PowerVR Series 5

Architecture Guide for Developers

Copyright © Imagination Technologies Ltd. All Rights Reserved.

This publication contains proprietary information which is subject to change without notice and is supplied 'as is' without warranty of any kind. Imagination Technologies and the Imagination Technologies logo are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.

Filename	:	PowerVR Series 5.Architecture Guide for Developers.1.0.8.External.doc
Version	:	1.0.8 External Issue (Package: POWERVR SDK REL_3.0@2149525)
Issue Date	:	30 Aug 2012
Author	:	Imagination Technologies Ltd

Contents

1. Introduