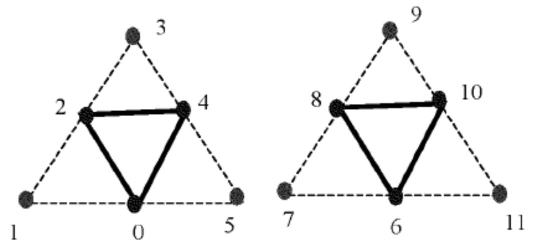


If GS is off, starting and ending indices of each line-list are ignored and following line lists are generated

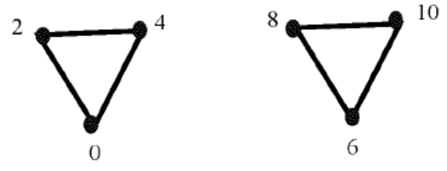


3.11.10 Tri list w/ adj

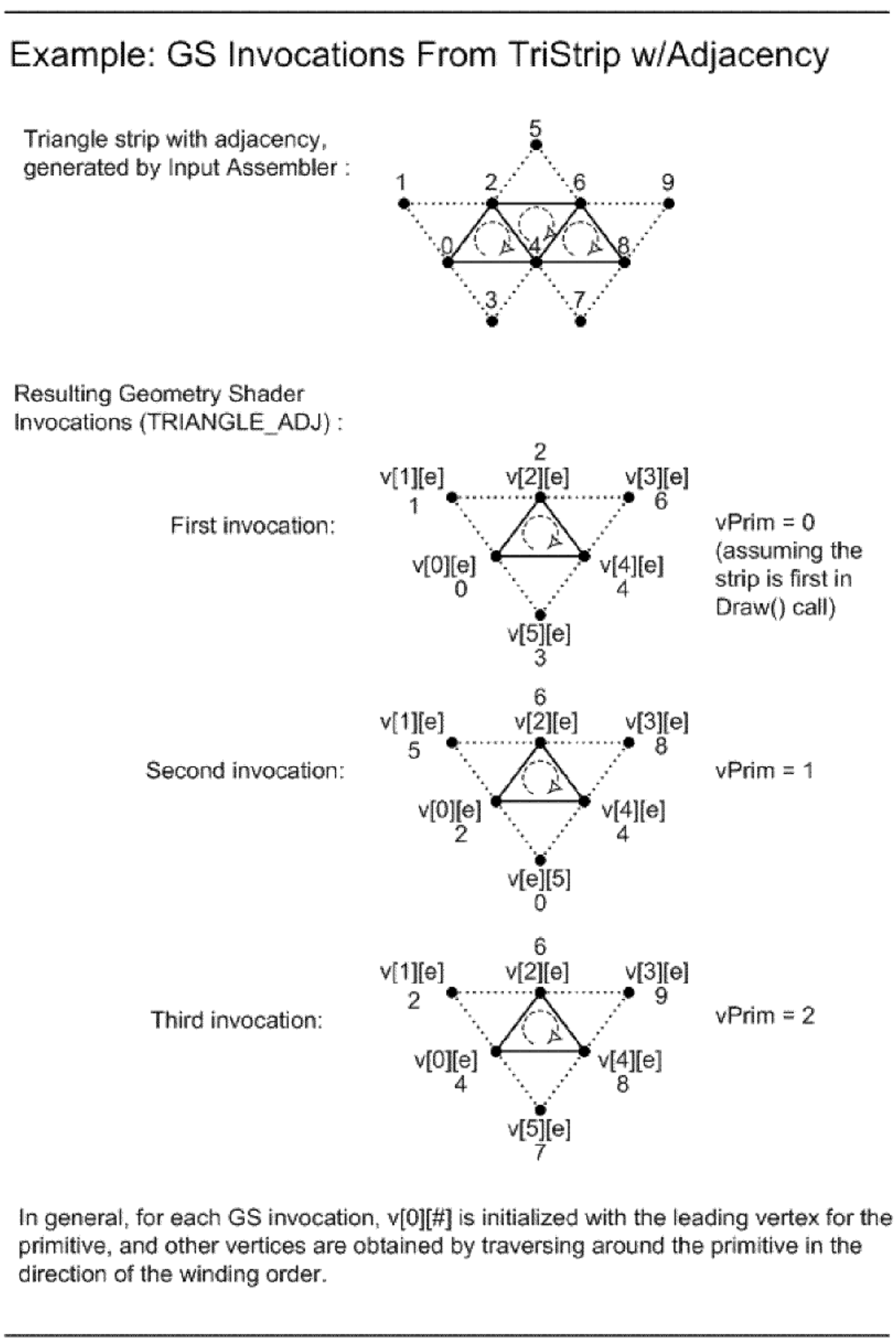
The above index list will give following as tri list w/ adj



If GS is off, odd vertices are skipped and following tri lists are generated



3.11.11 Triangle Strip with Adjacency


Example: GS Invocations From TriStrip w/Adjacency
Triangle strip with adjacency, generated by Input Assembler :
0 1 2 3 4 5 6 7 8 9
Resulting Geometry Shader Invocations (TRIANGLE_ADJ) :
First invocation:
v[0][e] 0
v[1][e] 1
v[2][e] 2
v[3][e] 6
v[4][e] 4
v[5][e] 3
vPrim = 0 (assuming the strip is first in Draw() call)
Second invocation:
v[0][e] 2
v[1][e] 5
v[2][e] 6
v[3][e] 8
v[4][e] 4
v[e][5] 0
vPrim = 1
Third invocation:
v[0][e] 4
v[1][e] 2
v[2][e] 6
v[3][e] 9
v[4][e] 8
v[5][e] 7
vPrim = 2
In general, for each GS invocation, v[0][#] is initialized with the leading vertex for the primitive, and other vertices are obtained by traversing around the primitive in the direction of the winding order.

The above index list will generate following as tri list w/ adj

0 1 2 6 4 3 (note that 5 is not used yet)
2 5 6 8 4 0 (note that 7 is not used yet)
4 2 6 10 8 7 (note that 9 is not used yet)
6 9 10 12 8 4 (note that 11 is not used yet)

If GS is off, odd vertices are skipped and following tri lists are generated

0 2 4
2 6 4
4 6 8
6 10 8

3.12 Geometry Shader (GS) Feature Support

The GS was first defined in DX10 and first implemented in R600 (Pele). Its unit of work is a primitive. The supported input primitive types are point, linelist, linestrip, trilist, tristrip, primitives with adjacency, and patches.

The VGT supports multiple GS modes. Scenario G is the generic mode that implements every feature and the other modes can be used to improve performance in certain circumstances. Scenario G is the most commonly used mode involving three shader stages. ES, GS, and VS (copy shader).

Scenario A involves no vertex modification, and outputs flat shaded parameters based only on primitiveID. Scenario B involves no vertex modification, and outputs flat shaded parameters based on primitiveID and vertex parameters. A and B require the API VS and GS code to be merged and executed on the hardware VS. Scenario B involves some replication of vertex work.

The basic idea of scenario C is that the VGT for a given input primitive type will generate N vertices (defined through VGT_GS_MAX_VERT_OUT using at most 11 bits) connected in the form of user specified output primitive type such as point, linestrip, tristrip. This mode requires merging of the API VS and GS, is executed on the hardware VS, supports amplification/decimation, keeps all data on chip, and performs best for low amplification cases.

The other special mode is point sprite mode. Starting with DX10 point-sprites are handled via the GS. Therefore, for GS cases where the input primitive is a point and the output is four vertices connected as a tristrip, the VGT handles this case by generating four points connected as strip. Point sprite mode requires merging of the API VS and GS and executes on the hardware VS stage.

The above cases are specified using the VGT_GS_MODE register.

3.12.1 Scenario A – flat shading based only on Primitive Id

The Geometry shader copies the input vertices to the output with no modification. A set of flat shaded parameters is also output, and these are dependent only on the primitive id – there is no contribution from any of the vertices. This is the simplest use for the GS. Since there is only a hardware VS stage there are no ring buffers used with Scenario A.

The simple implementation requires a modified version of the reuse logic. The main changes are:

a) Each primitive requires the submission of the provoking vertex to the vertex shader. The VGT will ensure the provoking vertex misses during the reuse check. It is required to maintain the ordering of a primitive's vertices for flat shading.
b) The reuse logic accepts not just source indices for the three vertices, but also a unique, sequential, "primitive id".
c) The provoking vertex "misses" and will have the "primitive id" attached to it. The source and destination address of the vertex is submitted to the vertex shader along with this primitive id. The vertex containing this primitive info is flagged such that the rasterization logic can perform flat shading based on the per-primitive parameters stored with this vertex.
d) The vertex shader first transforms its vertex, outputting the resulting parameters to the parameter cache. It then computes the flat-shaded parameters, performing a lookup based on the primitive id if necessary. The flat-shaded results are also written to the parameter cache at a location higher than the location of the last vertex parameter.
e) During the rasterization pass, the rasterizer reads the vertex parameters and interpolates these values to the current pixel position, passing the result to the pixel shader. It also reads in the flat-shaded parameters (based on its knowledge of which vertex has the primitive id flag set), and passes these values directly to the pixel shader without performing any interpolation.

3.12.2 Scenario B – flat shading based on one or more vertex parameters

In the second scenario, the GS copies the input vertices to the output with no modification. A set of flat shaded parameters is also output, and these are dependent both on the primitive id, and on one or more parameters of the three vertices. As an example, an application might want to color a polygon with the average color of the three vertices. Since there is only a hardware VS stage there are no ring buffers used with Scenario B.

The simple implementation requires a modified version of the scenario-A reuse logic. The main changes are:

a) For each "miss" requiring an attached primitive id, the reuse logic outputs not just the primitive id and the index of the vertex which needs to be transformed (the major vertex), but also the indices of all three vertices in the primitive (the minor vertices). (Note that one of the minor vertices will be the same as the major vertex). VGT will make sure that provoking vertex is non-hit. This is required to maintain ordering of a primitive vertices for flat shading.
b) The vertex shader first transforms its major vertex, outputting the resulting parameters to the parameter cache. It then transforms the three minor vertices – preferably only dealing with those parameters needed by the GS, and also preferably reusing some of the major vertex work. It then uses these transformed minor vertices to compute the flat-shaded parameters, performing a lookup based on the primitive id if necessary. The flat-shaded results are also written to the parameter cache at a location higher than the location of the last vertex parameter.
c) During the rasterization pass, the rasterizer reads the vertex parameters and interpolates these values to the current pixel position, passing the result to the pixel shader. It also reads in the flat-shaded parameters (based on its knowledge of which vertex has the primitive id flag set), and passes these values directly to the pixel shader without performing any interpolation.

3.12.3 Scenario C – 1 prim in N prim out case

In this mode of operation, the VGT will generate N vertices defined by VGT_GS_MAX_VERT_OUT connected in the form of user specified output primitive types, which can be one of the following - pointlist, linestrip or tristrip. This output primitive type is the same as the one declared as the GS output primitive type.

All data stays on chip and there are not separate VS and GS stages.

VGT will send following information per vertex to SPI:

Vector 1: vert_num | instance, primID, index0, index1
Vector 2: index2, index3, index4, index5

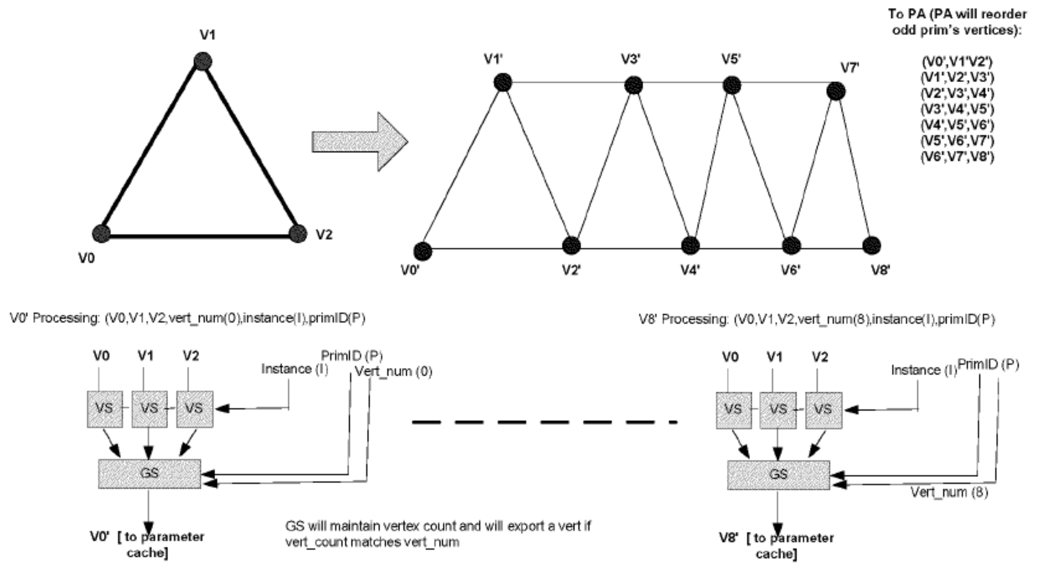
The vert_num varies 0 to N-1. Note that vert_num will take 31:22 bits of r0.x and instance takes bits 21:0.

When the GS_C_PACK_EN bit of the GS_MODE register is set, the indices will be packed together in vector 1 & 2 as follows

Vector 1 : vert_num | instance, primID, index0 | index1, index2 | index3
Vector 2 : index4 | index5

In this case all the prim types except triangle with adjacent information will take only vector 1. It will be shader responsibility to expand vector 1 to two GPRs. It is expected shader will take two instructions to do such expansion. Auto Index mode cannot be used with pack_en because auto index mode needs 32 bit indices and pack_en needs 16 bit ones.

The shader will have access to all the indices of a primitive including the primID and the vertex number (vert_num) for a GS. The vertex shader which is really the VS+GS, will execute on all the vertices of a primitive, followed by primitive based calculation (using the GS portion of code) and for a given vertex (vert_num), do vertex based calculation and vertex (vert_num) is written to the parameter cache. This case is expected to help for GS case when number of emits are small and number of alu operations per emit are low. It will also help in cases where off-chip bandwidth can be an issue for ES/GS rings.



V1
V0
V2
To PA (PA will reorder odd prim's vertices):
(V0',V1'V2')
(V1',V2',V3')
(V2',V3',V4')
(V3',V4',V5')
(V4',V5',V6')
(V5',V6',V7')
(V6',V7',V8')
V1'
V3'
V5'
V7'
V0'
V2'
V4'
V6'
V8'
V0' Processing: (V0,V1,V2,vert_num(0),instance(I),primID(P)
V8' Processing: (V0,V1,V2,vert_num(8),instance(I),primID(P)
V0
V1
V2
Instance (I)
PrimID (P)
Vert_num (0)
VS
VS
VS
GS
V0' [to parameter cache]
GS will maintain vertex count and will export a vert if vert_count matches vert_num
V0
V1
V2
Instance (I)
PrimID (P)
VS
VS
VS
GS
Vert_num (8)
V8' [to parameter cache]

Figure [SEQ Figure * ARABIC] Scenario C work model

Pros:

1) No trip to memory
2) Useful for certain GS apps such as cube map, particle system
3) Even for variable number of output, this mechanism will work. Shader can output position as NaN and "cut" as secondary position.

Cons:

1) redundant vertex and geometry operations
2) no streamout support
3) no viewport and rendertarget index (per vertex) support

1) If primitive information is not needed in a GS, or primitive is just a point, then we can send only vector 1

2) General GS case with "cut"

A "Cut" in the GS, will be stored in the auxiliary vector and sent to the PA. The PA will fetch this and based on the 'cut' value determine if a new strip needs to be started. Primitives with the 'cut' bit set will be dropped. The cut bit is sent after a valid vertex. Repeated cuts are equivalent to one cut. Therefore, in merged shader all the cut without a valid vertex will be dropped and repeated cuts after a valid vertex and a cut, will be dropped as well. Vertex shader puts cut information in bit 1 (literal 0x2)(bit next to edge_flag) of Y term of SX to PA auxiliary vector. If less primitives or vertices are needed than maximum case, cut bit should be set for remaining vertices to kill primitives. These remaining vertices can output vertex position any value. For the pointlist output type, cut should be added for each vertex after actual GS output vertices to up to VGT_GS_MAX_VERT_OUT.

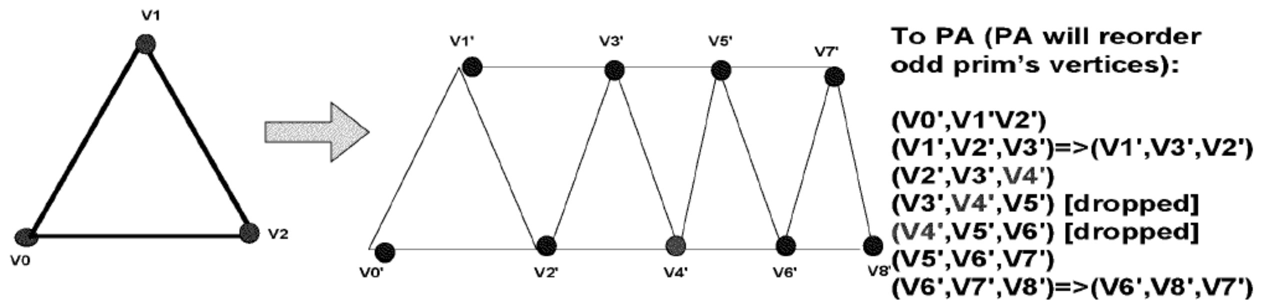
During PA processing of primitive list, the PA will toggle ordering of triangle strip provided by VGT. When cut occurs on vertex aux vector, primitive, where vertex is vertex 2 for triangles and vertex 1 for lines, will be kept. The remaining two primitives for triangles or one primitive for lines with vertex will be discarded. The following triangle primitive will begin again with a fresh toggle sequence.

PA functionality is turned on by setting use_vtx_gs_cut_flag and vs_out_misc_vec_ena in the PA_CL_VS_OUT_CNTL register and setting the gs_mode to SCENARIO_C in the VGT_GS_MODE register.

The PA will get un-ordered vertices when in GS scenario C mode. Whenever the PA gets the first primitive, it assumes the start of a strip and will reorder subsequent vertices.

The VGT provides the following information to the PA

A signal set to true for each primitive that is the first of the strip and a signal which is true for every last prim of a strip. Bits 0 and 1 of the edge flags are used for this respectively.



For a given strip, the PA will begin a new strip for first set of vertices which do not have cut in the auxiliary positions (V5 V6 V7 in the above example)

<u>What compiler has to do:</u>

1. Merge VS and GS shader
2. Set VGT_GS_MAX_VERT_OUT based on maximum number of vertices from GS, which could be based on GS declared GS_MAX_VERT or some optimized value
3. If output primitive type is "pointlist", remove cut from the GS shader
4. Introduce a cut at the end of shader, i.e. when all the vertices are emitted from GS. (This will enable discarding a trilist with two valid vertices, or line with one valid vertex)
5. If GS generates less than VGT_GS_MAX_VERT_OUT vertices then at the end of the shader introduce emitthencut for VGT_GS_MAX_VERT_OUT- number of vertices generated by GS. For such vertices,

emit may not write any position, just add auxiliary "cut" information. This is to reduce the export traffic to parameter cache.

Scenario C will not be useful for cases where variability between VGT_GS_MAX_VERT_OUT and actual emitted vertices is high or the shader is long.

3.12.4 Point Sprite Mode

In order to generate point_sprite in DX10 API, GS shader will be used. A given point is expanded to a quad which is rendered as two triangles. The Scenario G, GS path may not be optimal for such implementation because of high memory bandwidth requirement.

Therefore, an alternative option exists in VGT. In this case, primitive type is declared as point and sprite_en mode is set in vgt_gs_mode. VGT will form a quad out of single index as below.

V0: (2'b00, Index)
V1: (2'b01, Index)
V2: (2'b10, Index)
V3: (2'b11, Index)

Here index will be left shifted by two bits and upper two bits (as above) will identify distinct points of quad.

This quad is converted into two triangle as follows

OGL : (V0, V1, V3) and (V1, V2, V3)
D3D : (V0, V1, V2) and (V0, V2, V3)

After reuse only V0, V1, V2, V3 are part of vertex process vector on which GS for point sprite is executed. The upper 2 bits of these vertices are used to change the point position, color based on lower 29 bits index of each vertex.

This method only allow index value for point upto 2^29-1. For draw_call with index value more than 2^29-1, regular GS path will be used.

This mode can be used with stream out enable (reuse off) and draw_auto call.

If primitiveID is used in the shader not as flat shaded attribute, reuse should be turned off. In case of primitiveID enable, only leading vertex of point-sprite is made non-hit in the reuse buffer. The other vertices can be reuse and can have primitiveID of previous points. Therefore, if primitiveID for all the vertices of point-sprite quad is to be used, then reuse should be turned off.

3.12.5 Scenario G – Geometry shaders which create and/or modify geometry

Scenario G covers all Geometry shader cases, handling a variable amount of output. This scenario uses 3 internal passes through the shader complex (SH). These passes are the ES (export shader), GS (geometry shader), and VS (vertex shader). The GS is the only stage to match the API in name. For scenario G the VS is also known as the copy shader. Ring buffers are used to pass data between stages.

3.12.5.1 ES Thread

The ES is a vertex shader and from the API's perspective it's the VS or DS, the output of this shader is written to the ESGS ring buffer. A vertex reuse check is performed prior to sending ES verts to the SPI. A base address for ring buffer exports is send on the VGT to SPI ES wave interface. While the VGT sends indices to the ES, it prepares offsets for each GS primitive. These offsets point to each primitive's vertex data in the ESGS ring buffer. Note that no primitive information is sent PA for this step.

3.12.5.2 GS thread

Once, the VGT has prepared 64 primitives a GS wavefront is issued. Vertex data is fetched from the ESGS ring buffer using ESGS_ring_base + vertex_offset, where vertex_offset are specified as vertex indices for GS input primitives. The output of a GS shader is written to the GSVS ring buffer. The starting base address for exporting the output of a GS wave is specified per wave. Note that each GS input primitive can output a variable number of primitives. Output counters are updated as the GS emits vertices. Cut locations are recorded in a memory. No primitive information is sent PA in this pass as well.

3.12.5.3 VS Thread

This is the copy shader and it uses same interface as normal vertex shader. It is named the copy shader because its purpose is to copy data from the GSVS ring buffer to the parameter cache and position buffer. When a GS wave finishes the GoG (GS Output Geometry) block goes through the counters of completed GS instances in order and generates per vertex read addresses that are sent to the shaders via the SPI vsvert interface. Primitive information is sent to the PA. Note that, if stream output is enabled, strip primitive types are converted into lists. A base offset for streaming output per VS wave is sent to the SPI. The SQ uses this offset to calculate a streaming output address per VS wave.

3.12.5.4 ESGS Ring Buffer

Calculation for the recommended ESGS ring buffer size:

Max_gs_waves_supported * 2 * vert:prim ratio * SQ_ESGS_RING_ITEMSIZE * num_quad_pipes * 16 = ESGS ring size in dwords.

The vert:prim ratio is determined by the prim type. 4:1 is probably reasonable, but it could be set based on the prim type. Tri list is 3:1, adjacent line list is 4:1, adjacent tri list 6:1, and patch can be up to 32:1.

At minimum the ESGS ring buffer must hold the same number of waves as the depth of the reuse buffer. This is because in write optimize mode we allocate space for the entire wave even if there is a single vertex in the wave. So each of the vertices in the reuse buffer are holding onto an entire wave's worth of space.

3.12.5.5 GSVS Ring Buffer

Calculation for the recommended GSVS ring buffer size:

Max_gs_waves_supported * 2 * max_vert_count * num_quad_pipes * 16 * SQ_GS_VERT_ITEMSIZE = GSVS ring size in dwords.

3.12.5.6 VGT_GS_PER_ES register

This specifies maximum number of gs prims per es wave. The high reuseability of vertices may cause few ES vertices to generate many GS waves. Once the VGT_GS_PER_ES limit is hit, a ES wave is issued.

Depending on the chip there is a different amount of GS prim buffering and thus a different maximum GS per ES setting. There is more buffering for non-adjacent prim types, but since this is a single context register the worst case is likely to be used. Here's an equation showing the recommended setting.

VGT_GS_PER_ES = min((((GSPrimBufferDepth + 16)/2) + NumGsPrimsPerThread), 256);
GSPrimBufferDepth is defined in features.ale. gpu.vgt.gsprim_buff_depth

A maximum GS_PER_ES of 256 is a tradeoff between reducing ES workload due to reuse and ensuring GS resources are not idle for too long. For a chip like Pele with 64 NumGsPrimsPerThread this allows a reuse ratio of 4:1. 256 GS prims from 64 ES verts. If it is known that the application has no adjacent primitive types then NumGsPrimsPerThread can be multiplied by 3 in the previous equation.

VGT_GS_PER_ES must be greater than or equal to GS_PRIMS_PER_SUBGRP.

3.12.5.7 VGT_ES_PER_GS Register

This register specifies maximum number of es vertices per gs thread. This register setting could result from setting lower ESGS ring buffer size. For a given GS thread, once a specified number of es vertices are issued as part of number of es process vectors, GS thread can be issued. This limits number of es process vectors which forms a GS process vector.

Besides the ring buffer size VGT_ES_PER_GS is further limited by the SQ's thread allocation ratio for ES to GS threads. The maximum value is the ES:GS ratio * 4 * 16. Typically 2-6 es threads per gs -> 128-384 esprim per gs.

3.12.5.8 VGT_GS_PER_VS Register

This register specifies maximum number of gs threads per vs thread. It is possible GS shader outputs very few or zero vertices or partial primitives. Therefore, a VS process vector may take many many gs process vector. It is recommended this register is set to two which specifies that if a VS process vector generation already spanned over two GS threads, then issue VS process vector. In this case, VS process vector may even contain zero vertices. A zero vertex VS process vector will not be issued but GS threads memory will be invalidated. The maximum value this register can be set to number of GS threads which could be stored in GS2VS ring buffer.

3.12.5.9 Using the maximum number of GS wavefronts

There are four cut modes defined in VGT_GS_MODE. The proper mode should be used to ensure the best performance.

GS_CUT_1024	ALPHA+="GS_CUT_1024",
GS_CUT_512	ALPHA+="GS_CUT_512",
GS_CUT_256	ALPHA+="GS_CUT_256",
GS_CUT_128	ALPHA+="GS_CUT_128"

i.e. cut mode is set to
00 : if more than 512 emits in GS
01: more than 256 but less than equal to 512 emits in GS
10: more than 128 but less than equal to 256 emits in GS
11: less than equal to 128 emits in GS

This allows more GS wavefronts to be issued by the VGT and improves performance for certain apps. GS_CUT_128 allows for the maximum number of GS waves to be in flight.

3.12.5.10 ES Pass-through mode

It is expected in some of GS apps, vertex shader size to be much smaller than geometry shader size. For such apps, if instancing is off, VS (Hardware ES) can be folded into GS itself.

Pros:

- One pass to off-chip memory for writing to ES ring buffer is avoided and hence initial latency of starting GS threads is saved.

Cons:

- No vertex reuse
- Repetitive vertex evaluation in GS
- No instance ID support

In order to enable this mode, ES_PASSTHRU bit is set in VGT_GS_MODE register. VGT will need to be flushed using VGT_FLUSH between draw packets if ES_PASSTHRU bit is changed between two consecutive GS packets.

Driver will also merge VS and GS together. The ideal usage are GS cases where VS (Hardware ES) are passthru shader.

3.12.5.11 Force partial ES waves at the end of a prim group

Setting IA_MULTI_VGT_PARAM.bits.PARTIAL_ES_WAVE_ON=0 allows the combining of ES data across prim groups. Setting it to 1 forces partial es waves to launch at the end of a prim group. For the best performance, PARTIAL_ES_WAVE should be set to 0.

Programming Restrictions:
(1) For all modes (except old 7xx style compute mode)
(GS_PER_ES / PRIMGROUP_SIZE) must be lesser than (GPU__VGT_GS_TABLE_DEPTH - 2)

Partial ES waves can cause hang conditions if either of the above cases are not adhered to.

If setting PARTIAL_ES_WAVE is not desirable for performance reasons, setting the register IA_MULTI_VGT_PARAM.MAX_PRIMGRP_IN_WAVE can be used to eject the wave once the wave contains the programmed number of primgroups. If this is programmed to 0, no ejecting is performed.

3.12.5.12 Write combining optimizations for the ESGS and GSVS ring buffers

It's possible the ring buffer traffic will thrash the cache. To solve this there is a write through mode to the cache to make sure this type of data is not resident for long and doesn't thrash other data. To help performance in the new mode, the order of the data in the ESGS and GSVS rings will be changed such that write combining is explicit. The VGT will provide updated ring offsets, the read/write cache will support a write through (stream) mode and the shaders will use TA hardware to perform the address calculations to write/read from the ring buffers.

The old mode of operation will also be supported with off chip GS and must be used with on chip GS. Two new fields have been added to control whether this write combining mode is supported in each ring.

VGT_GS_MODE.bits.ES_WRITE_OPTIMIZE
VGT_GS_MODE.bits.GS_WRITE_OPTIMIZE

The data structure of the rings are fully described in GS_SI_AOS_proposal.vsd. The short explanation is the old mode stores all of vertex 0's data together, then all of vertex 1's data, etc. Write optimized mode packs dword 0 for vertex 0 next to dword 0 for vertex 1, so all dword 0's for a wavefront are written to the same cacheline.

Write optimized mode might require a bit more ring buffer space as even partial waves are allocated the output space needed by a full wave. For the old (read optimized mode) space is allocated on the fly so there is no wasted space for a partial wave.

3.12.6 Programming Restrictions

The table below shows some programming restrictions , the notes describe driver actions

	GFX 6	**GFX 7**		**GFX 8**		**Notes**
		<= 2 SEs	> 2 SEs	<= 2 SEs	> 2 SEs	
SWITCH_ON_EOI						
PrimId for Tess or GS	X	X	X	X	X	Validation
WD_SWITCH_ON_EOP == 0			X		X	Draw-time
PARTIAL_VS_WAVE						Automatically set by h/ware for stream-out
Tess + GS enabled	X*	X*				Validation *for bugs 357261 and 394014.
Distributed Tess no GS				X	X	Validation
SWITCH_ON_EOI==1			X	X*	X*	Draw-time (dep on EOI) *not needed if MAX_PRIMGRP_IN_WAVE==2, GS is disabled and using list prim types.
SWITCH_ON_EOI==1 && instCount > 1		X*				Draw-time (dep on EOI, instances) *only for Bonaire bug 504861
PARTIAL_ES_WAVE						Automatically set by h/ware for on-chip GS
PrimId for Tess or GS	X	X	X	X	X	Validation
GS table depth check		X	X	X	X	Validation
Distributed Tess with GS				X	X	Validation
SWITCH_ON_EOI==1		X	X	X	X	Draw-time (dep on EOI)
WD_SWITCH_ON_EOP						Automatically set by GFX7/GFX8 h/ware for <= 2SEs
SWITCH_ON_EOP == 1 || primtype == TRISTRIP_ADJ ||RESET_EN ||OpaqueDraw || NumSEs < 4		X	X	X	X	Draw-time
MAX_PRIMGRP_IN_WAVE						
Default				2	2	Validation

3.13 On Chip GS (GFXIP 7.2)

3.13.1 Introduction

When operating in onchip GS mode, the VGT will use LDS space to store the ESGS and GSVS ring buffers. This will eliminate the traffic to offchip memory that is currently necessary. Typically, scenarios with small amplification will benefit the most from this approach. Today, ES/GS draw calls use two offchip ring buffers that wrap individually, but are of a fixed size and base for the entire application. In the onchip GS mode, the VGT will partition the ES/GS work into smaller chunks called subgroups. Each subgroup gets allocated to a Compute Unit (CU) by the SPI and stays on the same CU for the duration of its lifetime (which is ES/GS/VS). To make the operation simple, there is no wrapping supported within the LDS ESGS and GSVS ring buffers. This is handled by making the subgroups small enough to fit entirely within the LDS space.

A new field called LDS_SIZE will be added to the ES resource control register in the SPI, this will contain the total amount of space available for the EG+GS subgroup.

There are 2 user defined parameters which control the size of a subgroup. And are calculated as follows

GS_PRIMS_PER_SUBGRP - The amount of GS prims that can fit in the LDS

GS_PRIMS_PER_SUBGRP = GSVS_LDS_SIZE_in_dwords / GSVS_RING_ITEMSIZE / VGT_GS_INSTANCE_CNT

ES_VERTS_PER_SUBGRP - The worst case number of ES vertices needed to create the GS prims specified above

ES_VERTS_PER_SUBGRP = min((ESGS_LDS_SIZE_in_dwords / ESGS_RING_ITEMSIZE)-5,
(GS_PRIMS_PER_SUBGRP * num_verts_per_prim))

We might find we want to target 64 or 128 GS_PRIMS_PER_SUBGRP and if these fit in available LDS space we use on chip rendering otherwise we use off chip rendering. This will also allow us to fit more than 2 groups per CU which might be important since pixel waves and tessellation will also compete for LDS space. Values less than 64 are supported and this may result in partial waves.

There is a caveat with GS_PRIMS_PER_SUBGRP. Adjacent primitive types and GS instancing use two slots per prim in the SPI staging registers. This means GS_PRIM_PER_SUBGRP must be capped at 128 for adjacent primitives and GS_PRIMS_PER_SUBGRP must be <= 128 / VGT_GS_INSTANCE_CNT.

When the VGT begins processing ES/GS work, it aggregates the work into a subgroup until either of the two parameters above are reached. At this point the subgroup is complete and the subsequent work begins a new subgroup. Any given ES/GS subgroup may contain multiple waves depending on the two parameters above. Each ES/GS wave in the subgroup has a first/last of subgroup flag to demarcate the start and end of the subgroup. A single original DX11 tessellation threadgroup may have multiple ES/GS subgroups in it. A DX10 GS scenario G draw call will also be split up into subgroups when in onchip GS mode.

The offchip ES/GS ring buffers and their state/pointers are not updated or used when a GS onchip draw call is being processed. The application can alternate between onchip and offchip GS operation between draw calls.

The VGT will pick up from where the last offchip GS packet was for the offchip ES/GS ring buffer pointers/state. There is no performance penalty when switching modes.

In order to prevent a deadlock at least 1 CU per SH must be off limits to LDS allocation for onchip GS and LS/HS thread groups.

Other programming restrictions are VGT_GS_MODE fields ES_WRITE_OPTIMIZE and GS_WRITE_OPTIMIZE must be 0 as they don't allow for partial waves to be allocated. ES_PER_GS should be >= ES_VERTS_PER_SUBGRP and GS_PER_ES should be >= GS_PRIMS_PER_SUBGRP.

Here's a summary of the register field programming.

VGT_GS_MODE.MODE = 3
VGT_GS_MODE.ES_WRITE_OPTIMIZE = 0
VGT_GS_MODE.GS_WRITE_OPTIMIZE = 0
VGT_GS_MODE.ONCHIP = 3
VGT_GS_ONCHIP_CNTL.ES_VERTS_PER_SUBGRP = using equation above
VGT_GS_ONCHIP_CNTL.GS_PRIMS_PER_SUBGRP = using equation above

3.13.2 Data flow for Dx10 (ES-GS-VS)

The VGT will break up the GS Scenario G draw call into multiple subgroups. All the waves from an ES subgroup will be submitted to the SPI. The SPI will allocate LDS space for the onchip ES/GS subgroup when it gets an ES wave with the first subgroup bit. The SPI will launch the ES subgroup on a CU that has enough LDS space available for the entire ES/GS subgroup. This information is available by snooping the VGT registers. The VGT will insert a FLUSH_ES_OUTPUT event as it does for off-chip GS. The SPI will send an ES_done signal back to the VGT for each FLUSH_ES_OUTPUT event after it finishes the preceding ES waves.

On receiving the ES_done, the VGT will submit all the corresponding GS waves to the SPI. The SPI will make sure these waves are assigned to the same CU that the ES (s) executed on. They may use the same or different SIMD units on the same CU. The SPI will send a GS_done signal back to the VGT after each GS wave completes (as it does in 10xx and is also the same as off-chip GS).

The VGT will dispatch the corresponding VS waves after it gets the GS_done signal. The SPI will also launch the VS work on the same CU and return a VS_done signal after each VS wave is complete. The SPI will deallocate the ES/GS LDS space for the entire subgroup when it sees a VS wave with the last subgroup bit set. At the end of each subgroup the GS (RCM) reuse buffer is cleared.

The diagram below illustrates the data flow for GC Scenario G

[EMBED Visio.Drawing.11]

3.13.3 Data flow for Dx11 (LS-HS-ES-GS-VS)

The VGT will break up the Dx11 draw call into threadgroups. All the LS waves for a given threadgroup will be dispatched to the SPI. The VGT will insert a FLUSH_LS_OUTPUT event after each LS threadgroup. The SPI will allocate LDS space for the LS/HS threadgroup when it receives the LS wave with first bit of threadgroup set, the offchip HS memory is also allocated at this time. In this mode, dynamic_hs MUST be 1 and num_ds_waves MUST be 0. This will enable both LS/HS and ES/GS to use LDS space (not necessarily the same LDS on a CU). Once all LS waves finish, the SPI returns a LS_Done signal to the VGT.

The VGT will then dispatch all the HS waves in the threadgroup followed by a FLUSH_HS_OUTPUT event. The SPI launches all the HS waves to the same CU that was used by the LS. The SPI will return a HS_Done signal to the VGT after all the HS waves in the threadgroup complete. The onchip LS/HS LDS space is deallocated when the last HS wave of the threadgroup with the lshs_dealloc bit set finishes. At this point all the local LDS data is also copied over to the offchip HS memory.

After the fixed function tessellator runs, the VGT begins to process the DS work (in this case the ES). The original threadgroup is now broken down into ES/GS subgroups. All the waves from an ES subgroup will be submitted to the SPI. The SPI will allocate LDS space for the onchip ES/GS subgroup when it gets an ES wave with the first subgroup bit. The SPI will launch the ES subgroup on a CU that has enough LDS space available for the entire subgroup, this may or may not be on the same CU that was used by the LS/HS threadgroup. The VGT will insert a FLUSH_ES_OUTPUT event after each set of ES waves. The SPI will send an ES_Done signal back to the VGT for each FLUSH_ES_OUTPUT event after the preceding ES waves finish.

On receiving the ES_done, the VGT will submit the corresponding GS wave(s) to the SPI. The SPI will make sure these waves are assigned to the same CU that the ES executed on. They may use the same or different SIMD units on the same CU. After each GS wave is complete, the SPI will send a GS_Done signal back to the VGT.

The VGT will dispatch corresponding VS waves after it gets the GS_Done signal(s). The SPI will also launch the VS work on the same CU and return a VS_Done signal after each wave completes. The SPI will deallocate the ES/GS LDS space when it sees a VS wave with the last subgroup bit set.

The SPI will deallocate the offchip HS memory when it sees a DS (ES) wave with the last wave in threadgroup bit (these first and last are for the entire original threadgroup, not the first/last ES/GS subgroup).

The Dx11 tessellator will monitor the DS verts and will internally clear its reuse buffer at the end of a subgroup. Some duplicated data might need to be sent on the next subgroup.

The first/last subgroup flags will be set to 0 in GS off-chip mode.

With on chip GS enabled HS and GS thread groups don't have access to all CUs. See the SPI spec for details

The diagram below illustrates the data flow for the full Dx11 pipeline using onchip GS

[EMBED Visio.Drawing.11]

3.13.4 Information provided to the ES/GS/VS shaders

ES : The ES can exist in two separate precompiled forms, one for on-chip and one for off-chip. The appropriate shader can be bound to the current draw packet based on which GS mode was selected. It is recommended to compile two forms of the ES as at compile time it is not known if the ES will be paired with an on or off chip GS.

GS : All the parameters needed to determine if the mode selected will be on-chip or off-chip are present when compiling the GS shader. (amount of amplification, size of the vertices etc). A decision can be made on the fly based on these attributes and the GS can be compiled as either on or off chip. The compiler/driver may choose to compile two forms of the GS shader

VS : There is no API VS so the driver/compiler will compile the VS the same as the GS

3.13.4.1 ES Shader

3.13.4.2 Data Fetch

3.13.4.2.1 Tessellation Off

When in on-chip mode with no Dx11 tessellation, the ES will fetch data from the offchip vertex buffer. This is done exactly like the way it is in the GS off-chip mode of operation.

VGT	Will provide index data that points to an off-chip vertex buffer
TA	Multiplies the stride with the index (in bytes), adds the vertex buffer base address, and returns the data from memory
SQ	Executes shader fetch instruction and sends it to the TA
SHADER	User/compiler passes a pointer which specifies where the vertex format descriptor is located. The descriptor is fetched into the shader. Then the shader executes an instruction to fetch the vertex data

3.13.4.2.2 Tessellation On

When Dx11 tessellation is active and on-chip mode is used, the ES shader (now the DS) will fetch data from the off-chip HS memory.

VGT	Provides patchid_per_thdgrp
SPI	Loads the per threadgroup offchip HS memory base chunk into and SGPR (in bytes)
TA	Multiplies the stride (1 dword) with the shader offset (in bytes), adds the HS offchip base address, and returns the data from memory
SQ	Encodes the instruction and sends it to the TA
SHADER	The shader MUST be defined in HS offchip mode The user/compiler will define a HS offchip surface The shader will setup pointers to the offchip resource and fetch it. Execute a fetch using the VGT patchid (and other parameters, size of each vert etc) and the resource and SPI offset The shader skips over the LS portion of the LDS using num_in_cntrl_points and the stride etc. This is available in the shader using user data SGPRs (16 are available) Driver/compiler uses constant buffers – can be arbitrary in size

3.13.4.3 Data Export

The ES shader will export data to the LDS.

VGT	Will provide a per wave starting offset into the LDS space. This will be 0 for the first wave of each subgroup
SPI	Will forward the LDS base address to the SQ (per wave)
SQ	The LDS base address is added to the offset calculated by the shader in the SQ
SHADER	The shader will use thread-id in wave and the per-wave offset provided to calculate the write address into the LDS write address = wave_offset + thread_id * ESGS_RING_ITEMSIZE Address computation will be different – old mode

3.13.5 GS Shader

3.13.5.1 Data Fetch

The GS shader will fetch data from the LDS which was written earlier by the ES.

VGT	Will provide relative dword offsets into the LDS space for fetching from the ESGS ring buffer.
SPI	Passes the LDS base to the SQ
SQ	The base from the SPI is added to the shader offset, fetches from the LDS, and returns data to the shader
SHADER	Executes lds_read using the VGT offset

3.13.5.2 Data Export

The GS shader will export data to the LDS.

VGT	Will provide a per wave starting offset into the LDS space. This will be 0 for the first wave of each subgroup
SPI	Will forward the LDS base address to the SQ
SQ	The LDS base address is added to the offset calculated by the shader in the SQ
SHADER	The shader will use thread-id in wave and the per-wave offset provided to calculate the write address into the LDS. The shader is responsible for skipping over the ES portion of the LDS. This data is fed by the user to the shader using user data SGPRs write address = ESGS_LDS_SIZE + wave_offset + thread_id * GSVS_RING_ITEMSIZE The user/compiler has determined the ESGS_LDS_SIZE also in constant buffers.

3.13.6 VS Shader

3.13.6.1 Data Fetch

The VS shader will fetch data from the LDS which was written earlier by the GS.

VGT	Will provide subgroup relative dword offsets into the LDS space for fetching from the GSVS ring buffer.
SPI	Passes the LDS base to the SQ
SQ	The LDS base address is added to the offset calculated by the shader in the SQ
SHADER	The shader will use the offset provided by the VGT. The shader is responsible for skipping over the ES portion of the LDS. This data is fed by the user to the shader using user data SGPRs Offset = ESGS_LDS_SIZE + vertex_offset (from VGT)

3.13.6.2 Data Export

The VS shader will export data to the position and the parameter caches.

There is no change between the on-chip and the off-chip versions for the VS exports.

3.13.7 Reducing bank conflicts

To reduce LDS bank conflicts pad VGT_ESGS_RING_ITEMSIZE and VGT_GS_VERT_ITEMSIZE to be odd. For example 0x8 is increased to 0x9. After recalculating VGT_GSVS_RING_ITEMSIZE it is also padded to be odd, if maxvertout is a multiple of 4.

3.14 Streamout

3.14.1 Introduction

Streamout was initially introduced with DX10 and has evolved since then. It is effectively an ordered append operation from the graphics pipeline with the oldest primitive writing to the lowest address. It is orthogonal to rasterization such that a stream can Streamout and/or rasterize. Vertex reuse is disabled so strips are automatically expanded to lists and only full primitives are streamed out. The VGT maintains counts indicating the number of dwords written to each of 4 buffers and the offset into those buffers.

The Streaming out of the data is done in the VS. The VGT is counting the amount of data to be streamed out.

Two events are necessary to setup and read Streamout status. SO_VGTSTREAMOUT_FLUSH adds the VGT_STRMOUT_BUFFER_OFFSET value for each buffer to the dwords written count and stores the result in the VGT_STRMOUT_BUFFER_FILLED_SIZE registers. There's a VGT_STRMOUT_BUFFER_FILLED_SIZE register per buffer and each pipeid has its own copy so there a 8 registers in total.

A VGT_STREAMOUT_RESET event is needed any time a VGT_STRMOUT_BUFFER_OFFSET register is written. This resets the internal dwords written counts and the event is pipelined so there's no performance impact. The CP has a Streamout Update packet which automates some of the programming.

There are only 4 buffers, but since all Streamout registers with the exception of VGT_STRMOUT_BUFFER_FILLED_SIZE* are multi-context the driver can setup work for future draws without idling the pipe.

As the VGT processes a primitive it sends increment signals to the CP indicating if the primitive is streamed out or wanted to Streamout. The primitives written count (in the CP) indicates the number of primitives that are written to the Streamout buffer and the primitives needed count (in the CP) indicates the number of primitives wanting to streamout. If the buffer never gets full these counts will match. If any buffer active for the current stream goes full all writes for the stream are stopped.

The following picture shows how multiple streams are stored with the legacy (read optimized) mode.

[EMBED Visio.Drawing.11]

3.14.2 Registers

See the register spec for a detailed listing of registers. Most of the registers are multi-context so new draws can be setup without idling the chip. Here we briefly describe the two registers that enable Streamout.

VGT_STRMOUT_CONFIG enables up to four streams and indicates which stream(s) rasterize.

VGT_STRMOUT_BUFFER_CONFIG has four bit fields for each buffer indicating which of the four streams write to the buffer. A buffer cannot be written by multiple streams and the emulator (c-model) will assert if it detects this programming.

3.14.3 Propagation of Streamout Data

After reset VGT0/SE0 has priority and the other VGT's cannot send VS waves to the SPI until they know where in the streamout buffer to begin. The dwords written and index data is communicated via a daisy chain which at this time is 36 bits wide. Since the WD and IA distribute work sequentially each VGT only needs to wait on the prior VGT in the chain.

2 SE designs have their inputs and outputs connected to each other, but a 4 SE design will have VGT0 receiving input from VGT3 and outputting to VGT1. Apart from the connection difference the internal logic does not differ between 2 and 4 SE's.

[EMBED Visio.Drawing.11]

3.14.4 Synchronizing Streamout Flushes/EOP

All VGT's need to synchronize at the end of a packet or when there is a sync event like SO_VGTSTREAMOUT_FLUSH, VGT_STRMOUT_RESET, or VGT_STRMOUT_SYNC.

Since there is no priority here a daisy chain requires all syncs to be forwarded resulting in higher latency as the number of SE's increases. For primarily this reason there is no daisy chain for syncs and the WD is used to accumulate and acknowledge sync events.

When a VGT sees a eop or sync event it will inform the WD via a single pulse and wait for acknowledgement. When the WD receives a pulse from all SEs it will send an acknowledge pulse and clear the internal register that collected the syncs.

[EMBED Visio.Drawing.11]

3.14.5 Different primitive types per stream

The DX API forces the GS output primitive type to be POINTLIST when multiple streams are enabled.

In ASICS up to 9xx, the VGT went a step further and allowed setting ANY one of the following primitive types (POINTLIST, LINESTRIP or TRISTRIP) but with the restriction that, the same primitive type was used for all the streams.

In 10xx and beyond, the VGT will support setting a unique primitive type per stream. This feature is implemented in a backwards compatible manner. A new field UNIQUE_TYPE_PER_STREAM is added to the VGT_GS_OUT_PRIM_TYPE register. If this bit is set, unique prim types per stream are supported and the other new fields in the register are used, one prim type per stream.

Outprim_type	(stream 0)
Outprim_type_1	(stream 1)
Outprim_type_2	(stream 2)
Outprim_type_3	(stream 3)

If the unique_type_per_stream bit is not set, the old mode of operation is maintained and OUTPRIM_TYPE is used for all enabled streams.

3.14.6 Multiple streams can be rasterized while streaming out

Prior to 10xx only one stream could be rasterized and the RAST_STREAM field was an ID indicating the stream to be rasterized.

If USE_RAST_STREAM_MASK is set to 1 RAST_STREAM_MASK is used instead of RAST_STREAM. RAST_STREAM_MASK is a 4 bit field with each bit representing a stream. If the bit is 1 the corresponding stream will be rasterized.

The SPI will shadow the new register to make sure parameter cache space is reserved for all the streams that want to rasterize.

3.14.7 D3DQUERYTYPE_STREAMOUTPUTSTATS

```
typedef struct D3DDEVINFO_STREAMOUTPUTSTATS {
  UINT64 NumPrimitivesWritten; /* Number of primitives written to the stream output resource
*/
  UINT64 PrimitiveStorageNeeded; /* Number of primitives that would have been written to the
stream output resource, if big enough */
} D3DDEVINFO_STREAMOUTPUTSTATS, *LPD3DDEVINFO_STREAMOUTPUTSTATS;
```

The data associated with this Query Type is D3DDEVINFO_STREAMOUTPUTSTATS. This structure contains statistics for monitoring the amount of data streamed out at the streamout stage of the Pipeline. Only complete primitives (e.g. points, lines or triangles) are Streamed Out, as counted by these stats. Should the primitive type change (e.g. lines to triangles), the counting is not adjusted in any way; the count is always total primitives, regardless of type. Naturally, though, only the difference between two independant statistic requests will provide meaningful information.

Note that with respect to the IA: adjacency vertices are counted. Partial primitives will be allowed to fall within range of values, similar to the way vertex caching behaves. So, when partial primitives are possible, statistics should fall between a pipeline that clips them as soon as possible (before even the IA counts them), or as late as possible (post clipper/ pre-PS). Stream Output and a NULL GS is flexible as to whether it actually causes GS invocations to occur or not.

VGT handles these STATS by sending increment signal for 64 bit counters NumPrimitivesWritten and PrimitiveStorageNeeded in CP for valid GS outputs. VGT also check if any of enabled streamout buffer is full (i.e. can't store a full primitive worth of data), if it is then only PrimitiveStorageNeeded is incremented otherwise both of the counters are increment.

3.15 DrawAuto (DrawOpaque)

DrawAuto is intended to consume data streamed out from prior draw packets. These prior draws will streamout vertex buffers arranged as lists. The IA is given a buffer size (BufferFilledSize) and the vertex stride. Starting at 0 the IA accumulates the stride to generate a unique index for each vertex. The primitive type is specified by the driver and should match the type that was streamed out of the prior packet. DrawAuto is enabled via the use_opaque bit in VGT_DRAW_INITIATOR and it was first implemented as required by the DX10 spec.

3.16 DX11/OpenGL4 Tessellation

As described in the overview here are the basic steps executed when tessellation is enabled.

- Generate LS wavefronts/vertices and send them to the SPI
- Upon completion of LS, generate HS wavefronts and send them to the SPI
- Upon completion of HS, retrieve tessellation factors from memory, and execute the fixed function tessellator stage.
- Create DS wavefronts/data from output of tessellator. Send the DS wavefronts to the SPI as either ES wavefronts (Geometry Shader enabled) or as Vertex Shader wavefronts (Geometry shader disabled)
- Optionally (if Geometry Shader enabled) create GS wavefronts/data and send them to the SPI
- Generate Primitive information and send it to the PA

3.16.1 Hardware Shader Names

LS : Vertex shader with the outputs to Local Data Store (LDS)
ES : Vertex or Domain shader with the outputs to Export memory
VS : Vertex shader with the outputs to Parameter Cache
DX11 API VS should be compiled as {LS,ES,VS}
DX11 API DS should be compiled as {ES, VS}

- If all shader stages are "bound", VS is used as LS, DS is used as ES;
- If only domain shader is not "bound", VS is used as ES;
- If only GS is not "bound", VS is used as LS, DS is used as VS;
- Etc.

3.16.2 Data Flow for Various Threads

1) SPI on receiving LS wave with "first-wave" will allocate LDS memory as required for LS/HS threadgroup
2) SPI will only allocate enough GPRs required per LS wave (earlier plan was to allocate num_LS_waves_per_threadgroup * num_gpr_per_ls_wave)
3) SPI will deallocate LS GPRs when a LS wave is done
4) All the LS waves are submitted to same SIMD until LS wave with "last_wave" is received", which tells SPI to start assigning next coming LS waves to SIMD which have enough resources for LS wave.
5) VGT sends VGT_LS_FLUSH event after each LS threadgroup
6) SX on receiving VGT_LS_FLUSH will send ls_flush_done signal to VGT
7) VGT on recieiving ls_flush_done will issue HS waves if
    a. There is enough space in tf_buffer for a HS threadgroup.
8) SPI on receiving the HS wave with "first_wave" set will allocate num_waves_per_hs_threadgroup*gprs_per_hs_wave GPRs. SPI will send HS waves of HS threadgroup to same SIMD where LS waves were issued.
   The "first_wave" helps SPI to determine which SIMD and "last_wave" tells SPI that next HS threadgroup could go to different SIMD. SPI also allocate barrier resource per HS threadgroup.
9) SPI sends "first_wave" and "last_wave" information to SQ
10) SQ sends SPI thread_done for hs_waves with "last_wave" information, which tells SPI to deallocate GPR and barrier resources.
11) VGT sends VGT_HS_FLUSH event at the end of HS threadgroup
12) SX on getting VGT_HS_flush event will send hs_flush_done signal to VGT
13) VGT on getting hs_flush_done signal will start reading tf_buffer for tessellation factors of patches in previously issued hs threadgroup.
14) Tessellation engine in VGT will start generating DS threads and waves.
15) SPI will send DS waves starting with "first_wave" set to same SIMD
16) SPI will allocate GPRs for a DS wave before issuing it to SIMD
17) SPI could send several such DS to same SIMD if enough GPRs available for issuing DS waves. Waves are submitted to same SIMD until "last_wave" is received.
18) GPRs for a DS wave are deallocated when DS is completed.
19) SQ sends wave_done for DS with Last_wave information to SPI on completion of DS waves.
20) If wave_done with "last_wave" set is recieived by SPI, the LDS is deallocated by SPI.

Note that the Tessellation Factor (TF) Buffer is located in cacheable memory. It is not in dedicated memory as shown in some of the following pictures.

3.16.3 LS Waves

1) VGT processes patch vertices as point and generates LS wavefronts with number of vertices for **VGT_LS_HS _CONFIG.NUM_PATCHES**. Reuse is turned off.
2) VGT computes rel_vertex_index in the same manner as auto_index, it will be just auto-index within a threadgroup
3) VGT sends ls_wave after each wavefront; VGT sets "first_wave" bit for the first wavefront of the threadgroup and "last_wave" bit for the last wavefront of the threadgroup
4) VGT sends VGT_ FLUSH_LS_OUTPUT event after the thread group

3.16.4 HS_Waves

1. VGT generates HS threads based on number of patches in a LS threadgroup at the same time when LS threadgroup is created. Such threadgroups are submitted to the same core as to where LS threadgroup were submitted.
2. VGT issues HS waves when
   a. it receives "sx_vgt_ls_flush_done" for LS thread, they are received in the same order as submitted for a given core
   b. The TF buffer has enough storage to store TF from HS outputs given by **VGT_TF_PARAM.TF_STRIDE**; The space in TF buffer is computed based on number of patches in threadgroup; VGT records such number along with core_id in a fifo.

   The base address for TF is computed by VGT, such address is the offset in the tf_buffer. The **VGT_TF_RING_SIZE** are specified as single context state registers.

3. VGT sets "last_wave" bit for the last wave for a given threadgroup. VGT also sets "first_wave" bit for first wavefront.
4. VGT sends VGT_SX_HS_FLUSH event after sending all the wave issues for a given threadgroup.

3.16.5 DS waves

1. On receiving the sx_vgt_hs_flush, TE in VGT reads the TF data as pointed by stored offset into TF buffer for a given patch. TE generate DS wavefront within a DS threadgroup (same as VS or ES of GS is "off" or "on"). It frees up TF buffer entries after being read for future thread allocation. VGT reads TF fifo information for each core in ping-pong manner.
2. VGT also sends "first_wave" bit in DS thread interface to specify the first thread of group of threads to be submitted to same SIMD. It sets "last_wave" bit for the last thread for a super-primitive
3. VGT sends following information with each thread: (u,v (it is assumed shader will calculate w for triangular domain), patch_id, rel_patch_id)
4. VGT sends ES_FLUSH event after DS threadgroup if GS is "on".

3.16.6 Driver/Compiler Tasks

3.16.6.1 Threadgroup size computation

The following computation is done by the compiler/driver to compute number of waves per threadgroup for LS/HS/DS

[EMBED Equation.3]

We can set num_patches to one by default. It is possible that if num_patches are too high, hardware will hang, as SPI will not able to allocate enough LDS. In the hardware we will have status bits on num_patches so that we can find the reason of hardware hang.

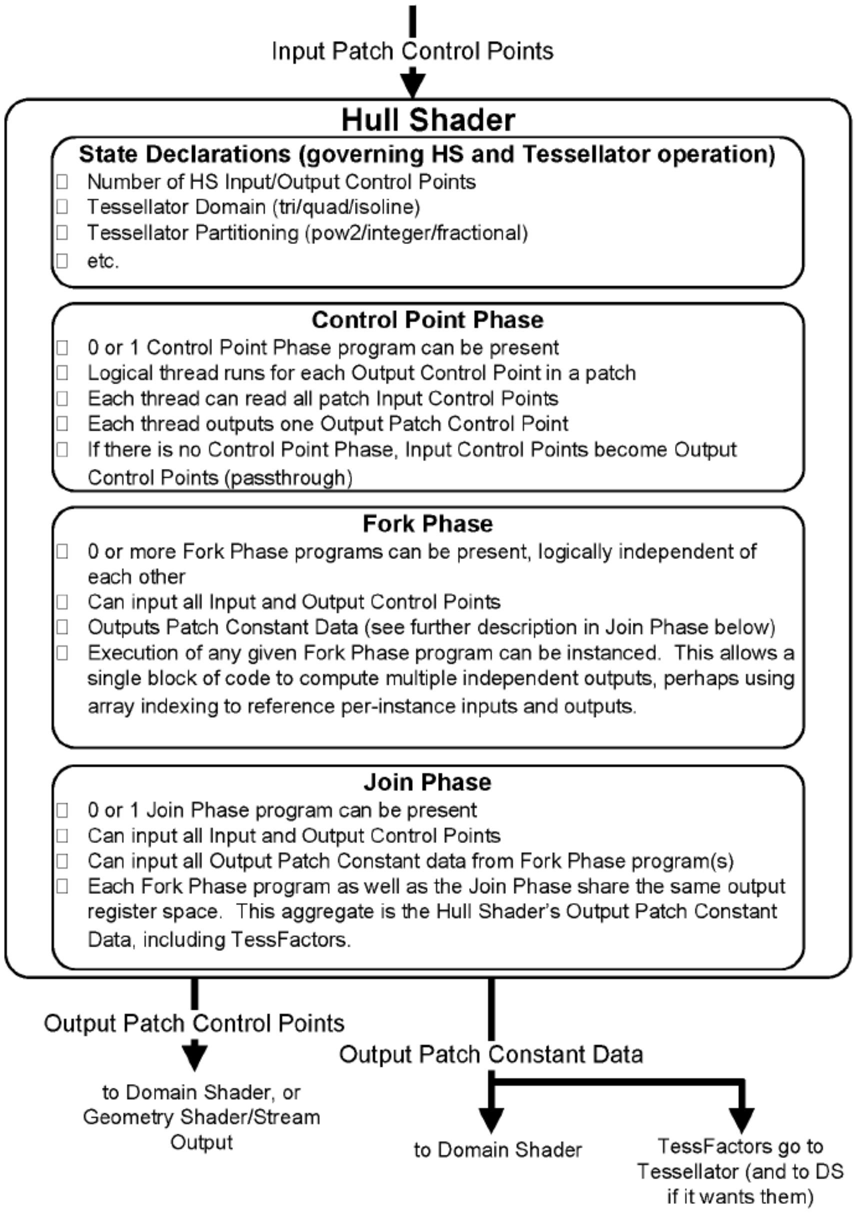
3.16.6.2 LS

r0.x = vertex index, r0.y=rel_auto-index with in threadgroup, r0.z=instance/step_rate0, r0.w=instance

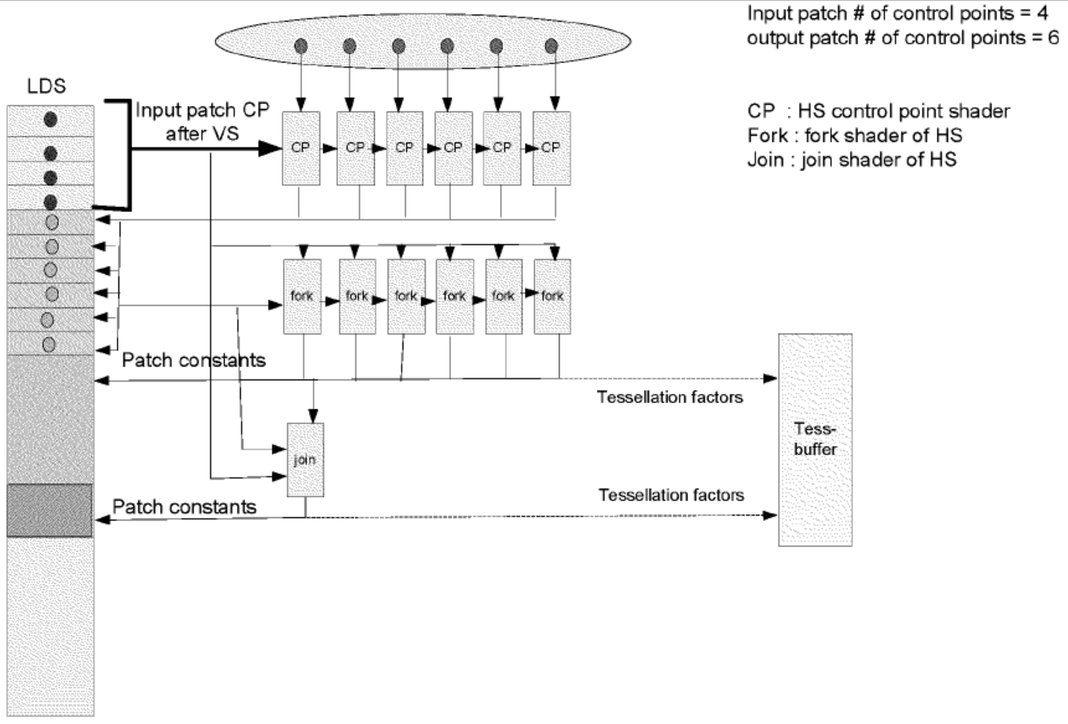
Fetch vertex data using r0.x as vertex index
VS
Export VS output to location r0.y*VGT_LS_SIZE.stride

3.16.6.3 HS

The following figure shows the DX11 breakdown of HS.


Input Patch Control Points
Hull Shader
State Declarations (governing HS and Tessellator operation)
Number of HS Input/Output Control Points
Tessellator Domain (tri/quad/isoline)
Tessellator Partitioning (pow2/integer/fractional)
etc.
Control Point Phase
0 or 1 Control Point Phase program can be present
Logical thread runs for each Output Control Point in a patch
Each thread can read all patch Input Control Points
Each thread outputs one Output Patch Control Point
If there is no Control Point Phase, Input Control Points become Output Control Points (passthrough)
Fork Phase
0 or more Fork Phase programs can be present, logically independent of each other
Can input all Input and Output Control Points
Outputs Patch Constant Data (see further description in Join Phase below)
Execution of any given Fork Phase program can be instanced. This allows a single block of code to compute multiple independent outputs, perhaps using array indexing to reference per-instance inputs and outputs.
Join Phase
0 or 1 Join Phase program can be present
Can input all Input and Output Control Points
Can input all Output Patch Constant data from Fork Phase program(s)
Each Fork Phase program as well as the Join Phase share the same output register space. This aggregate is the Hull Shader's Output Patch Constant Data, including TessFactors.
Output Patch Control Points
to Domain Shader, or Geometry Shader/Stream Output
Output Patch Constant Data
to Domain Shader
TessFactors go to Tessellator (and to DS if it wants them)

The compiler may compile HS shader as follows for maximizing parallelism.

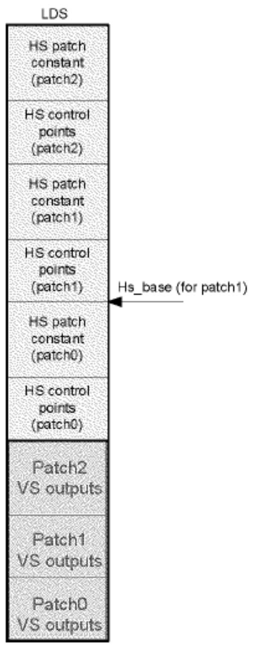


The following is the input to the GPRs.

VGPR0 = primitiveID or PatchID (32-bits)
VGPR1[7:0] = relative patch id with inthe threadgroup (0..255);
VGPR1[12:8] = patch output control point ID (0..31);
tf_base is loaded into a SGPR. See the SQ shader programming doc for details.
lds_offset (the base address in LDS) is specified by a SPI resource register

3.16.6.3.1 CP Phase

// Parallel CP generation
Compute offset where HS output will be written hs_base = **vgt_ls_hs_tb_config**.num_patches ***vgt_ls_size**.output_size + r0.y***vgt_ls_hs_alloc**.hs_output
Fetch input control point using r0.y* **vgt_ls_size**.output_size + input_controlpoint_id***vgt_ls_size**.stride
HS // Control point computation
Write to hs_base+ r0.x***vgt_hs_size**.stride
Barrier //make sure all the output control points are written to LDS


LDS
HS patch constant (patch2)
HS control points (patch2)
HS patch constant (patch1)
HS control points (patch1)
Hs_base (for patch1)
HS patch constant (patch0)
HS control points (patch0)
Patch2 VS outputs
Patch1 VS outputs
Patch0 VS outputs

3.16.6.3.2 Fork Phase

```
//patch constant data generation (Fork (n); given that p is number of HS output control points; each HS
CP will do f = ceil(n/p) forks instances)
For(i=0;i<f;i++){
   if(i*r0.x < n){
         Fetch a given HS output data from hs_base +out_contrl_point_id*vgt_hs_size.stride
         Fetch a given HS input data from r0.y* vgt_ls_size.output_size +
         input_controlpoint_id*vgt_ls_size.stride
         Compute patch constant data
         If patch constant data is not tessellation factor
               Write to hs_base + vgt_hs_size.output_size + const_id*vgt_hs_patch_const.const_stride
         Else // it is tessellation factor
                                                   Write to r0.y*vgt_tf_param.tf_stride + tf_num
               Write to hs_base + vgt_hs_size.output_size + const_id*vgt_hs_patch_const.const_stride
               // assumes 32-bit tessellation factor write
               // tessellation factors are also written as patch constant data
   }
}
```

3.16.6.3.3 Join Phase

```
//patch constant data generation (Join : done once per patch)
If(r0.x==0){
         Fetch a given HS output data from hs_base +  out_contrl_point_id*vgt_hs_size.stride
Fetch a given HS input data from r0.y* **vgt_ls_size**.output_size +
input_controlpoint_id***vgt_ls_size**.stride
Fetch is patch constant data from hs_base + **vgt_hs_size**.output_size + const_id*const_stride
Compute patch constant data
**If** patch constant data is not tessellation factor
    Write to hs_base + **vgt_hs_size**.output_size + const_id***vgt_hs_patch_const**.const_stride
**Else** // it is tessellation factor
                                        Write to r0.y***vgt_tf_param**.tf_stride + tf_num
     Write to hs_base + **vgt_hs_size**.output_size + const_id***vgt_hs_patch_const**.const_stride
    // assumes 32-bit tessellation factor write
    // tessellation factors are also written as patch constant data
}
```

Tessellation factors are written in the following order:

For isoline
TessFactor_U_LineDetail, TessFactor_V_LineDensity

For tri
tessFactor_Ueq0, tessFactor_Veq0, tessFactor_Weq0, insideTessFactor

For quad
tessFactor_Ueq0, tessFactor_Veq0, tessFactor_Ueq1, tessFactor_Veq1, insideTessFactor_U, insideTessFactor_V

3.16.6.4 DS

R0.x =patch_id/primitiveID
R0.y = rel_patch_id // within threadgroup
R0.z =u
R0.w = v
It is assumed lds_offset (the base address in LDS) is sent to SQ by SPI for reading/writing the LDS

// compiled shader
hs_base = **vgt_ls_hs_tb_config**.num_patches ***vgt_ls_size**.output_size + r0.w***vgt_ls_hs_alloc**.hs_output
Fetch patch data from LDS using hs_base + control_point_id*stride
DS
Export VS output to parameter cache or to ES ring

3.16.6.5 Linkage between different shaders

Different shaders will need to access several strides. It is expected that constant buffer will be used to use render state data in the shader.

3.16.7 Off-chip memory for tessellation

Initially the HS output is written to LDS, but as tessellation levels increase this becomes a bottleneck because it requires all DS waves to execute on the same compute unit. Writing the HS output to cache backed off chip memory allows DS waves to execute on any compute unit. The decision to write the output off chip is made dynamically in the HS.

There will be an off-chip LDS buffer, specified in 8K chunks, available for use by any thread group. The HS data besides writing to on-chip LDS will also write to off-chip LDS if the tessellation level is higher than a threshold.

Typically we envision this being a hybrid mode where some DS waves stay on chip so they can consume the LDS data and only waves above a threshold read from off chip memory. The threshold is defined by NUM_DS_WAVES. It's possible to set NUM_DS_WAVES to 0 resulting in all DS input data being fetched from off chip memory.

Knowing no DS waves will fetch from LDS allows us to free LDS space sooner. To implement this, there is a bit on the VGT_SPI_hswave interface called lshs_dealloc.

If num_ds_waves=0 and dynamic_hs=1.
The VGT will set the lshs_dealloc bit on the last hs wave of the threadgroup.
The SPI will deallocate the LDS space that was reserved (allocated during LS execution).

The lshs_dealloc bit is also present on the ES and VS interfaces (used depending on which was the DS). It is used to identify the last DS wave that went onchip and tell the SPI to deallocate the LDS. When using num_ds_waves=0, this bit will not be set for the threadgroup. All waves in the threadgroup will be marked as offchip.

If dynamic_hs =1 and num_ds_waves=0, then the shader must specify that the threadgroup is offchip.
If dynamic_hs=0 then the programming of num_ds_waves will be ignored and all waves execute onchip.

When completely null threadgroups are seen, there is a null ds wave sent which is tagged as both first and last (same as 9xx)
If the num_ds_waves was 0, this will be tagged offchip and the lshs_dealloc will be 0.
If the num_ds_waves was greater than 0, this will be tagged onchip and the lshs_dealloc will be 1

3.16.8 Tessellation Redistribution

Prior to gfx8, the tessellation solution involved the incoming packet being split up into primgroups which were sent to individual VGT units. The primgroups were made up of a number of predetermined input patches.

At the end of the HS stage, the user determines how much tessellation is necessary for each patch. Each patch can produce anywhere from 0 to over 8 thousand primitives. Since the amount of tessellation is not known at the time of primgroup creation, it is not possible to estimate the post tessellation workload that each VGT will have to handle. This can result in an unbalanced load that can cause duty cycling as well as degraded performance since a given Shader Engine may be overloaded leaving other SEs idle.

Beginning in gfx8 with distributed tessellation, the intention is to rebalance the work after the HS stage in order to generate new primgroups with post tessellation primitives. This enables the workload to be distributed more uniformly amongst the available SEs achieving overall higher performance.

The WD will receive feedback from the VGT when a DX11 pipeline is enabled with patch processing about the number of patch threadgroups issued to LS/HS for processing. After receiving HS threadgroup done signals from the SPIs, this data enables the WD to serialize and redistribute the patches across shader engines for a well-balanced load on the PA/SC distribution bus and fifos and shader system.

3.16.8.1 Threadgroup handling

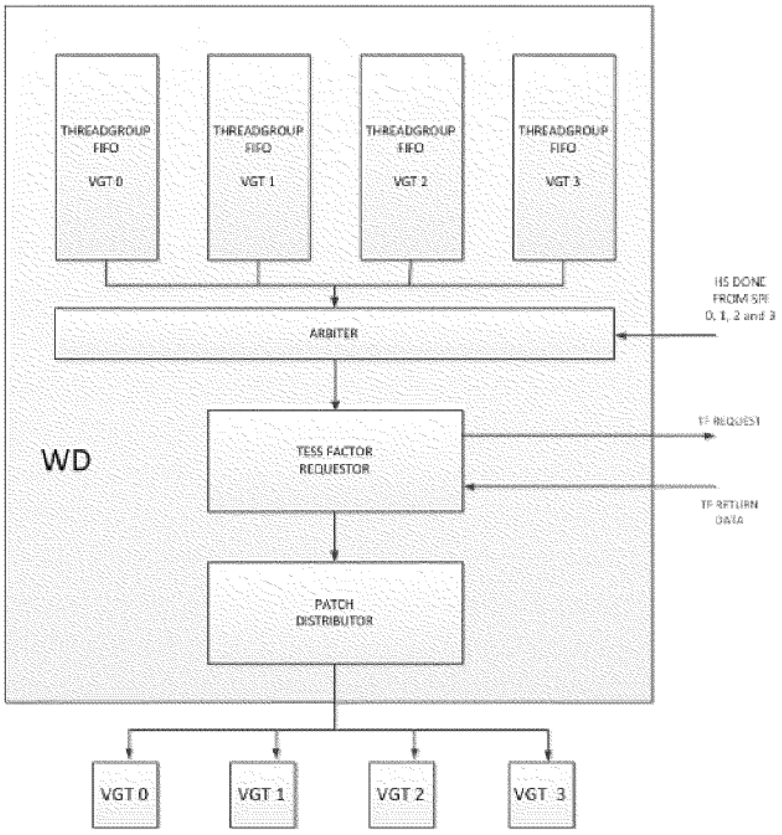
The VGT will now also send the WD threadgroup information which is used to put the data back in order and redistribute patches. The VGT_WD_thdgrp interface was added to carry this info.

At the end of the HS, the shader writes out the tessellation factors to memory. The shader also copies the HS output data to the HS offchip memory.

The SPI sends a per threadgroup HS_done signal to the WD when the all the HS waves of the threadgroup complete.

The change to a distributed tessellation model requires a centralized tessellation factor fetcher to be located in the work distributor (WD) block. Formerly the tessellation factor fetching was implemented in each VGT, such that the tessellation fetching rate scaled with the number of VGTs. Now, a single fetcher must be able to fetch at the same rate as previously provided by the multiple fetchers. The peak rate does not need to be any faster than a patch per cycle, as this is a system limit. The worst case patch is a quad patch and it requires 6 tessellation factors, so a design that can provide 6 TFs per cycle is sufficient. The memory system can supply 8 tessellation factors per cycle, so it is already sufficient to meet this goal.

3.16.8.2 WD Tessellation Redistribution



The WD needs to process threadgroups in the order they were issued originally by the WD/IA/VGT. There is a HS threadgroup done counter per VGT which is incremented when the respective HS_done is received from the SPI.

At the beginning of the packet, the VGT fifo with a threadgroup tagged first_primgroup will be processed first. This will ensure the order that the LS was launched in is maintained. After all threadgroups from the primgroup are popped off, the next_fe_id field will be used to determine which fifo to read from next. This is only valid on the last threadgroup of the primgroup.

After all factors for a threadgroup are fetched, the WD sends the respective VGT a signal to indicate the TFM can now be deallocated. This is a new interface, VGT_WD_thdgrp.

All the tessellation factors for the patches in an entire threadgroup are fetched from memory and the patches are then sent to the distribution logic.

The WD also inserts the VGT_id into the VGT fifos. This will be used to tag a new OFFCHIP_HS_DEALLOC event. This bit will propagate through the VGT and will be passed on to the SPI on the ES or VS wave (depending on which one was the DS). The SPI will use this to deallocate the offchip HS memory. The Resource Management section below gives more details. This event is not inserted when the legacy completely onchip tessellation mode is used i.e. dynamic_hs=0

3.16.8.3 Distribution Heuristics

The WD will distribute patches to the VGTs over the new WD_VGT_patch interface. The patches are distributed amongst the 4 VGT units based on heuristics detailed below. The last patch that goes to each VGT is tagged as EOPG. These are now new primgroups. The next_fe_id for this EOPG will be the next VGT that the distribution logic is going to use, the original pre-tessellation value has been dropped. The patch that had the original EOP will maintain the original next_fe_id value before distribution. This will let the SC be in sync while going in and out of distributed tessellation mode

A new field, DISTRIBUTION is added to the VGT_TF_PARAM register this controls the tessellation distribution mode

A new register called VGT_TESS_DISTRIBUTION is also added, this controls the fine grain distribution thresholds

3.16.8.3.1 Legacy Mode

No distribution is done, all the patches belonging to the original threadgroup are sent to the same VGT the LS/HS ran on.

This is enabled by setting VGT_TF_PARAM.DISTRIBUTION_MODE = NO_DIST

3.16.8.3.2 Distributing Patches

This is enabled by setting VGT_TF_PARAM.DISTRIBUTION_MODE = PATCHES

The WD processes patches from a threadgroup and keeps track of the minimum of the two inside tessellation factors (for quad). This number is accumulated for each patch. Multiple patches are sent to the same VGT until this number is greater than or equal to a user programmed threshold. Triangle tessellation keeps track of just the inside factor and isoline tracks the minimum of the two tessellation factors. Only the integer part of the accumulation value will be used

These thresholds are defined in the following new register and contain different values per domain type. These are expected to set to some value that works well for all cases and not changed often (if at all). The threshold values are expected to be integers.

VGT_TESS_DISTRIBUTION.ACCUM_ISOLINE

VGT_TESS_DISTRIBUTION.ACCUM_TRI

VGT_TESS_DISTRIBUTION.ACCUM_QUAD

3.16.8.3.3 Splitting Patches – Donut Distribution

This is enabled by setting VGT_TF_PARAM.DISTRIBUTION_MODE = DONUTS.

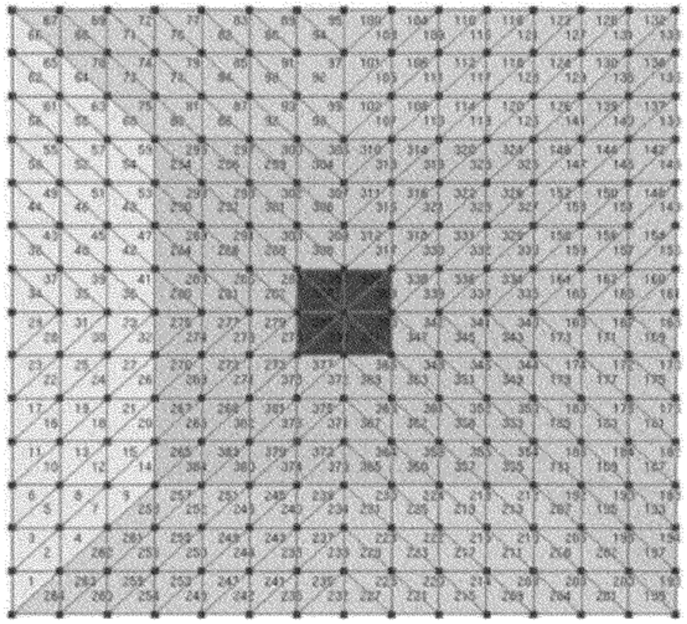
This mode also enables the PATCHES mode. In addition to this, individual patches can be split up amongst the VGTs. When the minimum of the inside factors (integer part) for a single patch is greater than or equal to a user controlled value, the patch is broken down into donuts. Each VGT will process a single donut.

If the number of donuts is NOT a multiple of the number of VGTs present, the final donut switches to the next VGT. If not it doesn't increment the VGT_id, this helps to balance the workload. If 4 VGTs are present, all even number of donuts will NOT switch.

The threshold value is contained in VGT_TESS_DISTRIBUTION.DONUT_SPLIT. There is only one value irrespective of the tessellation type. The threshold value are expected to be integers.

When processing isolines, the distribution is not in the form of donuts but as a group of individual lines, the number is defined by the register above.

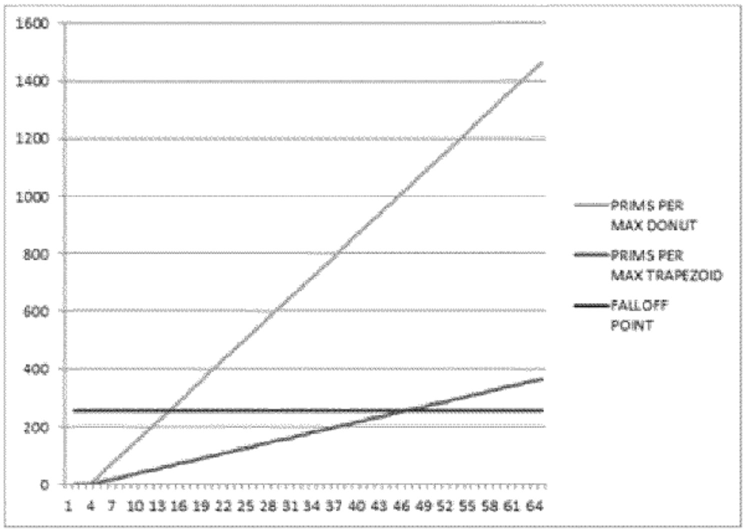
The picture below illustrates how a quad domain patch is broken up into donuts.



The outermost donut also shows the 4 component trapezoids. Each donut will be sent to a different VGT as described above. There are three donuts in the picture above (multi-colored, gray and red), each donut consists of 3 rings.

3.16.8.3.4 Splitting Patches – Trapezoid Distribution

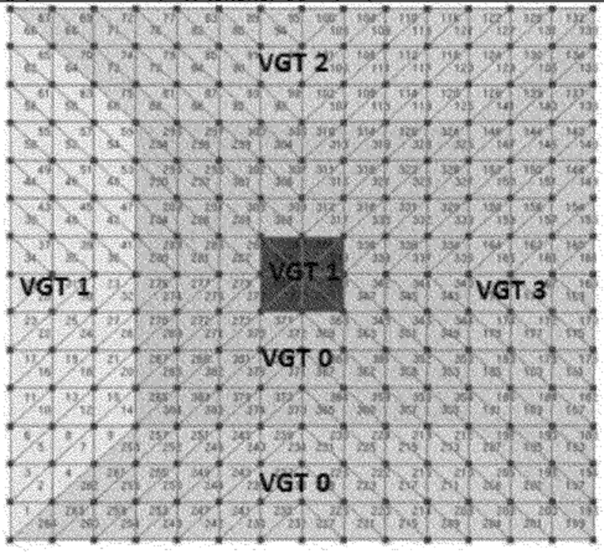
At higher tessellation levels, the number of primitives even within a single donut can be very large and may lead to duty cycling due to limited PA-SC FIFO space. The chart below shows the worst case scenario which causes a performance drop beyond 256 primitives to a single VGT (not the typical value, typically this number will be larger based on the distribution of primitives to different SCs)



Trapezoid Tessellation distribution extension further adds the ability to split up donuts into trapezoids which will allow higher tessellation factors to be handled at rate even in worst case situations.

Each donut is made up of 4 trapezoids (when in quad domain and 3 in tri domain). Trapezoids for a single donut are shown in yellow, green, orange and blue in the diagram at the beginning of this section. Each one will be sent to a different VGT. At a certain point there will not be enough primitives in the trapezoids and at this point the distribution of that patch will scale back to entire donuts.

In the picture shown, the distribution for a patch is shown assuming there are 4 VGTs present. The outer most donut gets broken down into trapezoids, the 2 interior donuts stay untouched (based on programmable thresholds)



This mode is enabled by setting VGT_TF_PARAM.DISTRIBUTION_MODE = TRAPEZOID. Setting this will also enable donut and patch tessellation distribution modes. Patches will still need to satisfy the respective thresholds as described earlier.

VGT_TESS_DISTRIBUTION.TRAP_SPLIT will contain an integer value that will be compared against the number of donuts in a patch. Any donut above this threshold will be broken into trapezoids. This value will be ignored if the DONUT_SPLIT threshold is not satisfied (a patch cannot be broken into trapezoids unless it was also going to create donuts). Note that the value in this register is in number of donuts *not* tessellation factor like all the others.

The TRAP_SPLIT register will be restricted to NOT allow a value of 1 (the hardware will force values of 0 and 1 to be bumped up to 2). This will mean the special case interior donut is never broken into trapezoids and will make it easier for validation (there should also be no need to break up the smallest donut into trapezoids). In production this threshold value is expected to be set between 3 and 4

Trapezoidal distribution will *not* be supported for isoline tessellation. This mode is implemented by processing 3 lines as a single donut and the number of primitives on 3 lines will not going to have performance issues.

3.16.8.4 DS Operation

The SPI will launch the DS waves on any available compute unit. The HS output data is fetched from offchip memory and used for barycentric interpolation. The DS waves of a given threadgroup can be launched on a SE that the original threadgroup did NOT start on. The SPI manages the offchip HS memory per SE and the DS waves will need extra information to let them fetch from the correct SE.

The WD will include a per SE offchip_lds counter that increments for each offchip tessellation threadgroup that gets issued on that SE. This counter will be passed from the WD to the VGT for each patch and then be sent back to the SPI on the DS (ES or VS) wave interface. This will allow the SPI to use this value to determine how to fetch from the offchip HS memory.

VGT_HS_OFFCHIP_PARAM.OFFCHIP_BUFFERING specifies the current number of offchip buffers and the offchip_lds value should reset to 0 for each SE whenever that register value changes (it does not reset if the register is written but the value does not change). The OFFCHIP_BUFFERING value is biased by 1 with a max

of 128 per SE. The programmed value in the register is divided between the number of SE in the variant, regardless of front-end harvesting, and each SEs value should count from 0 to ((OFFCHIP_BUFFERING+1) / NUM_SE) – 1. The number of offchip buffers must be an integral multiple of the NUM_SE
The existing event_id field on both the ES and VS interfaces will be expanded to 7 bits and will hold the offchip_lds counter value.

A new 2 bit field will be added to both interfaces called parent_se and this will hold the value of the parent SE of the threadgroup.

The SPI uses either the OFFCHIP_HS_DEALLLOC event or the last wave and lshs_dealloc bits to deallocate the offchip HS memory.

The table below details when the first/last wave flags are issued on the DS waves as well as when the OFFCHIP_HS_DEALLOC event is inserted

	DYNAMIC HS	**NUM DS WAVES**	**DEALLOC EVENT**	**FIRST/LAST WAVE FALGS**
Legacy all onchip	0	X	NO	YES
Legacy Dynamic HS	1	> 0	YES	YES
Legacy all offchip no distribution	1	0	YES	NO
All offchip + distribution	1	0	YES	NO

3.16.8.5 Resource Management

The table below summarizes the resources used and their management schemes

RESOURCE	**ALLOCATION**	**DEALLOCATION**
LDS	SPI allocates memory when it sees a LS wave with the first_wave bit set. This indicates this is the first LS wave of a threadgroup	The SPI deallocates the LDS space for the threadgroup when it sees an HS wave with the lshs_dealloc bit set. This is the last HS wave of the threadgroup
Offchip HS Memory	SPI allocates memory when it sees a LS wave with the first_wave bit set. This indicates this is the first LS wave of a threadgroup	After the WD fetches all the tess factors for a threadgroup, it will insert a new event OFFCHIP_HS_DEALLOC into all the VGTs. This event will be seen by all the SPIs and they will sync up. The event will only be seen on the ES or VS wave depending on which one was the DS. The VGT_id will be available on the new parent_se field of the DS wave.
Tess Factor Memory	The HSM in the VGT block will check for space and allocate space before launching any HS waves of a threadgroup. Space is allocated for the entire threadgroup.	The WD will send a deallocate signal after all the tess factors of a threadgroup are fetched. It will send the dealloc to the VGT which processed the original threadgroup. The deallocation is done by the VGT HS block.

3.17 Handling oViewport and oRenderTarget (array index)

The geometry shader can output oViewport and oRenderTarget. oViewport selects a viewport matrix for a given GS output primitive and oRendertarget specifies to which render target the primitive needs to be rendered. Geometry shader will output oViewport and oRendertarget with each vertex as part of second position for a given vertex.

If oViewport is used in GS and is not consistent for all vertices of a strip, the VGT needs to expand GS output primitives as a list. To output lists set VGT_REUSE_OFF to turn off reuse. The PA will get viewport index information from the provoking vertex and pass it along for other vertices of the primitive.

Fast path scenarios can also be used to output oViewport and oRenderTarget.

There is an exception allowing reuse to be enabled with oViewport. If all emits from a GS are to the same viewport array index reuse can be enabled as the PA is assured to have the correct value. In addition the PA_CL_CLIP_CNTL.VTE_VPORT_PROVOKE_DISABLE register field needs to be set to 1.

DX 11.3 adds functionality to pass the viewport array index when the GS stage is disabled. In this case, the exception allowing reuse to be enabled is if all primitives within an instance are being rendered to the same viewport array index. This can work because the hardware guarantees that there is no reuse across instances and primitives do not span instance boundaries.

The hardware also guarantees there's no reuse between the output of two patches. The driver/compiler may be able to use this behavior when determining if reuse needs to be disabled.

3.18 D3DQUERYTYPE_PIPELINESTATS

```
typedef struct D3DDEVINFO_PIPELINESTATS {
  UINT64 IAVertices; /* Number of vertices IA generated (not subtracting any caching) */
  UINT64 IAPrimitives; /* Number of primitives IA generated */
  UINT64 VSInvocations; /* Number of times Vertex Shader stage is executed */
  UINT64 GSInvocations; /* Number of times GS is executed */
  UINT64 GSPrimitives; /* Number of primitives GS generated */
  UINT64 CInvocations; /* Number of times clipper executed */
  UINT64 CPrimitives; /* Number of primitives clipper generated */
  UINT64 PSInvocations; /* Number of times PS is executed */
} D3DDEVINFO_PIPELINESTATS, *LPD3DDEVINFO_PIPELINESTATS;
```

VGT handles IAPtimitives, VSinvocations, GSinvocations, counts. These are handled by sending increment signals to CP which has 64 bit counters for such pipelinestats.

The data associated with this Query Type is D3DDEVINFO_PIPELINESTATS. This structure contains statistics for each stage of the graphics Pipeline. For each stage, the value for number of invocations must fall between two numbers: infinite cache & no cache. Here's some examples of the interaction between the IAVertices, IAPrimitives, and VSInvocations with respect to Post-VS caching

- Draw Indexed Tri Strip of 4 prims (with all indices the same value): valid IAVertices only 6, valid IAPrimitives only 4, valid VSInvocations 1 - 12.
- Draw Indexed Tri List of 4 prims (with all indices the same value): valid IAVertices only 12, valid IAPrimitives only 4, valid VSInvocations 1 - 12.
- Draw Tri Strip of 4 prims: valid IAVertices only 12, valid IAPrimitives only 4, valid VSInvocations 6 - 12
- Draw Tri List of 4 prims: valid IAVertices only 12, valid IAPrimitives only 4, valid VSInvocations only 12

Partial primitives will be allowed to fall within range of values, similar to the way vertex caching behaves. So, when partial primitives are possible, statistics should fall between a pipeline that clips them as soon as possible (before even the IA counts them), or as late as possible (post clipper/ pre-PS). Stream Output and a NULL GS is flexible as to whether it actually causes GS invocations to occur or not.

With respect to IAVertices and VSInvocations, adjacent vertex processing may be optimized out if the GS does not declare the adjacency vertices as inputs to the GS. So, when the GS does not declare adjacent vertices as inputs, IAVertices and VSInvocations may or may not reflect the work implied by the adjacent vertices. If the GS declares adjacent vertices, then the IAVertices should include the adjacent vertices (with no regard to any post-VS caching); and VSInvocations should include the adjacent vertices (along with any effects of post-VS caching).

3.19 DMA Max Size Index Clamping

VGT_DMA_MAX_SIZE specifies maximum number of valid indices which is independent of VGT_DMA_SIZE which specifies the size of the entire index buffer. The requirement is that the DMA engine needs to return zeroes for any index values that are greater than MAX_SIZE and less than DMA_SIZE.

In R10xx, we continue to fetch indices after DMA_MAX_SIZE as long as they are less than DMA_SIZE. The DMA engine then clamps the out of range indices to zero. This still causes issues since the out of range indices could generate page faults.

From 11xx onwards, the DMA engine will not issue DMA fetches for out of range indices and will continue to return zeros for these.

The WD block will be responsible for implementing this feature. DMA requests that are within the valid range will be handled normally. A special request will be issued for the out of range data. This will be tagged with information to inform the DMA engine to not request this data but still insert 0x0 for each index required into the data stream to the Grouper.

This example shows how this will be implemented

Register Programming
```
base address = 0xA3406D2294   (40 bit address)
index type   = 1         (32 bit indices)
dma size     = 16
max dma Size = 7
```

DMA Basics
The DMA requests are made for 256 bytes of data.
This data is returned in 8 chunks of 32 bytes each.
The base address for the fetch must be aligned to 256 bytes.

Here is how the data looks like in memory. An entire request of 256 bytes is shown, divided into 8 chunks of 32 bytes (256 bits)

[EMBED Visio.Drawing.11]
This shows the data in memory in chunks 4, 5 and 6

```
6  XX XX XX 10 0F 0E 0D 0C    5 out of range indices, followed by 3 don't
care indices
5  0B 0A 09 08 07 06 05 04    5 valid indices, followed by 3 out of range
indices
```

```
4  03 02 01 XX XX XX XX XX     5 don't care values, followed by 3 valid
indices
```

The first fetch will have number of indices set to 8, and 2 mask bits will be set indicating 512 total bits are to be fetched.

```
base address = 0xA3406D2294
mask = 0x30
```

This fetch will return the following data

```
0B 0A 09 08 07 06 05 04
03 02 01 XX XX XX XX XX
```

At this point the DMA engine will detect that there are 3 out of bound indices and zero these out when the data is sent to the Grouper.

```
00 00 00 08 07 06 05 04
03 02 01 XX XX XX XX XX
```

There will be an extra out of range fetch issued.

This fetch will have number of indices set to 5 (3 out of range indices were accounted for earlier)

```
base address = 0xA3406D22C0
out_of_range = 0x1
```

This will be a special type of fetch that will not cause any TC request.

The DMA engine will insert 0x0 into the index stream going to the Grouper.

```
00 00 00 00 00 00 00 00
```

Even though only 5 0x0s were necessary, the DMA engine will insert them at a granularity of 256 bits.

The Grouper will use the 5 indices that it cares about and at that point the end of packet will cause the overhanging 3 0x0s to be discarded in the Shifter.

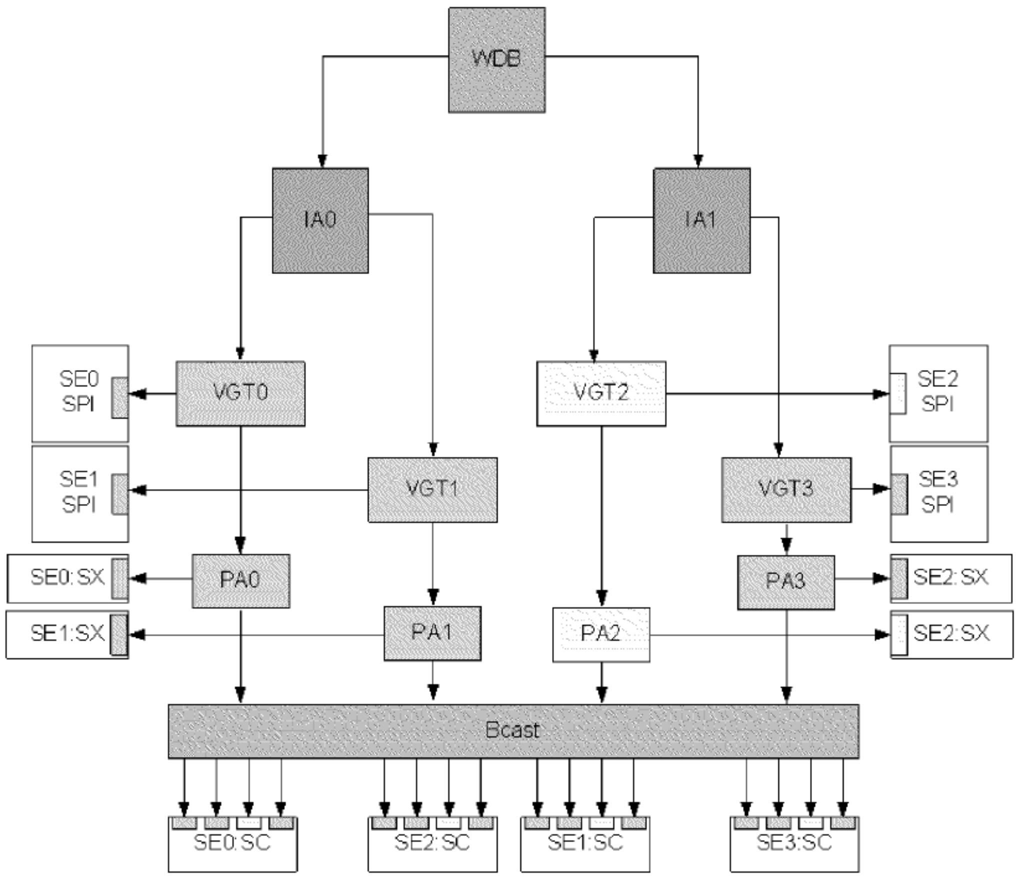
Note that the DMA engine will always insert extra 0x0s aligned to 256 bits for all fetches (not just out_of_range ones). This is new for 11xx.

This is the final stream of indies that goes to the Grouper

```
00 00 00 00 00 00 00 00
00 00 00 08 07 06 05 04
03 02 01 XX XX XX XX XX
```

3.20 Front-end Harvesting (GFXIP 7.2)

Frontend harvesting allows VGT-PA pairs to be harvested out of the design for lower SKUs.



- There will be a 4-bit e-fuse backed register CC_GC_PRIM_CONFIG that enables up to 3 VGT-PA pairs along with respective IA if both VGT-PAs are disabled.
- This register will also have a GC_USER_PRIM_CONFIG user version to allow non-e-fuse based configuration.
- Most of the VGT-PAs will be completely harvested, the clocks will be turned off at the MGCG.
- In addition, the sub-blocks in the SPI, SX (position buffer), and SC that receive work from VGT/PA can opportunistically be harvested/disabled as well.
- All control signals should be masked at the destination block using the registers to avoid manufacturing defects in the harvested blocks from causing a malfunction.
- The parameter Cache could be either harvested per SE, or folded around to improve performance.
- The WD block will know if an IA only has one active VGT and perform some load balancing between them, i.e. adjust work group size to 1x or 2x of prim group based on how many VGT are enabled for a given IA. If both VGTs for a given IA are disabled, that IA will not be sent any workgroup.
- IA will send primitives to only enabled its enabled VGTs and disabled VGT clocks will be turned off except for streamout synchronization purpose.

- CP will only get streamout and query count updates for enabled VGT so CP should also look into harvesting registers to see which IA/VGTs are enabled.

3.21 High Priority Graphics

The intention of this feature is to be able to process higher priority graphics (HP) work dispatched by the CP or the driver. This type of work will be able to jump ahead of already buffered normal low priority draw calls (LP). The WD/IA will be able to switch out low priority graphics workloads at a ***packet boundary***. Any low priority draw packet that has been started when a high priority draw call is seen will finish in its entirety before the switch happens. Either of the two input pipes into WD can be selected to be higher priority based on register programming.

[EMBED Visio.Drawing.11]

The diagram above shows the main components in the WD and IA used to handle switching in and out of high priority graphics. There are separate input FIFOs to store the incoming Draw Commands and DMA index fetch requests for each priority queue. There are also separate DMA return index FIFOs for both. There is a DMA requester which arbitrates between the low and high priority DMA fetches and it issues DMA requests over a single memory controller interface. The darker Distributer blocks shown in the WD exist only in projects which support 4x prim rate. In 2x prim rate parts the DMA and draw calls are passed through the WD to the IA without splitting.

3.21.1 Selecting Pipe Priority

Programming the HP_PIPE_SELECT field of the new register GFX_PIPE_PRIORITY determines which pipe has higher priority during arbitration.

3.21.2 Dispatch Initiators

The ADC unit has been moved to the CP, dispatch initiators are no longer processed.

3.21.3 Draw Command Arbitration

Draw commands received by the WD are routed into separate draw initiator FIFOs, one for each input pipe. After each draw completes, the draw command arbiter decides which draw command FIFO to process next. The GFX_PIPE_PRIORITY register will determine which pipe is HP. Note for 4x prim rate parts that will see sub-draw calls, all sub draws from a packet will be completed before arbitrating the next draw call. Draw command arbitration is located in the WD instead of the IA because in parts with 4 SEs, there are 2 IAs and the arbiters can get out of sync.

Begin processing ***High Priority*** draw command if ***any*** of these conditions are true

- A draw command is pending in the HP queue

Else begin processing the next LP draw command.

3.21.4 DMA Request Arbitration

There are separate DMA requestors for the HP and LP. This allows arbitration on a per-request granularity. Arbitration logic decides which DMA request is processed and this is sent by the DMA engine to the TC. Return index data from the TC is also stored into two separate data FIFOs, one for each priority. Each IA has independent arbiters since there are separate return data fifos in each IA.

Service ***Low Priority*** DMA request if ***any*** of these conditions are true

- No HP DMA requests are present and a LP DMA request is present
- A HP DMA request is present, but a LP DMA fetch is already underway for a LP draw command ***currently being processed***

Else service High Priority DMA request

Note: At any time if the return data fifo for the requestor currently being serviced gets full, that requestor will stop creating requests and any pending requests from the other queue will be serviced. This applies for both LP and HP.

3.21.5 Register Controlled Hysteresis

It is possible that the CP/driver would like to insert consecutive high priority draw calls with some bubbles in between them. To enable the WD to stay in high priority mode, the CP (or the driver) will insert a register write to inform the WD a series of high priority work is going to begin. Another scenario is if there are bubbles in the pipe that cause the arbiter to keep switching between LP and HP.

To handle these situations a new register GFX_PIPE_CONTROL has been added. This has a HYSTERESIS_CNT field that operates as follows. After the last HP packet has been processed, a counter is loaded with this value and counts down. Once the count is 0, the next arbitration is allowed to occur.

A count of 0 implies immediate switching is allowed

A max value of 0x1fff will prevent the other pipe from having access to the hardware – a lockout.

3.21.6 Reserving contexts (states) for HP draws

In order to ensure that HP draw calls always have a context (state) available, the CP will implement the following protocol.

- The CP will increment an up-down counter for every CONTEXT_DONE event sent to the WD on each pipe.

- The WD is responsible for sending the CP a signal on a new ***WD_CP_context_done*** interface when the event is popped off the draw FIFO for each pipe. Bit 0 set indicates pipe0 and bit 1 set indicated pipe1. The CP will decrement its counters on receiving this signal. Note that the CONTEXT_DONE event will still take time to propagate through the pipeline but is guaranteed to return to complete and is not stuck in the LP FIFO.

- The CP uses a programmable register to determine how many contexts (states) the LP draw call is allowed to have outstanding.

3.21.7 Pipe Id Propagation

A pipe_id bit will be provided from the IA to the CP for the query counts that it handles. The CP will maintain two copies of the counts and report them back individually to the driver.

A bit is also propagated from the IA to the downstream VGT(s) which indicates the pipe id the draw call belongs to. This bit is propagated by the VGT to its downstream blocks for tracking purposes (simulation only). All relevant downstream blocks will determine if the draw call being processed is a high priority one by looking up state data.

3.21.8 Handling Streamout

In 10xx all the streamout registers are multi-context with the exception of the ***VGT_STRMOUT_BUFFER_FILLED_SIZE_**** registers. To support cases where LP and HP draw calls with streamout need to continue where they left off these sets of registers will be duplicated. This will allow a separate set of counters for LP and HP streamout.

All internal registers that are reset by a VGT_STRMOUT_RESET event need to have LP and HP versions. This includes the dword written counts and streamout index values.

The STREAMOUT_RESET event will carry state that will enable the VGT to look up whether LP or HP was used.

The SO_VGTSTREAMOUT_FLUSH event will also have state date to indicate which pipe id is to be flushed.

3.21.9 Context Suspend

This feature is added in case it's needed for a bug workaround. It's controlled by GFX_PIPE_CONTROL.CONTEXT_SUSPEND.

The WD will insert a CONTEXT_SUSPEND event between all transitions from low and high priority graphics. This event will be used by the SC to circumvent some timing issues. The SC will also use (CONTEXT_DONE || CONTEXT_SUSPEND) to create the SC_SEND_DB_VPZ event to update the DBs copy of the viewport array Zmin and Zmax.

Adding the CONTEXT_SUSPEND allows the SC to not keep track of state_id internally and eases the validation effort.

3.22 Graphics Pipeline Reset (GFXIP 7.2)

3.22.1 Problem Definition

The idea is to discard the draw packets for a given VMID in the GPU with minimal change such that we don't process full draw call. But this is done independent of compute packets. Therefore, we can't use hardware reset mechanism. Any amplifying point in the graphics pipeline should de-amplifying by killing or not generating primitives, killing or not generating pixels while keeping intact the functionality of graphics such as shader stages completion, position/parameter cache allocation/deallocation, end of packet etc.

CP enables reset by generating a reset signal and specifying a VMID through a register. CP does not change the value this register until it gets the context done for a given VMID packets. At soon as, CP sets register to reset the gfx pipeline for a given VMID, it will stop sending any draw packets/ dma requests to WD for a given VMID. It is possible CP sends dma requests for a given VMID, but not send draw for reset VMID, but vice versa is not possible. WD/IA/VGT on getting reset signal (through register) will not amplify any GS output, tessellate patches and will terminate a current draw call at a prim group boundary for a given VMID. PA will cull all the primitives and deallocate position cache. SC will kill all the pixels and deallocate parameter caches. CB/DB will flow until all the pending pixels works are completed.

On receiving reset, SQ will enable shaders to jump to a exception code which will enable shader code to finish gracefully. For example, for VS, it jump to code where NaN is written to position and shader is completed. LS /GS/PS will jump to code to end the shader. HS will jump to code with a instruction signaling completion of HS code to SQ (I think this is special instruction which is used today to signal HS completion).

SPI will issue all the pending waves for all the shader type and they will all be gracefully completed with proper LS_done, GS_done, HS_done, VS_done signals to VGT. GS shader will jump to code as well so it does not need to do all the emits.

3.22.2 Implementation Details

3.22.2.1 WD impact

On getting reset signal for a given VMID, WD will stop sending workgroup for reset VMID but will keep sending other VMIDs primgroups and events. It will also stop sending any more dma_requests as well for reset VMID. If WD is in the middle of sending dma request and/or draw for reset VMID, WD will mark that workgroup as the last workgroup and send workgroup with eop. In case of draw opaque/ draw auto, the current prim group belongs to reset VMID, the work group with start index is sent and this prim group is sent as last workgroup with eop. Note that other IA will get null packet with eop.

In case WD_switch_on_eop is on (i.e. workgroup size = draw call), the full draw packet if in the middle of being sent, the full packet is sent.

We don't see a case there it is possible WD could send draw workgroup for reset VMID but dropped corresponding dma_request workgroup. It is however possible, WD could send dma_request workgroup for reset VMID but does not send draw workgroup for reset VMID.

For 4SE Systems, the WD Shunt point must be disabled by setting VGT_RESET_DEBUG.WD_DISABLE to 1. This is needed because the DMAs can come before the corresponding draws and the DMAs and draws have independent IA switching logic in the WD. If a draw is killed before all the workgroups are sent, the next IA logic can get out of sync for the following draws and DMAs.

3.22.2.2 IA impact

IA processes all workgroups submitted to it.

- On the DMA side, all the dma fetches will be tagged with VMID so that return DMA data also stored with VMID in the DMA fifo.
- DMA unit will looks for any new dma requests and will drop those requests for reset VMID.
- If a draw call for reset VMID is processed, the grouper gets invalid signal for indices of draw call until all the indices in DMA fifo for reset VMID are dropped. O
- nce DMA unit does not have any indices, fetch request and dma requests for reset VMID, DMA unit send a req_empty signal to grouper indicating that there is no more indices, requests for given reset VMID.
- At that point, if draw call for reset VMID was being processed, the indices are fetched as min_index and such primitive is marked as last primitive for draw call, i.e. eopg is sent with this primitive.

- At this point grouper can move to next draw call or event.
- For draw auto and opaque call previous auto index value is repeated for current processed primitive and current primitive is market as last with primitive for a prim group through eopg.

3.22.2.3 VGT impact

All the shader stages returns proper ls/hs/es/gs/vs done signals to VGT including emits and cut. The HS unit on receiving LS done will issue HS waves. On receiving HS done, tessellation factor unit will issue tessellation factor request, but will assume returned value to be NaN for reset VMID. So HS unit can drop patches without generating any vertices. The tessellation factor memory will be deallocated as tessellation factors are read.

The GS waves are issues when ES done signals are received and on GS done signal, GoG block start reading cut memory but will treat all the counters as zero and will not issue any vertices/primitives for reset VMID. The cut memory get deallocated as counters are read.

One getting the packet with eop, the output unit will accumulate all the deallocation (as done today, so no change) and send eop to PA.

3.23 Streaming Performance Counters

Refer to the "Streaming Performance Counters" document for a detailed explanation.

The current performance monitoring mechanisms are coarse and lack the ability to sample the counters frequently or at a finer granularity. The proposed enhancement is to address this limitation by using as much of the current hardware as possible.

The following performance counters were added.

- Vgt_pa_clipp_valid_prim is the number of valid primitives sent to the PA.
- Reused_es_indices are the number of hits in the Reuse Check Module (RCM) used for the ES hardware stage.
- Vs_cache_hits are the number of parameter cache (PC) hits when the hardware VS writes to the PC.
- Gs_cache_hits are the number of PC hits for the GS. Relevant for a merge case like scenario A.
- Ds_cache_hits are the number of PC hits when the hardware VS is the API DS.
- Total_cache_hits is the sum of vs_cache_hits, gs_cache_hits, ds_cache_hits.
- Strmout_stalled is the number of cycles spent waiting for a streamout sync from another SE.
- Ds_prims is the number of primitives output from the DX11 tessellator.

#case Thebe

3.24 Dispatch Draw

The dispatch draw feature is intended for use in cases where primitive intensive geometry is first processed by a compute shader to eliminate culled primitives etc. The resulting indices that pass the compute shader are then rendered though the normal graphics pipeline. In order to keep this process tightly coupled, various blocks in the design provide sync mechanisms to enable small portions of the dispatch (compute shader) to run and the output can be consumed by the second stage. The main advantage being there is no need to process the entire dispatch and wait before issuing the draw call. Refer to the main dispatch draw spec (//gfxip/gcB/doc/design/arch/Dispatch Draw.docx) for more details on the feature. This document will focus on the WD/IA/VGT changes necessary to implement this feature.

There are some restrictions on the draw portion of the dispatch draw

- Only Dx9 style draw calls with the VS-PS path will be supported
- Any instancing is unrolled by the user. The draw received by the WD will only have 1 instance
- Streamout is not supported because the dealloc vertex can't be streamed out and we don't give a mask to the VS indicating which verts to streamout.
- Dispatch draws cannot be tagged as High Priority Graphics
- Only list primitive types will be supported (point, line and triangle)
- VGT_DMA_MAX_SIZE has to be programmed equal to the VGT_DMA_SIZE for each sub draw call
- Each sub draw must have the number of indices be an integral multiple of the indices needed to form a primitive (list)
- If primitive id is expected to be correct, each subdraw must be equal to 2*primgroup size
- The match index value cannot show up in the application's index buffer so if the vertex buffer is larger than the match index Dispatch Draw cannot be used. i.e. if the match index is 0xFC000000 the vertex buffer cannot exceed that many (> 4 billion) locations which is > 66 GB for a vertex stride of 16 bytes.

3.24.1 Global Wave Id

The VGT needs to provide a global wave id for each VS wave. In order to maintain a global wave id, the VGT units need to sync up at EOPG. A new interface called the VGT_VGT_wave is added to allow this operation. It carries 12 bits containing the wave id between the VGTs.

The global wave id resets to 0 and increments for each wave. It rolls over at the value specified by the VGT_VS_MAX_WAVE_ID register. The register value is biased by 1, so wave id values will range from 0 and max_wave_id. The global wave id is reset to 0 only when this register is written.

The global wave id is incremented when the first vertex is created. This means if waves are combined across primgroups, the wave id will have been allocated at the beginning of the wave.

3.24.2 DMA Ring Wrapping

There is a ring buffer that stores the output of the dispatch portion of the call. The WD will get sub DMA and DRAW calls which fetch indices from the ring for the draw portion. The main difference between this fetch and normal DMA fetches is that wrapping logic needs to be added to the IA since this is a ring buffer.

These are the new fields are added to the VGT_DMA_INDEX_TYPE register.

3.24.2.1 BUF_TYPE

This can be one of the following enumerators

VGT_DMA_BUF_MEM	Normal draw call DMA fetch from the TC hub
VGT_DMA_BUF_RING	Dispatch Draw DMA fetch from the ring buffer used to store indices from the dispatch

VGT_DMA_BUF_SETUP	This indicates a Dispatch Draw ring buffer setup transfer. On seeing this transfer - The VGT_DMA_SIZE is latched as the ring size. In this mode the size is interpreted as chunks of 256 bytes. - The VGT_DMA_BASE and VGT_DMA_BASE_HI are latched to indicate the ring base. The base needs to be aligned to 256 bytes. - The internal ring offset is reset to 0
VGT_DMA_PTR_UPDATE	This indicates a Dispatch Draw ring pointer update transfer. On seeing this transfer the IA will increment its internal pointer by the amount carried in the VGT_DMA_SIZE register.

3.24.2.2 NOT_EOP

This flag is used to determine if the DMA call is the termination of a draw. The user may choose to tag each sub DMA/DRAW call as a EOP which will result in partial waves at the end of each sub draw. This will allow other HPG requests in the system to be serviced quicker but at the cost of dispatch draw performance. Conversely the user could combine all the sub DMA/DRAWS as a single draw by setting the NOT_EOP flag on all but the last sub DMA/DRAW. This will result in optimal performance for the dispatch draw but will lock out any HPG arbitration until the complete end of the dispatch draw.

3.24.2.3 RDREQ_POLICY and ATC

Since the indices are being fetched though the texture cache, new bits are needed to completely define the type of fetch being issued. The user programming of the *rdreq_policy* and the *atc* will be propagated to the TC on the fetch interface. See the [HYPERLINK \l "_TC_Interface_for"] for details on the values of these fields

3.24.3 Special Deallocation Primitive

In order to support the deallocation of the ring, a special primitive will be inserted into the index stream by the user. This primitive will have all the first index set to a special value with two components. A mask and a data0 portion.

A new register called VGT_DISPATCH_DRAW_INDEX.bits.MATCH_INDEX is added which contains the user programmable mask used to recognize the deallocation primitive. For 16 bit indices this is expected to be *0x0000FC00* and for 32 bit indices it will be *0xFC000000*. The first index of the incoming primitive will be tested against this mask register and if the mask portion matches then the prim is classified to be a dealloc prim.

```
      de-alloc_prim   =   ((Index0   &   MATCH_INDEX   )   ==   MATCH_INDEX)   &&
DISPATCH_DRAW_EN
```

At this point 2 values are calculated, data0 and data 1 as follows

```
      data0 = (Index0 & !MATCH_INDEX) & 0xffff

      data1 = Index1 & 0x7fff
```

These values are accumulated within each wave unless the accumulation is disabled by setting these two new register fields.
DIS_DEALLOC_ACCUM_0 and DIS_DEALLOC_ACCUM_1 in VGT_SHADER_STAGES_EN. The accumulators will be reset to 0 at the end of each wave.

Each dealloc prim will make a single entry into the reuse table which will be tagged as not-hit for future checks and will generate a single degenerate primitive to the PA.

One vertex will be sent to the SPI which will enable the shader to access the deallocation amount. The vertex is constructed as follows

```
    index = 0x80000000 | (data1 <<16) | data0
```

3.24.4 Minimum Index Ring Buffer (IRB) Size

If the following conditions are met, the minimum IRB size is max(max_output_of_a_compute_work_group_in_prim*2, sub_draw_size_in_prims*2) + max_output_of_a_compute_work_group_in_prim. Alternatively a slightly more conservative minimum calculation that can be used is max (max_output_of_a_compute work_group*3, sub_draw_size_in_prims*3).

1. Where sub_draw_size_in_prims has to equal an integer multiple of 2*primgroup_size. All but the last sub_draw of a parent dispatch draw call needs to be an integer multiple of 2*primgroup_size. If advanced CP PFP micro-code issues variable sized subdraws, then they must all be an integer multiple of 2*primgroup_size (except for the final group), and the minimum Index Ring Buffer Size calculation must use the largest sub draw size ever issued.

2. When using dispatch draw, VGT_VTX_VECT_EJECT should be set to PRIMGROUP_SIZE – 2

This minimum size equation is sufficient even if PARTIAL_VS_WAVE_ON is 0 and NOT_EOP is 1. If there are greater than 2 SE's the sub draw size or multiplier in the IRB equation need to be increased.

For example the equation for 4 SE's is max(max_output_of_a_thread_group*4, sub_draw_size_in_prims*4) + max_output_of_a_thread_group.

What drives the minimum IRB size (and VRB size) is a requirement to prevent a dealloc vert/prim from getting stuck in the IRB or VGT after a prim group finishes. In order to free space in the IRB/VRB, an additional prim group is needed to ensure that any dealloc vert in an un-launched partial VS wave can get scheduled to be processed and thus free up IRB/VRB space.

Setting VGT_VTX_VECT_EJECT ensures the next prim group will force out any VS verts waiting in the un-launched partial VS wave from the prior prim group. The minimum IRB/VRB size ensures a future compute work group will be able to output enough prims to trigger a sub draw, which sends a prim group to the VGT and flushes out any partial VS waves containing dealloc verts.

Here's a specific example where a compute work group outputs at most 169 prims * 3 indices to the IRB, including the dealloc prim. Primgroup_size is 91 and the IRB size is 1280, creating the conditions where the IRB is small enough to hang.

The first 11 compute work groups output the max number of prims. Then there's an compute work group output of 501 indices. After this allocation there have been 6078 indices written to the IRB. Dividing by the number of indices in the IRB by the number of indices in a subdraw (546 indices per subdraw derived from primgroup size), (6078/546) indicates 11 sub draws were sent leaving 72 indices in the IRB to be processed. Including these 72 remaining indices, there are 501 indices left to be de-allocated from the IRB. Next there are multiple compute work groups resulting in a bunch of smaller allocations totaling 408 indices. Adding 72+408 results in 480 indices in the IRB to be processed. Not enough indices for a sub draw to be issued.

At this point, if the next compute work group wants to allocate 507 indices for example, the IRB size of 1280 locations less the 501+408=909 indices still in the IRB, prevents the next compute work group from exporting to the IRB and creating enough indices for the next sub draw to be issued. This results in a non-recoverable hang condition.

By adding the extra term max_output_of_a_work_group to the IRB minimum size equation, we ensure the compute work group will have room to allocate.

3.24.5 TC Interface for DMA indices

There will be a new path added for index fetching from the TC. The assumption is that the indices generated by the dispatch portion of the dispatch draw will get hits in the cache. A new interface will be added between each IA and the TC for these index fetches.

In gfx8, the MC interface was removed and the TC interface is the only path for index fetching from gfx8 on

A new field called RDRED_POLICY has been added to the VGT_DMA_INDEX_TYPE register, it can be programmed to the following values to control the TC for the index fetch.

- VGT_POLICY_LRU
- VGT_POLICY_STREAM
- VGT_POLICY_BYPASS

A new field called ATC has been added to the VGT_DMA_INDEX_TYPE register, it can be programmed to the following values to control the TC for the index fetch.

- GPUVM
- ATC

3.24.6 Systems with Multiple IAs

In a system with 4 SEs, there are 2 IAs and there needs to be a mechanism for them to share the ring pointer value.

The WD will break up the incoming dispatch draw call into workgroups and distribute them amongst the IAs. When a workgroup is sent say to IA0, the WD will also send the amount by which the ring pointer moved on IA0 to IA1 using a special null transfer. This will let the other IA update its internal pointer and use the correct value when it receives the next workgroup.

The new transfer from the WD will be identified by setting the BUF_TYPE on the WD_IA_dma interface to VGT_DMA_PTR_UPDATE. This is a new enum added to VGT_DMA_BUF_TYPE. The WD_IA_dma__size field will hold the ring pointer increment amount (number of indices)

The WD will also forward the DMA ring setup transfer to all the active IA units. The Grouper will have to force EOPG when it detects the last prim of a sub-draw. This may not be tagged as EOP or EOI from the WD and the Grouper may not have accumulated enough primitives to detect it as an EOPG. This is necessary since the WD switches IAs at workgroup boundaries and cases where the subdraw size is not a multiple of the workgroup size will have issues if the EOPG isn't forced.

The subdraw size must be an integral multiple of the workgroup size.

If IA harvesting changes, the dispatch draw ring must be reprogrammed (setup transfers are expected)

#endcase

3.25 Index Sizes

The DMA can support 32, 16, or 8 bit indices, controllable by the INDEX_TYPE field of the VGT_INDEX_TYPE and VGT_DMA_INDEX_TYPE registers.

8 bit indices were added in gfx8 and motivated by OpenGL ES which requires support for 8 bit indices.

DMA base addresses will be allowed to be byte aligned when using 8 bit indices.

After the DMA data is returned by the TC for 8 bit indices, the internal index type will be changed to 16 bits and data will be expanded to 16 bits each.

This was done to avoid changing the more complex shifter logic in the grouper downstream. This logic only handles 16 or 32 bit indices.

3.26 Global VS Wave ID

Global VS wave id can be made available to the shader by setting the VS_WAVE_ID_EN field of the VGT_SHADER_STAGES_EN register.

The global waveid will be available in the VS as long as it is not the copy shader (GS enabled)

The user will also need to program SPI_SHADER_PGM_RSRC2_VS.DISPATCH_DRAW_EN in order for the SPI to load the waveid value to an SGPR.

Note that the global wave id value is always available for dispatch draws irrespective of the programming of this field.

3.27 Quality of Service

A new single context register WD_QOS has been added. When the DRAW_STALL field of this register is set, the WD will stop sending work downstream on the WD_IA_draw interface.

The hardware scheduler is expected to set it to drain graphics work out of the system, typically to cease further graphics workload creation during compute time quanta.

The WD will be able to send more work downstream once this field is cleared.

3.28 DMA Reuse

Prior to gfx8, DMA index data was re-fetched for every instance of a draw initiator, there was no reuse available. This feature adds a reuse circuit which eliminates extra index fetches for draw initiators with multiple instances. Indices are only fetched for the first instance.

There are a some restrictions that must be satisfied for this feature to be enabled

- The instance size (number of indices in the draw initiators) must be smaller than the return data fifo in hardware. VGT_NUM_INDICES will be compared against VGT_DMA_DATA_FIFO_DEPTH. The register depth is in dwords and the index type will be converted into dwords before the comparison

- When multiple IAs are present, the WD cannot be in the mode where a single instance is being split up between multiple IAs, the number of primitives formed by VGT_NUM_INDICES (based on

VGT_PRIMITIVE_TYPE) must be lesser than a workgroup size, which is ((IA_MULTI_VGT_PARAM.PRIMGROUP_SIZE+1) * 2)

- When multiple IAs are present, if any of the VGTs in the system are disabled, the number of primitives will be checked against primgroup size NOT workgroup size

There is no software programming necessary to enable this feature (there is also no way to disable it). The hardware will automatically detect when it is safe for the feature to be enabled and will switch to this mode.

4 Hardware Implementation: Top Level

4.1 Top Level Drawing (shows hierarchy of block)

This drawing shows a 4 prim/clock design. A 2 prim/clock design has one WD, one IA, and two VGTs. In configurations with 2 and 1 VGTs, the WD block is still present but acts as a passthrough unit.

[EMBED Visio.Drawing.11]

[EMBED Visio.Drawing.11]

4.1.1 Work Distributer (WD)

The Work Distributer (WD) is at the front of the pipe and its purpose is to support scaling of prim rates beyond two per clock. The WD receives draw command data from the GRBM and distributes the workload to 2 IA blocks. Each IA distributes work to 2 VGT units. If there are 4 Shader Engines (SE), there is a total of 4 VGTs, 2 IAs and 1 WD.

The WD works on coarse draw and dispatch granularity and can distribute work to the two IA units at a high rate. At this stage only sub-draw commands and sub-DMA fetches are transferred. Each IA processes work from the WD at 2 prims/clock and distributes prim groups to the VGTs. Each VGT will independently process 1 prim/clock for a total system throughput of 4 prims/clock.

The main functions of the WD are

- Split up incoming draw commands between the two IA units downstream. This cannot be done for certain cases (explained later) for these special scenarios, the entire draw command will be sent to a single IA unit

- Split up incoming DMA requests between the IA units (the requests match up with the split draw commands). Also route immediate data as needed and generate starting offsets for auto indexed draw calls.

The diagram below shows the main components of the WD. The High Priority (HP) blocks are only present in designs that support high priority graphics.

[EMBED Visio.Drawing.11]

4.1.1.1 Work Distribution

The IA distributes work amongst the two VGTs based on primgroup size. This is the number of primitives sent to one VGT before switching to the next one. The WD distributes work in a similar fashion but at twice the granularity (2 * primgroup size) to each IA. This allows each IA to maintain the primgroup size switching between the VGTs. It also means primgroup size must be an even number. Patches are a special case and a primgroup size of 1 is allowed.

4.1.1.1.1 Auto Index Draw calls

The incoming draw call will be split up between the two IA blocks based on the workgroup size. In addition to this a starting index offset is sent over the draw command interface to each IA. This allows the Grouper in the IA to generate auto index data for its portion of the draw call.

In order to keep the IA's DMA block in sync with the Grouper the CP needs to send a dummy DMA command. The IA will identify a dummy command by checking for VGT_DMA_BASE and VGT_DMA_BASE_HI to be 0xFFFFFFFF. Complete details of the process can be found in the CP packet spec's DRAW_INDEX_AUTO section.

4.1.1.1.2 Immediate Index Draw calls

Not supported.

4.1.1.1.3 DMA Index Draw calls

The original DMA index request needs to be split up between the IAs just like the draw command. Each IA will receive relevant portions of the draw command with the corresponding broken up DMA fetch requests. The DMA engine in each IA will be responsible for fetching the indices that are needed.

#case internal
See dispatch draw for extended functionality.
#endcase

4.1.1.1.4 Draw Opaque calls

No change in operation.

4.1.1.2 Draw Command Interface Design

There is a draw command interface per IA and also per priority graphics.
So for projects with 4x prim rate (2 IAs) and high priority graphics, there are 4 such interfaces.

The interface name will have the following nomenclature, ***WD_IA<X>_<hp | lp>_draw***
Where X indicates the IA id, and hp or lp will indicate high or low priority graphics

The table below doesn't show the prefix for the names.

Name:	Bits:	Description:
WD_IA_draw_transfer_type	3	0 - Draw command Major Mode 0 1 - Draw command Major Mode 1 – Deprecated in gfx8 2 - EVENT_INITIATOR 3 - EVENT_ADDRESS

		4 - State transfer for VGT_MIN_VTX_INDX 5 - State transfer for VGT_MAX_VTX_INDX 6 - State transfer for VGT_INDX_OFFSET 7 - Index transfer for Immediate Data
WD_IA_draw_source	2	0 – DMA draw call 1 – IMMED_INDX draw call 2 – AUTO_INDX draw call 3 – DRAW_OPAQ draw call Only valid for transfer_type = MM0 Draw / MM1 Draw
WD_IA _draw_num_indices	32	**Transfer Type :** Draw Command **-** Number of indices in the sub draw call. **Transfer Type :** Event – Carries event information **Transfer Type :** State data for Min/Max/Offset – Carries register information **Transfer Type :** Immediate Data – Carries immediate index
WD_IA_draw_index_type	1	**Transfer Type : Draw Command** Type of indices being requested 0 – 16 bit indices 1 – 32 bit indices 2 – 8 bit indices
WD_IA_draw_prim_type	6	The primitive type being processed
WD_IA_draw_auto_indx_start	32	The starting index value for auto index draw calls when the transfer type is a draw call
WD_IA_draw_state_sel	3	Indicates state select for the draw call
WD_IA_draw_primid_base	32	This is the primitive id for the first primitive in the sub draw call. The IA will enumerate the rest of the primitives based on this
WD_IA_draw_eoi	1	Indicates end of instance
WD_IA_draw_eop	1	Indicates end of packet

4.1.1.3 DMA Request Interface Design

There is a DMA request interface per IA and also per priority graphics.
So for projects with 4x prim rate (2 IAs) and high priority graphics, there are 4 such interfaces.

The interface name will have the following nomenclature, ***WD_IA<X>_<hp*** | ***lp>_dma***
Where X indicates the IA id, and hp or lp will indicate high or low priority graphics

The table below doesn't show the prefix for the names.

Name:	Bits:	Description:
WD_IA_dma_base	40	Base address for the sub DMA request.
WD_IA_dma_size	32	Size of the DMA request
WD_IA_dma_index_type	1	Type of indices being requested 0 – 16 bit indices 1 – 32 bit indices 2 – 8 bit indices
WD_IA_dma_swap_mode	2	Swap mode for the DMA request
WD_IA_dma_vmid	4	Virtual Memory ID for the DMA fetch
WD_IA_dma_priv	1	Set if the request is privileged
WD_IA_dma_out_of_range	1	This bit indicates that this request is for out of bound data. The DMA

		engine will not issue a fetch to the TC for this request but will internally generate 0x0 for all the indices being requested and feed them to the Grouper

4.1.1.4 Restrictions

The following types of draw commands cannot be split up between the two IAs and will only be sent to one IA. When processing such draw calls, only one IA and 2 VGTs will be used.

- Any draw call with the following primitive types. These primitives need extra indices that will not be available after the DMA/Draw split happens.
  - DI_PT_POLYGON
  - DI_PT_LINELOOP
  - DI_PT_TRIFAN
  - DI_PT_TRISTRIP_ADJ

- Any draw call which enables reset indices (***VGT_MULTI_PRIM_IB_RESET_EN.bits.RESET_EN***). Reset indices cause strips to be broken and require reconstruction of primitives. The index data required may not be available after the DMA/Draw split.

A new register field will indicate when the WD is supposed to send an entire draw call to a single IA unit (***IA_MULTI_VGT_PARAM.bits.WD_SWITCH_ON_EOP***)

The IA will decide whether to send data to both its VGT units based on the same 10xx register (***IA_MULTI_VGT_PARAM.bits.SWITCH_ON_EOP***)

When using Line Stippling, both the IA and the WD need to be in switch_on_eop =1 mode.

4.1.1.5 EOP / EOPG Handling (PA/SC)

Normally the work being distributed by the WD is split up based on the workgroup size (2x primgroup size). There are two register fields that allow the splitting to be controlled.

IA_MULTI_VGT_PARAM.bits.SWITCH_ON_EOP – Makes the IA switch between VGTs at the end of a packet instead of the end of a primgroup

IA_MULTI_VGT_PARAM.bits.WD_SWITCH_ON_EOP – Makes the WD switch between IAs at the end of a packet instead of the end of a workgroup

4.1.1.5.1 EOP

These set of rules is enforced to make sure the VGTs and the SCs are synced up on where the EOP signals are expected.

- The WD will send a null EOP to the IA that doesn't receive the real EOP
- Each IA will send a null EOP to the VGT that doesn't receive the real EOP
- The two rules above ensure that each EOP is seen on all VGTs
- The PA will not drop any EOP and forward them to the SC which will sync up all its FIFOs on seeing the EOP

4.1.1.5.2 EOPG

The SC will need to monitor the control registers mentioned above to track how to expect EOPG signals from the VGT

4.1.1.5.2. 4 Shader Engine system

The draw call will always start on an even SE (0 or 2 depending on which IA received the start of the draw). This allows the following table to be used

WD SWITCH ON EOP	**IA SWITCH ON EOP**	**RULE**
0	0	The SC will switch to the next SE on getting an EOPG
0	1	Invalid Case
1	0	Ping-pong between the SEs in a VGT pair at EOPG Switch to the next even SE at EOP
1	1	The SC will switch to the next even SE on getting an EOPG/EOP

4.1.1.5.2. 2 Shader Engine system

The draw call will always start on the next available SE to avoid small batch draw calls all being routed to SE0.

WD SWITCH ON EOP	**IA SWITCH ON EOP**	**RULE**
x	0	The SC will switch to the next SE on getting an EOPG
x	1	The SC will switch to the next SE on getting an EOPG/EOP

In a 2 SE system wd_switch_on_eop is always considered to be true.

4.1.1.5.3 Front-end Harvesting (GFXIP 7.2)

VGT/PA pairs can be harvested out of the design. The SC must shadow the harvesting register to mask off the SEs that do not have a working VGT. The WD/IA will make sure no work is dispatched to the disabled VGT.

The rules above will be affected by which VGTs are enabled.

4.1.1.5.4 High Priority Graphics

Each PIPE (low and high priority) will independently maintain which SE it started from.

In a 2 SE configuration, the IA will send both LP and HP draw calls to SE0 and track their SE id switching independently.

In a 4 SE configuration, the WD will send both LP and HP draw calls to IA0, SE0 and track their SE id switching independently.

This is for validation purposes to match the emu and rtl.

4.1.2 Input Assembler (IA)

4.1.3 Vertex Geometry Tessellator (VGT)

[EMBED Visio.Drawing.11]

4.2 Data Flow Diagram

4.2.1 VS -> PS

4.2.2 ES -> GS -> VS -> PS

From the API's perspective this looks like VS -> GS -> PS.

[EMBED Visio.Drawing.11]

4.2.3 LS -> HS -> VS -> PS

From the API's perspective this looks like VS -> HS -> DS -> PS.

4.2.4 LS -> HS -> ES -> GS -> VS -> PS

From the API's perspective this looks like VS -> HS -> DS -> GS -> PS.

4.3 Switching Stages

There is a variable amount of overhead involved in the VGT to switch between the data paths described above. This can range from none "fully pipelined" to various stalls at different points. The table below summarizes these transitions

	DX9	DX10	DX11 GS Off	DX11 GS on	PassThru	DX11 NGG on	NGG on
DX9	NO STALL	PI STALL	WD STALL	WD STALL	PI STALL	WD STALL	PI STALL
DX10	PI STALL	NO STALL	WD STALL	WD STALL	PI STALL	WD STALL	PI STALL
DX11 GS Off	WD STALL	WD STALL	NO STALL	NO STALL	WD STALL	WD STALL	WD STALL

DX11 GS on	WD STALL	WD STALL	STALL	NO STALL	WD STALL	WD STALL	WD STALL
PassThru	PI STALL	PI STALL	WD STALL	WD STALL	NO STALL	WD STALL	PI STALL
DX11 NGG on	WD STALL	WD STALL	WD STALL	WD STALL	WD STALL	NO STALL	WD STALL
NGG on	PI STALL	PI STALL	WD STALL	WD STALL	PI STALL	WD STALL	NO STALL

5 Hardware Implementation: Interfaces

Interfaces are documented separately in the interface specs.

6 Hardware Implementation: External Interfaces

The IA/VGT do not interface with any non-GFXIP blocks.

7 Hardware Implementation: SubBlocks

The top level hierarchy and interfaces have already been described. This section details the micro-architecture below the top level.

7.1 GS Reuse Check Module

The RCM tests indices for vertex index reuse and assigns a write offset for the first vertex of the ES wavefront. This write offset is the location where ES output will be written in the ESGS ring buffer. The RCM's reuse buffer stores up to 16 indices and is reset at the end of a draw instance or subgroup.

The Reuse Check Module also generates GS primitives along with a write address for the GS wave. ES and GS data is sent to the SPI. GS prim data includes offsets pointing to the vertex data in the ESGS ring buffer. For example, a triangle prim type requires each GS prim to have 3 offsets and depending on vertex reuse consecutive GS prims may receive the same offsets.

The RCM outputs ES verts and GS prims to the SPI. All verts (4 waves worth of single cycle verts) are buffered in the SPI, but the RCM contains extra buffering for GS prims as defined by the gpu.vgt.gsprim_buff_depth feature.

If GS instancing is enabled the RCM will generate the extra GS prims, ensuring full vertex reuse.

The RCM ensures proper wrapping for the ESGS and GSVS rings. A wavefront cannot wrap so if a wave won't fit it's moved to the bottom and extra space is allocated at the end of the ring buffer.

7.2 GS Thread Manager

ES and GS waves are built in the RCM and passed to the Thread Manager (TM) via ES and GS tables. When the ES table is not empty the TM checks for available ring buffer space before sending the ES wavefront to the SPI. There's also the option to limit the number of ES waves in the ESGS ring which is enforced here. This is programmed via VGT_CACHE_INVALIDATION.ES_LIMIT.

There are more conditions to meet before a GS wave can be launched. Waves are launched in order and the corresponding ES waves must finish prior to launching the GS waves. There must also be available GSVS ring buffer space, counters, cut memories, and counter table locations. The peak number of GS waves per VGT is defined by the feature gpu.gc.num_max_gs_thds.

When GS done's are received space is reallocated to the ESGS ring buffer.

The Thread Manager also contains the VS table (FIFO). It contains dealloc sizes for each VS wave. When VS done's are received (only during scenario G) the TM reads the VS table to determine how much space to reallocate to the GSVS ring buffer.

7.3 GS Counter Manager

When a GS wave is sent to the SPI a counter table location is written. A state machine is associated with each table location and it waits for the proper GS done before generating emits and cuts. Emits, cuts, various flags, and alloc sizes are sent to the GoG.

7.4 GS Output Geometry Unit (GoG)

Receives emits and cuts from the Counter Manager and sends verts and prims to the output block. Each vert has an associated address used to tell the shader where to fetch in the GSVS ring buffer. If necessary partial VS waves are forced out based on the GS_PER_VS register setting.

After each VS wave is complete the GoG writes to the VS table. The GS Thread Manager contains the VS table and its section of the spec describes the table's purpose.

7.5 Hull Shader block

This document describes the Hull Shader (HS) sub-block of the Vertex Geometry Tessellator (VGT) unit. This block implements two primary areas of shader functionality associated with DX11 Tessellation: The Vertex shader control (utilizing Local Shader (LS) interfaces) and the Hull Shader Control (using Hull Shader interfaces). In addition, the HS blocks supports compute shaders by issuing compute shader wavefronts over the LS interfaces.

For tessellation, the HS receives work in the form of a sequence of patches, each comprised of a number of vertex indices (patch input control points). Patches are received a single vertex per clock. The HS groups these indices into threadgroups for LS processing. Individual vertices are sent via the LSVERT interface. Wavefronts are sent via the LSWAVE interface. Simultaneously HS begins preparing HS threadgroups, sending vertices immediately, but delaying sending an HSWAVE until the corresponding LS threadgroup is reported as being complete.

For each threadgroup an entry into a Tessellation Input FIFO (TIF) is made. This FIFO is read by the TE, and dictates the number of patches the TE will tessellate. Events are also sent to the TE module.

[EMBED Visio.Drawing.11]

7.5.1 HS Functional Blocks

[EMBED Visio.Drawing.11]

7.5.2 Local Shader Manager (LSM)

The Local Shader Manager (LSM) is primarily responsible for the generation of LS threadgroups, which in turn are comprised of LS verts and LS waves. The interface for issuing LS verts and LS waves is between the VGT and (in dual core systems) one of two SPI units, SPI_00 and SPI_10. Work is provided to the LSM block by the grouper (GRP) block, which sends patch data to the HS a single vertex at a time. In dual core configured systems the LSM round robins thread groups to each of the two cores such that two consecutive thread groups always go to different cores Each vertex received from the GRP is transmitted as soon as a credit is available on the interface (i.e. no resource allocation is required). When a wavefront 's worth of vertices have been received by the LSM, the LSM creates one LM wavefront.

When the LSM has sent all of the wavefronts associated with a single threadgroup, it sends an SX_FLUSH_LS event. It is necessary for the HSM block to know when all the work for the LSM threadgroup is complete, as it is not permitted to start the HS wavefront for the threadgroup until all LS waves for the threadgroup are complete. This insures that the Local Data Store (LDS) contains all LS processed patch input vertices, prior to executing any HS threads that must have access to ALL LS processed input vertices.

A secondary function of the LSM is to support Compute Shader wavefront requests. When the GRP input to the LSM indicates CS data, the LSM bypasses it's internal logic and directly issues the CS wave request over the LS wave interface. In addition, for the first wave (indicated by a flag) of a compute shader threadgroup, the threadGroupID (which appers on the GRP interface) is transferred by issuing a single LS_Vert interface request, with the CS flag asserted.

[EMBED Visio.Drawing.11]

7.5.3 Hull Shader Manager (HSM)

The Hull Shader Manager (HSM) is primarily responsible for the generation of HS threadgroups, which in turn are comprised of HS verts and HS waves. The interface for issuing HS verts and HS waves is between the VGT and (in dual core systems) one of two SPI units, SPI0 and SPI1. Work is provided to the HSM block by the Local Shader Manager (LSM). A work request from the LSM is a request to create enough work to Hull Shader one Patch. An "EndOfThreadGroup" Flag allows the HSM to segrated the patch requests into groups of requests each related to a single threadgroup. Upon receipt of a request to create one patch, the HSM may issue HS_Verts immediately, (interface credit permitting). As the HSM creates sufficient verts to constitute a wavefront, it will send the wavefront(s) to the SPI, but prior to sending the wavefront associated with a given threadgroup, the HSM waits to receive a SX_VGT_ls_done

Prior to sending the wavefront to the SPI, the HMS allocates a chunk of Tessellation Factor Memory (TFM) – enough to contain all the tessellation factors required for a threadgroup of HS shader output. This allocation is requested from the Tessellation Factor Memory Manager (TFMM) block. When the allocation has been granted (which provides a base address within the TFM) the HMS may commence issuing the wavefronts associated with the threadgroup. Only the first wavefront of the group must pend until this allocation is complete, as the allocation is done for the entire threadgroup. Subsequent wavefronts for the group may be sent to the SPI as long as a credit is available on the interface.

[EMBED Visio.Drawing.11]

7.5.4 Tessellation Factor Memory Manager (TFMM)

The Tessellation Factor Memory Manager manages the allocation and deallocation of storage available in the tessellation factor memory (TFM). This memory is instanced per core, but is otherwise globally accessible to all SIMDs within a core. Requests for allocation of this memory come from the Hull Shader Manager (HSM) block and are defined simply by a request size. If the TFM has the requested amount of contiguous space available the TFMA grants the request to the LSM, and provides the base address of the granted block of memory (always contiguous space).

The TFM is managed as a ring and deallocation happens in the exact order of allocation. The consumer of the TFM data is the Tessellation Engine (TE), or a module on behalf of the TE. It provides a signal, te11_hs_threadgroup_done, that unconditionally deallocates, in allocation order, a range of the TFM.

Since the TF ring in memory is shared by two VGTs, the ring will be split in half with VGT0 using the top half and VGT1 using the bottom half.

Reset: When reset is present the TFMM initializes it's BASE address (per core) to zero, and it's SIZE registers (per core) to the value of VGT_TF_RING_SIZE.SIZE. Additionally, the Deallocation FIFO is reset to the empty state.

Rollover of the Ring: While the ring nature of memory management requires rollover, mode control determines whether rollover is permitted to occur within the range of data allocated by a single allocation. Only if VGT_TF_RING_SIZE__ALLOW_WRAP is set is allocation allowed to return a segment of memory that includes a wrap form maximum memory address to address 0. If this state setting is not asserted, any allocation that would otherwise cause such a warp is forced to allocate its space starting with address zero. contiguous. [EMBED Visio.Drawing.11]

7.5.5 Tessellation Input FIFO (TIF)

This FIFO is written by the Hull Shader Manager (HSM) in order to give a block of work to the Tessellation Engine (TE). Each such entry in the FIFO contains enough work for the TE to fully process on threadgroup, which is itself comprised of some number of patches.

In order to keep events synchronized with Draw Packets, Events are also written in correct order, to this FIFO. Correct order means that an event received between Packet A and packet B is written to the FIFO AFTER all threadgroups created on behalf of packet A, but BEFORE all threadgroups

Tessellation Input Fifo Contents

Tessellation Input FIFO may contain either events or threadgroup data. A single bit (msb) stored in the FIFO indicates which of the two data formats is stored in the FIFO. The two formats are described below.

Tesselation Input Fifo Contents -- ThreadGroup Data

Name:	Bits	BitPos:	Description:
hs_te11_tess_input_state_sel	3	2:0	State select of the threadgroup
hs_te11_tess_input_core	1	3:3	Indicates which core received the threadgroup
hs_te11_tess_input_num_patches	8	11:4	The number of patches in the threadgroup
hs_te11_tess_input_tf_addr	9	20:12	The starting address used to fetch tessellation factors
hs_te11_tess_input_null	1	21:21	Indicates a null threadgroup. No patches are to be processed. This is used to send an eop when there is no threadgroup to process.
hs_te11_tess_input_eop	1	22:22	Indicates the end of packet
hs_te11_tess_input_lgi	1	23:23	Last group of instance
spare	8	30:24	spare
is_event	1	31	must be set to zero for the non-event data in this description

Tesselation Input Fifo Contents -- Events

Name:	Bits	BitPos:	Description:
hs_te11_tess_input_state_sel	3	2:0	State select
event_data	28	30:3	Event specific data
is_event	1	31	must be set to "1" for event data

7.6 TE11 block

7.6.1 Overview

This document describes the DX11 Tessellator Engine (TE11) sub-block of the 8XX Vertex Grouper Tessellator unit. The Dx11 Tessellator will tessellate patches based on tessellation factors (TF) for each edge of the patch. There can be 2, 4 or 6 TFs per patch. Based on these factors, TE11 will break up the patch into numerous points, lines or triangles based on the tessellation topology.

The TE11 receives work from the VGT_HS block in the form of threadgroups. Each threadgroup defines a number of patches, a starting address into the TF memories used to fetch tessellation factors and other state information. The TE11 will process each patch from an input threadgroup, request the number of tessellation factors it needs for each patch and tessellate the patch based on various state data (partition, topology, axis, etc.). TE11 will output vertex data and primitive data to either the GS block or the Output block depending on whether or not the GS is enabled. The vertex data out of TE11 will be u,v values that the Domain Shader uses to calculate the actual new vertex data.

7.6.2 Block Diagram

[EMBED Visio.Drawing.11]

7.6.3 TE11 Sub Block Diagrams

[EMBED Visio.Drawing.11]

Since the above diagram was drawn the Threadgroup-to-Patch block has been split up so the TF request logic is separate.

[EMBED Visio.Drawing.11]

7.6.4 TF interface to read/write cache

Overview

The new read/write cache will be used to access the Tessellation Factor Memory. This will cause the following changes:

- VGT will support TFM ring size of up to 2^16 dwords.
- The TFM ring will be shared by both VGTs (SE0VGT will use the top; SE1VGT will use the bottom).
- VGT will pass the 16-bit TF threadgroup base offset to SPI on the HS wave interface.
- SPI will load the TF base offset into a SGPR for use by the shader for TF write addressing.
- VGT will have a new read request interface to the TC and return data interface from the TC.
- VGT will have a latency hiding FIFO to store TF return data (1 dword @ 512 clocks of latency).
- Driver will need to set up a TF Memory resource.
- Driver will need to program a TF Ring base address register and TF ring size register for VGT.
- The BUFFER_ACCESS_MODE register is deprecated, all TF writes are assumed to be in the old PATCH_MAJOR
  format, this is because moving the TF memory to cache backed space eliminated the write conflicts in the GDS

The VGT will make requests of 512 bits per request to the read/write cache and receive the return data over 2 clocks, 256 bits per clock.

TF Read Requestor subblock

The TF Read Requestor will make requests to the read/write cache based on threadgroup information it receives from the HS block. It will receive the relative starting address in the TF memory for the threadgroup and the number of patches in the threadgroup. From this information it will calculate the starting request address and the ending request address for that threadgroup:

relative starting address [dword aligned] = TF relative starting address (from HS block)

relative ending address [dword aligned] = TF relative starting address + (patches/threadgroup * tf/patch) + 1

Note: the +1 in the ending address equation accounts for the HS offchip control factor. The number of tess factors per patch (tf/patch) is based on the type of tessellation (isoline = 2; tri = 4; quad = 6)

starting request address [512-bit aligned] = (TF base address << 2) + (relative starting address >> 4)

ending request address [512-bit aligned] = (TF base address << 2) + (relative ending address >> 4)

Note: the above equation assumes that the TF base address is 256-byte aligned.

The TF requestor will then begin sending requests to the read/write cache beginning at the starting address and then incrementing the address by 1 until the ending address has been reached.

The TF requestor will also send information on the request tag that will be used to address the return data fifo:

Write address into the TF Data FIFO [5 bits]

The read-only cache interface (CLIENT_TC_rdreq) is described in the document client_tc.doc.

The TF requestor will also be responsible for ensuring that the TF Return Data FIFO does not overflow. It will keep track of the full logic for that FIFO and only issue requests when there is space available in the FIFO.

The TF requestor will also send certain parts of the threadgroup data to the vgt_te11_ttp block which is needed for patch generation. It will also generate the starting TF pointer for the threadgroup and send that with the other threadgroup data. This value is set to the first dword location of valid TF data minus 1. See diagram:

[EMBED Visio.Drawing.11]

The above example shows a stream of tess factors for one threadgroup. The starting TF in the stream is the 6th dword in the first 512-bit data return. The ending TF of the stream is the 10th dword of the third 512-bit data return. Therefore, the start pointer for this threadgroup will look like this.

Start TF pointer = 0x5

The ending TF pointer does not need to be sent since the read side knows how many patches (and therefore how many tess factors) it needs to process per threadgroup.

TF Return Data subblock

The TF Return Data block will receive data at 256 bits per clock and write that data to a storage FIFO. The tag information returned with the data will contain the write address into the memory. Since the request is for 512 bits yet data is returned in 2 clocks, this block will need to use {tag address, 1'b0} for the first piece of data returned and then the {tag address, 1'b1} for the second. Also, since this data can be returned out of order, each location in the memory will have a corresponding valid bit which gets set when the data is returned and reset when that location is read. In this way, the write side does not act as a traditional FIFO

The read side of the TF Data FIFO will act as a traditional FIFO. Data will be read out in order (ie. read pointer will always increment by 1). The only difference is that the valid bit will need to be set for the location that the read pointer is accessing.

The TF Data FIFO will be sized to provide 1 tessellation factor (32-bits) per clock at 512 clocks of latency. It is believed that around 512 clocks will be the worst case latency of requests if the request is a miss in the cache. This means that the size of the TF Data FIFO should be 64x256. Using this size FIFO, we will see a benefit when the tess factors are in the cache. For example, if the latency of a hit in the cache is 128 clocks then VGT would be able to maintain a rate of 4 tess factors per clock. This will allow an increase in the performance of non-tessellated patches (tf = 1.0) and null patches (tf <= 0.0 or tf = NAN).

The TF Return Data block will provide the 256-bit read data bus from the TF Data FIFO to the vgt_te11_ttp block.

7.6.5 TE11 Threadgroup-to-Patch (vgt_te11_ttp)

The TE11 Threadgroup-to-Patch block takes each threadgroup input from the TF requestor block and converts it to multiple patches to be sent serially to the next TE11 stage. A controller at the front-end of the TTP block will then pull the proper tess factors out of the 256-bit data chunk and store them in the tess factor FIFOs. There are 7 FIFOs, one for the control factor and 6 for the tess factors themselves.

On the way into the tess factor FIFOs, the tess factors will be run through a float-to-fix converter which also handles min/max clamping. Therefore, the stored tess factors will be in 7.16 fixed point format and ready for use by the vgt_te11_su block.

Since there is only slight performance advantages to providing all 6 tess factors in one clock (only null patches would benefit from this), the TTP will only write two tess factors per clock into the Tess Factor FIFOs. This will increase the performance of non-tessellated (tf = 1.0) quad patches from 6 clocks per patch to 4 clocks per patch (this type of patch produces 4 vertices which will take 4 clocks to create). It will also increase the rate at which null quad patches are processed from 6 clocks to 3 clocks. Limiting the performance to 2 TF per clock also reduces area since only 2 float-to-fix converters will be necessary as opposed to 6 converters needed to handle a patch per clock.

All of the tessellation factors for the patch and state information is sent to the TE11 Setup submodule for each patch in the threadgroup. This block also sends a flag to mark the end of a threadgroup and the end of a packet.

7.6.6 TE11 Setup (vgt_te11_su)

The TE11 Setup block receives 1 patch at a time and calculates all values need to tessellate that patch. It provides 6 "magic" numbers plus parity flags that are used by the Point Generation algorithm. These "magic" numbers are described in "TE11 Algorithm" section.

7.6.7 TE11 Point Generation (vgt_te11_pg)

The TE11 Point Generation block receives patch information from the Setup block and creates all of the tessellated points of the patch.

#case internal
The Microsoft algorithm calculates every point in the patch and stores it in memory to be used during the connectivity pass.
#endcase
#case kryptos
A third party algorithm calculates every point in the patch and stores it in memory to be used during the connectivity pass.
#endcase

However, a single patch can have up to 4225 points so this is not efficient for the hardware. Therefore, this block creates the points for an outside edge and an inside edge in parallel so that the Connectivity block can create output primitives in the proper order. It will start with the outside left edge and the inside left edge and create points from bottom to top. It will then create points on the 2 top edges followed by the right side and finally the bottom edges of the ring. Once the outer ring is complete, the process will repeat for the next inside ring. The patch will be complete when all rings within the patch have been completed. This process of point generation forms a snake pattern (see diagram) and does not require any point storage because points are created in the order they are connected. Each piece of point data will be a u,v coordinate used by the Shader to create the new vertex data. Each new primitive created will also be tagged with the patch ID.

[EMBED Visio.Drawing.11]

There are 2 special case where the points are generated in a different fashion. Both occur during the last ring of the patch. The first special case occurs when the patch ends as a polygon. This means that the last ring has no inside edges. So, TE11 processes the top and right edges in the outside edge math unit and processes the left and bottom edges in the inside math unit. this creates a stream of points that then can be connected as a group of triangles in the middle of the patch.

[EMBED Visio.Drawing.11]

The other special case occurs when a patch ends with a line in the middle. In this case, the points in the middle are processed by the inside edge math unit. It processes the line from left to right and then turns around and regenerates the points from right to left (excluding the rightmost point). This is done because there will be triangles above the line and below the line that reuse the same points of the line. If the line is less than 14 points long, the reuse buffer in the Connectivity block will ensure that the repeated points will only be sent to the shader once.

[EMBED Visio.Drawing.11]

Both of the special cases described above can also occur with the V dimension being greater than the U dimension. This means that the polygon or line will be vertical instead of horizontal. This causes different edges to be processed in the math units.

[EMBED Visio.Drawing.11]

7.6.8 TE11 Connectivity (vgt_te11_con)

The TE11 Connectivity block receives tessellated point data from the Point Generation block and creates primitives based on the topology (point, line or triangle). It will send out the vertex data in strip form and will send relative indices for the primitives. It will also have to resend indices that have been reused if the distance between the newest index and the reused index is more than 14. This is only necessary when the topology (output primitive type) is triangle and is due to the width of the Parameter Cache.

#case internal
The TE11 reuse logic does have an advantage over the Microsoft algorithm. Microsoft handles it's reuse based on an index to the (u,v) value it stores in memory. The TE11 handles it's reuse on the actual (u,v) values. This helps in cases where degenerate triangles are formed due to Microsoft's algorithm. With some values of tessellation factors, the algorithm will produce the same (u,v) value for multiple points on an edge. However, Microsoft considers these points as unique and sends all of them as output. The TE11 will send the first point and then see that later points have the same (u,v) value and not send them. This saves shader processing.

The connectivity of the output vertices will be determined by a set of lookup tables which will be accessed by Tessellation Factor information. The Microsoft algorithm uses only one 32-entry LUT that is looped through to determine when a triangle can be created. Using only one table means that it could take up to 32 clocks to create one primitive. This is very inefficient for the hardware because the performance requirements are to produce one primitive per clock. After some analysis, we found that we could break the one LUT into 32

separate tables. By selecting 1 of the 32 tables (using the "half tess factor" value), we could loop through the new table and create one primitive per clock.
#endcase
#case kryptos
The TE11 reuse logic does have an advantage over the third party OS algorithm. The third party OS handles it's reuse based on an index to the (u,v) value it stores in memory. The TE11 handles it's reuse on the actual (u,v) values. This helps in cases where degenerate triangles are formed due to the third party OS's algorithm. With some values of tessellation factors, the algorithm will produce the same (u,v) value for multiple points on an edge. However, the third party OS considers these points as unique and sends all of them as output. The TE11 will send the first point and then see that later points have the same (u,v) value and not send them. This saves shader processing.

The connectivity of the output vertices will be determined by a set of lookup tables which will be accessed by Tessellation Factor information. The third party OS algorithm uses only one 32-entry LUT that is looped through to determine when a triangle can be created. Using only one table means that it could take up to 32 clocks to create one primitive. This is very inefficient for the hardware because the performance requirements are to produce one primitive per clock. After some analysis, we found that we could break the one LUT into 32 separate tables. By selecting 1 of the 32 tables (using the "half tess factor" value), we could loop through the new table and create one primitive per clock.
#endcase

7.6.9 TE11 Algorithm

[SHAPE * MERGEFORMAT]

Initial Setup

This is the first step in the tessellation algorithm which mainly involves clamping the hull shader specified values to legal ranges (a final sanity check, the compiler will automatically insert HLSL code after the hull shader for some clamping and calculations as well). Parity computation is also done here this parity is used in various stages of the algorithm. Invalid patches are flagged for culling

- Detect invalid patches that need to be culled
```
if(any outside edge factor is <= 0 || any outside edge factor = Nan){
    mark the patch for culling
}
```

- Perform hardware clamping checks on the **outside** tessellation factors as follows
```
if(partition_type = integer || power of two){
    clamp to range 1:64

    After clamping,  factor = ceil(factor)

        No distinction between power of 2 and integers. Only round to the next integer, not to the next
        power of 2.
}
if(partition_type = even){
    clamp to range 2:64
}
if(partition_type = odd){
    clamp to range 1:63
}

if(tessellation_type =  isoline){
    clamp line density factor (2nd tess factor) to the range 1:64

    factor = ceil(factor)
}
```

- Perform hardware clamping checks on the **inside** tessellation factors as follows. Only different from the external factors clamping if partition type is ODD and any one of the external or internal factors is greater than 1.0f
```
if(partition_type = integer || power of two){
    clamp to range 1:64  , After clamping,

    factor = ceil(factor)

        No distinction between power of 2 and integers.
}
if(partition_type = even){
    clamp to range 2:64
}
```

```
if(partition_type = odd){

    // Note that this clamps a Nan to the lower bound

    if( any of the outside OR inside factors is > 1.0f) {
        clamp to range (1 + 2^-16)  :  63
        } else {
        clamp to range 1:63
        }
}
```

- Perform float to fixed conversion of the factors here

- For tessellation factor compute the parity as follows

```
If(partition_type =  power of two){
    Set the parity as EVEN
}
If(partition_type =  integer){
    tess_factor = ceil(tess_factor)
    if is resulting tess_factor is odd set parity to ODD
    or set it to EVEN
}
If(partition_type = even || odd     &&   tess factor is an external factor){
    Set the parity to the partition_type
}
If(partition_type = even || odd     &&   tess factor is an internal factor){
    Set the parity to the partition_type
    Except if the tess_factor = 1.0f, set the parity to EVEN  --- this is the only exception ---
}
```

- Detect special case for the base level of tessellation as follows

```
If(partition_type = integer || power of two     &&   tessellation_type = tri || quad ){
    If(all external tess_factors = 1.0f){
        Set a flag to process the base_case
        }
}
```

Discard patch

The incoming patch has been flagged as one that needs to be culled. This should produce no vertices or prims out of the tessellator. *[find out how this will be handled, null etc]*

Handle Special Cases

If a special case was detected during the Initial Setup phase, generate the correct hardcoded primitives and exit. Special case handling is only valid when the tessellation_type is tri or quad

```
if(special flag for base_case is set){
    if(tessellation_type =  quad ){
        number_of_points = 4
        number_of_tris = 2

        define_point(0,0)
        define_point(1,0)
        define_point(1,1)
        define_point(0,1)

        if(output_prim_type = clockwise triangle){
            define_triangle(0,1,3)
            define_triangle(1,2,3)
        }
        else if (output_prim_type = counter clockwise triangle){
            define_triangle(0,3,1)
            define_triangle(1,3,2)
        }
        else if (output_prim_type = points){
            ouput the 4 points as a point list
        }
    }
        else if(tessellation_type =  tri ){
        number_of_points = 3
        number_of_tris = 1

        define_point(0,1)
        define_point(0,0)
        define_point(1,0)

        if(output_prim_type = clockwise triangle){
            define_triangle(0,1,2)
        }
        else if (output_prim_type = counter clockwise triangle){
            define_triangle(0,2,1)
        }
        else if (output_prim_type = points){
            ouput the 3 points as a point list
        }

    }
}
```

Intermediate Value Calculation

The algorithm used to compute the parametric positions of new points generated by the Tessellator use certain values repeatedly. These are based on the tess_factor for any given edge. Since the set of tess_factors will be the same per draw packet, it makes sense to compute the intermediate values as a pre-process.

The RemoveMSB function used here is detailed in the following section

There are 5 values (highlighted in the bullets below) that are needed per tess_factor, so in the worst case 6*5 numbers will be pre-computed for future evaluations

These are the steps involved in the computations. All values are now in fixed point representation

```
half_tess_factor = (tess_factor+1)/2

if(partition_type = odd ||  half_tess_factor = 0.5){
    half_tess_factor = half_tess_factor + 0.5
}

floor_half_tess_factor = floor(half_tess_factor)

ceil_half_tess_factor = ceil(half_tess_factor)

half_tess_factor_fraction = half_tess_factor – floor_half_tess_factor

num_half_tess_factor_points = ceil_half_tess_factor >> 16

if(floor_half_tess_factor = ceil_half_tess_factor){
    split_point = num_half_tess_factor_points + 1

        This is set so that the splitting is ignored. When the ceil and floor half factors are equal, the division is
        regular. We only walk to the half way point so setting the split beyond the half makes sure its ignored
}
else if(partition_type = odd){
    if(floor_half_tess_factor = 1.0){
        split_point = 0
        }
    else{
        split_point = (RemoveMSB((fxpFloorHalfTessFactor>>16)-1)<<1) + 1
    }

}
else{
        split_point = (RemoveMSB(fxpFloorHalfTessFactor>>16)<<1) + 1
}

num_floor_segments = (floor_half_tess_factor * 2 ) >> 16

num_ceil_segments = (ceil_half_tess_factor * 2 ) >> 16


inv_num_floor_segments = fixed_reciprocal (num_floor_segments )

inv_num_ceil_segments = fixed_reciprocal (num_ceil_segments )
```

The fixed_reciprocal is a lookup table that contains 65 possible values.

The RemoveMSB function

This is the RemoveMSB function that was used in the preceding code segment

```
int check

if( val <= 0x0000ffff ) {
        check = ( val <= 0x000000ff ) ? 0x00000080 : 0x00008000
}
else  {
        check = ( val <= 0x00ffffff ) ? 0x00800000 : 0x80000000
}

for( int i = 0 ;    i < 8 ;  i++, check >>= 1 ) {
        if( val & check )
                return (val & ~check)
}

return 0
```

Calculate the number of points (tri and quad only)

In this stage the number of points for the outermost ring and all the inner rings are computed. This part of the algorithm is only used for tri and quad tessellation types.

This part computes the number of exterior points

```
if(tessellation_type = tri){
    number_of_sides = 3
}
else if(tessellation_type = quad){
    number_of_sides = 4
}

number_of_outside_points = 0

Do for number_of_sides{
    parity = parity computed for this edge's tess_factor

        if(parity = odd){
            number_of_points_for_edge = (ceil(0.5+(tess_factor+1)/2)*2)>>16
        }
        else{
                number_of_points_for_edge= ((ceil((tess_factor+1)/2)*2)>>16)+1
        }

        number_of_outside_points = number_of_outside_points  + number_of_points_for_edge
}

number_of_outside_points = number_of_outside_points  - number_of_sides
```

This part computes the number of interior points

```
if(tessellation_type = quad){
        parity = parity computed for this edge's tess_factor

            if(parity = odd){
                number_of_points_for_U = (ceil(0.5+(tess_factor_U+1)/2)*2)>>16
                    number_of_points_for_V = (ceil(0.5+(tess_factor_V+1)/2)*2)>>16

                    min_number_of_points = 4
            }
            else{
                    number_of_points_for_U= ((ceil((tess_factor_U+1)/2)*2)>>16)+1
                    number_of_points_for_V= ((ceil((tess_factor_V+1)/2)*2)>>16)+1

                    min_number_of_points = 3
            }

        number_of_points_for_U = max (number_of_points_for_U , min_number_of_points)
        number_of_points_for_V = max (number_of_points_for_V , min_number_of_points)

        number_of_inside_points = (number_of_points_for_U - 2)  *  (number_of_points_for_V  - 2)

    }
    else if(tessellation_type = tri){
        parity = parity computed for this edge's tess_factor

            if(parity = odd){
                number_of_points_for_U = (ceil(0.5+(tess_factor_U+1)/2)*2)>>16
                    min_number_of_points = 4
            }
            else{
                    number_of_points_for_U= ((ceil((tess_factor_U+1)/2)*2)>>16)+1
                    min_number_of_points = 3
            }

        number_of_points_for_U = max (number_of_points_for_U , min_number_of_points)

        number_of_interior_rings = (number_of_points_for_U  >> 1 ) -1

            if(parity = odd){
             number_of_inside_points=3*(number_of_interior_rings*(number_of_interior_rings+1)-
    number_of_interior_rings)
        } else{
                 number_of_inside_points= 3*( number_of_interior_rings *( number_of_interior_rings +1)) + 1
            }

    }
```

Calculate point position

This is the function called by all point generation cases (tri/quad/isoline or inner and outer rings).

Its input are the 5 intermediate values computed for the edge under evaluation and an index on the edge.

(index, num_half_tess_factor_points, half_tess_factor_fraction, split_point, inv_num_floor_segments, inv_num_ceil_segments)

These are the evaluation steps :

```
flip = false

if(index >= num_half_tess_factor_points){

    index = (num_half_tess_factor_points << 1) - index

    if(parity = odd){
        index = index - 1
        }

        flip = true
}

if(index = num_half_tess_factor_points){
    this is a special case since the fixed point math cannot produce 0.5 exactly

    result = 0.5

    return
}

ceil_index = index
floor_index = index

if(point > split_point){
    floor_index = floor_index – 1
}

floor_location = floor_index * inv_num_floor_segments
ceil_location   = ceil_index   * inv_num_ceil_segments

result = floor_location * ( 1.0 - half_tess_factor_fraction)
                +
         ceil_location * half_tess_factor_fraction

result = (result + 0.5) >> 16     // rounding


if(flip){
    result = 1.0 - result
}
```

Quad outer ring point generation

```
for(edge = 0;  edge < 4 ; edge++ ){
        parity = edge & 0x1

        startPoint = 0
        endPoint = num_points_for_tess_factor - 1

        for(p = startPoint ;  p < endPoint ;  p++) {
            int q = (edge < 2) ? p : endPoint - p // reverse order

            parity = edge parity
                result = Calculate point position (q, intermediate values of edge)

                if( parity ){
                define_point((edge == 1) ? 1.0 : 0,  result)
            } else{
                define_point(result,   (edge == 2) ? 1.0 : 0)
            }
        }
  }
```

Quad interior point generation

```
startRing = 1 // the outermost ring (0) was special cased earlier

min_num_points = min (number_of_points_for_U, number_of_points_for_V)

numRings = min_num_points >.>1

// process each interior ring
for(ring = startRing ;  ring < numRings ;  ring++){

    startPoint = ring

    endpoint[0] = number_of_points_for_U – 1 – startPoint    // represents U
    endpoint[1] = number_of_points_for_V – 1 - startPoint    // represents V

    for(edge = 0 ;  edge < 4 ;  edge++ ){

        parity[0] = edge & 0x1
        parity[1] = (edge+1) & 0x1

        // Step 1 is to find out the coordinate along the axis that DOES NOT change

        if( edge = 0 || edge = 3){
            perpendicularAxisPoint = startPoint
                } else {
                    perpendicularAxisPoint = endPoint[   parity[1]   ]
                }

                // here, insideParity[0] is the parity for U, and insideParity[1] is the parity for V
                Set local parity = insideParity [   parity[1]     ]

                // here the intermediate values of edge [0] are for U and [1] are for V
                // basically trying to index into the right intermediate numbers
                perp_result = Calculate point position (perpendicularAxisPoint, intermediate values of inside
        edge [parity[1]])
        // Step 2 is to find out the coordinate along the axis that CHANGES

            Set local parity = insideParity [   parity[0]     ]

            // move along all the points for this inside edge
            for(p = startPoint ;  p < endPoint[  parity[0]  ]   ;  p++){
          if(edge < 2){
            q = p
                } else {
                    q = endPoint[  parity[0]  ]  -   (p - startPoint)
                }

                parallel_result = Calculate point position (q, intermediate values of inside edge
            [parity[0]])

                if( parity[0] ){
                    define_point(perp_result ,  parallel_result)
                }
                else{
                    define_point(parallel_result  , perp_result)
                }

        }// end of interior edge walking
    }// end for all quad edges
}// end for all interior rings
```

Special case handling

For EVEN tessellation partition there can be a degenerate row of points left over instead of a complete ring, these must be handled specially, this happens when there are unequal number of points on the inside edges U and V

```
// U dimension is greater than the V and the parity of V is EVEN
if( (number_of_points_for_U   > number_of_points_for_V ) &&  insideParity[1] == EVEN){

    startPoint = numRings
    endPoint = number_of_points_for_U   - 1 – startPoint
    Set local parity to insideParity[0]

    for( p = startPoint ;  p <= endPoint ;  p++ ){

            result = Calculate point position (p, intermediate values of inside edge U )

        define_point(result  ,  0.5)
        }

  // V dimension is greater than the U and the parity of U is EVEN
} else if( (number_of_points_for_V   > number_of_points_for_U ) &&  insideParity[0] == EVEN){

    startPoint = numRings
    endPoint = number_of_points_for_V   - 1 – startPoint
    Set local parity to insideParity[1]

    for( p = startPoint  ; p <= endpoint ;  p++ ){

            result = Calculate point position (p, intermediate values of inside edge V )

        define_point(0.5, result)
        }
}
```

Triangle outer ring point generation

```
for(edge = 0 ;  edge <  3 ;  edge++ ){

    parity = edge  &  0x1
    startPoint = 0
    endpoint = num_points_for_tess_factor - 1

    for(p = startPoint ;  p < endPoint ;  p++){

                if(parity){
                    q = p
                } else{
                    q = endPoint - p
                }

        Set local parity = edge parity

            result = Calculate point position (q, intermediate values of edge )

        if(edge = 0){
            define_point(0, result)
                }
        if(edge = 1){
            define_point( result, 0)
                }
        if(edge = 2){
            define_point(result,  1.0 – result )
                }
        }
}
```

Triangle interior point generation

```
Set local parity = edge parity

startRing = 1

numRings = num_points_for_inside_tess_factor >> 1

for(ring = startRing ;  ring < numRings ;  ring++){
    startPoint = ring
    endPoint = num_points_for_inside_tess_factor  - 1 – startPoint

    for(edge = 0 ;  edge < 3 ;  edge++ ){

        parity = edge  &  0x1
        perpendicularAxisPoint = startPoint

            perp_result = Calculate point position (perpendicularAxisPoint, intermediate values of inside edge
        )

        perp_result = perp_result * 2/3
        perp_result = (perp_result +0.5)>>16




        for(int p = startPoint ;  p < endPoint ;  p++,){
            if(parity){
                q = p
            } else {
                q = endPoint - (p - startPoint)
            }

            parallel_result = Calculate point position (q, intermediate values of inside edge )


                        // reciprocal is the rate of change of edge-parallel parameters as they are pushed into
                the triangle
            deriv = 2


                // edge0 VW, has perpendicular parameter U constant
            if(edge = 0){
                define_point(perp_result  ,   parallel_result - (perp_result + 1)/deriv)
                        }


                // edge1 WU, has perpendicular parameter V constant
            if(edge = 1){
                define_point(parallel_result - (perp_result + 1)/derive  ,  perp_result  )
                        }


                 // edge2 UV, has perpendicular parameter W constant
            if(edge = 2){
                define_point(          parallel_result - (perp_result + 1)/derive  ,
                                                  1.0 -  (parallel_result - (perp_result + 1)/derive  )   )
                        }

        } // end of all points on the edge
    } // end of all edges
} // end of all rings
```

<u>Handle special case which requires a point in the center</u>

```
if(partition_type != ODD){
    define_point(0.333, 0.333)
}
```

Stitch Transition

This is called to determine the connectivity on the outermost ring. The tessellation levels can have different values for the two edges being connected and a look up table is used to determine when to move the points.

The input required for this part of the algorithm is

insideNumHalfTessFactorPoints
insideEdgeTessFactorParity

outsideNumHalfTessFactorPoints
outsideTessFactorParity

There are 32 hardcoded tables which control the connectivity.

0	{1}
1	{2,1}
2	{2,1,3}
3	{4,2,1,3}
4	{4,2,5,1,3}
5	{4,2,5,1,6,3}
6	{4,2,5,1,6,3,7}
7	{8,4,2,5,1,6,3,7}
8	{8,4,9,2,5,1,6,3,7}
9	{8,4,9,2,10,5,1,6,3,7}
10	{8,4,9,2,10,5,11,1,6,3,7}
11	{8,4,9,2,10,5,11,1,12,6,3,7}
12	{8,4,9,2,10,5,11,1,12,6,13,3,7}
13	{8,4,9,2,10,5,11,1,12,6,13,3,14,7}
14	{8,4,9,2,10,5,11,1,12,6,13,3,14,7,15}
15	{16,8,4,9,2,10,5,11,1,12,6,13,3,14,7,15}
16	{16,8,17,4,9,2,10,5,11,1,12,6,13,3,14,7,15}
17	{16,8,17,4,18,9,2,10,5,11,1,12,6,13,3,14,7,15}
18	{16,8,17,4,18,9,19,2,10,5,11,1,12,6,13,3,14,7,15}
19	{16,8,17,4,18,9,19,2,20,10,5,11,1,12,6,13,3,14,7,15}
20	{16,8,17,4,18,9,19,2,20,10,21,5,11,1,12,6,13,3,14,7,15}
21	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,1,12,6,13,3,14,7,15}
22	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,12,6,13,3,14,7,15}
23	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,6,13,3,14,7,15}
24	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,13,3,14,7,15}
25	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,26,13,3,14,7,15}
26	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,26,13,27,3,14,7,15}
27	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,26,13,27,3,28,14,7,15}
28	{16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,26,13,27,3,28,14,29,7,15}

```
29  {16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,26,13,27,3,28,14,29,7,30,15}
30  {16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,26,13,27,3,28,14,29,7,30,15,31}
31  {32,16,8,17,4,18,9,19,2,20,10,21,5,22,11,23,1,24,12,25,6,26,13,27,3,28,14,29,7,30,15,31}
```

```
if( insideEdgeTessFactorParity = ODD ){
        insideNumHalfTessFactorPoints = insideNumHalfTessFactorPoints  - 1
}

if(outsideTessFactorParity = ODD ){
        outsideNumHalfTessFactorPoints = outsideNumHalfTessFactorPoints  - 1
}

max_half_point = max(insideNumHalfTessFactorPoints, outsideNumHalfTessFactorPoints)
// select the appropriate table for calculations
PointPositionTable = hard coded tables [  max_half_point  - 2]
```

<u>Special case to begin</u>

```
if( outsideNumHalfTessFactorPoints > 0 ){
        // Advance outside
        DefineClockwiseTriangle(outsideP,  outsideP+1,  insideP)

        outsideP++
}
```

<u>Walk the first half of the strip</u>

```
if(max_half_point > 1) {
        for(i = 0 ;  i < max_half_point – 1  ;  i++) {
            if( (PointPositionTable[i] < insideNumHalfTessFactorPoints)){
                // Advance inside
                DefineClockwiseTriangle(insideP,  outsideP,  insideP+1)

                        insideP++
            }
                if((PointPositionTable[i] < outsideNumHalfTessFactorPoints)){
                // Advance outside
                DefineClockwiseTriangle(outsideP,  outsideP+1,  insideP)

            outsideP++
            }
        }
}
```

<u>Handle special cases for the middle</u>

```
if( (insideEdgeTessFactorParity != outsideTessFactorParity) || (insideEdgeTessFactorParity = ODD))  {

        if( insideEdgeTessFactorParity == outsideTessFactorParity ){
            // Quad in the middle
            DefineClockwiseTriangle(insideP,outsideP,insideP+1)

            DefineClockwiseTriangle(insideP+1, outsideP, outsideP+1)

                insideP++
            outsideP++
        }
        else if(insideEdgeTessFactorParity = EVEN) {
            // Triangle pointing inside
            DefineClockwiseTriangle(insideP, outsideP, outsideP+1)

                outsideP++
        }
        else {
            // Triangle pointing outside
            DefineClockwiseTriangle(insideP, outsideP, insideP+1)

            insideP++
       }
}
```

<u>Walk the second half of the strip</u>

```
if(max_half_point > 1) {
    for(i = max_half_point - 2 ;  i >= 0 ;  i--) {
            if((PointPositionTable[i] < outsideNumHalfTessFactorPoints)){
                    // Advance inside
                DefineClockwiseTriangle(outsideP, outsideP+1, insideP)

                        outsideP++
            }
            if((PointPositionTable[i] < insideNumHalfTessFactorPoints)) {
                // Advance outside
                DefineClockwiseTriangle(insideP, outsideP, insideP+1)

                        insideP++
            }
        }
}
```

<u>Special case to end</u>

```
if(outsideNumHalfTessFactorPoints > 0) {
        // Advance outside
        DefineClockwiseTriangle(outsideP, outsideP+1, insideP)

        outsideP++
}
```

Stitch Regular

This is the portion of the code used to generate connectivity for all the interior rings (i.e. everything except the outermost ring)

It needs the following terms as its inputs

bTrapezoid = Boolean used to indicate whether the potrion being stitched has 2 extra triangles at the beginning and the end
diagonals = type of diagonals

numInsideEdgePoints

<u>Special case to begin</u>

```
if( bTrapezoid ){
        DefineClockwiseTriangle(outsidePoint,  outsidePoint+1,  insidePoint)

        outsidePoint++
}
```

<u>Symmetric central part</u>

```
if(diagonals = DIAGONALS_INSIDE_TO_OUTSIDE){

    // all the diagonals point one way, this is the simplest case

        for( p = 0 ;  p < numInsideEdgePoints - 1;   p++ ){
            DefineClockwiseTriangle(insidePoint,  outsidePoint,  outsidePoint+1)

                DefineClockwiseTriangle(insidePoint,  outsidePoint+1,  insidePoint+1)

                insidePoint++
                outsidePoint++
        }
}


if(diagonals = DIAGONALS_INSIDE_TO_OUTSIDE_EXCEPT_MIDDLE){

    // First half
        for( p = 0 ;  p < numInsideEdgePoints/2-1 ;  p++ ){
            DefineClockwiseTriangle(outsidePoint,  outsidePoint+1,  insidePoint)

                DefineClockwiseTriangle(insidePoint,  outsidePoint+1,  insidePoint+1)

                insidePoint++
                outsidePoint++
        }

        // Middle
        DefineClockwiseTriangle(outsidePoint,  insidePoint+1,  insidePoint)
        DefineClockwiseTriangle(outsidePoint,  outsidePoint+1,  insidePoint+1)

        insidePoint++
        outsidePoint++

        p+=2


        // Second half
        for(  ;  p < numInsideEdgePoints ;  p++ ) {
            DefineClockwiseTriangle(outsidePoint,  outsidePoint+1,  insidePoint)

                DefineClockwiseTriangle(insidePoint,  outsidePoint+1,  insidePoint+1)

                insidePoint++
                outsidePoint++
        }
}


if(diagonals = DIAGONALS_MIRRORED){

    // First half, diagonals pointing from outside of outside edge to inside of inside edge
        for( p = 0 ;  p < numInsideEdgePoints/2 ;  p++ )  {
            DefineClockwiseTriangle(outsidePoint,  insidePoint+1,  insidePoint)

            DefineClockwiseTriangle(outsidePoint,  outsidePoint+1,  insidePoint+1)

                insidePoint++
                outsidePoint++
        }
    // Second half, diagonals pointing from inside of inside edge to outside of outside edge
        for( ;  p < numInsideEdgePoints-1 ;  p++ )  {
            DefineClockwiseTriangle(insidePoint,  outsidePoint,  outsidePoint+1)

            DefineClockwiseTriangle(insidePoint,  outsidePoint+1,  insidePoint+1)

                insidePoint++
                outsidePoint++
        }
}
```

<u>Special case to end</u>

```
if( bTrapezoid ){
    DefineClockwiseTriangle(outsidePoint,  outsidePoint+1,  insidePoint)
}
```

Handle Isolines

Isoline processing doesn't fit in very well into the triangle and quad partition type algorithms and is explained as a standalone section. It does use some common functions which have been elaborated earlier in this document.

Point generation

```
numPointsPerLine = num_points_for_tess_factor_line_detail

numLines = num_points_for_tess_factor_line_density – 1

for(line = 0 ; line < numLines  ;  line++){

    for(point = 0 ;  point <.numPointsPerLine ;  point++){

        Set local parity = line detail parity

        result_U = Calculate point position (point, intermediate values of line detail )

        Set local parity = line density parity

        result_V = Calculate point position (line, intermediate values of line density )

        define_point(result_U,  result_V)
    }
}
```

Connectivity generation

```
for(line = 0 ,  pointOffset = 0  ;  line <.numLines   ;  line++){

    for(point = 0  ;  point <.numPointsPerLine   ;  point++){

        if(point > 0){
            DefineLine(pointOffset-1,    pointOffset)
        }

                pointOffset++
    }
}
```

8 Performance

There is one WD block with the following performance characteristics

- Output 1 sub-packet per clock
- Draws for some prim types cannot be broken into sub-packets. I.e. if reset index is enabled, adjacent primitives, etc.

Each IA drives 2 VGT's and its peak rates are

- The IA output rate is 2 prims/clock in most cases
- Adjacent primitives are sent 1 per clock
- Patch primitives are sent 2 control points/clock

Each VGT has the following peak rates

- Input rate of one primitive/clock
- Output rate of one vertex or primitive per clock
- With GS scenario G enabled GS prims are launched at up to one prim/clock
- With GS scenario G the copy shader is loaded at one vert/clock and a new strip is started for each GS prim's output. This results a peak prim rate for pass through triangles of one vert/clock and 0.33 prims/clock.
- The maximum number of GS waves/VGT is controlled by the feature num_max_gs_thds.
- The maximum number of ES waves/VGT is determined by the number of unique vertices/prim and the features gsprim_buff_depth and gs_table_depth.
- The maximum number of VS waves/VGT is determined by multiple factors outside the WD/IA/VGT including the position buffer, parameter cache, primitive (PA clipp) FIFO and PA clipv FIFO. When the VS is a copy shader there's a hard limit of 64 VS waves/VGT.
- In a 4SE part, if tessellation type is triangle then the peak rate for TF 1 case with threadgroup size of 32 is 0.24 prims per clock. Each VGT will send 32 prims in 132 clock cycles (96 clocks (1 vert per clock) + 4 clocks (offchip_hs_dealloc events) +32 clocks (in triangle tessellation WD will send 1 patch to each VGT in 3 clocks but since there are 4 SEs each VGT will be idle for 1 clock cycle for a threadgroup size of 1. If threadgroup size is 32 then there are 32 idle clock cycles)). Vert rate will be 0.72 (96/132)

Prim groups greater than 256 as seen by the SC are likely to cause performance bottlenecks. Even smaller prim groups can see issues if the parameter cache is not large enough to hold an entire prim groups worth of VS waves.

9 Programming Model

#case internal
The location and filename for this document in R8xx is as follows:
//gpu8/doc/misc/8xx_programming_guidelines.doc
#endcase
#case kryptos
The location and filename for this document in R8xx is as follows: 8xx_programming_guidelines.doc
#endcase

10 Intellectual Property (IP)

No third party intellectual property is used in the design of the WD, IA, or VGT.

11 Area Estimates (TODO)

Area estimates will be added shortly

12 Power (TODO)

Power goals for the blocks will be added shortly.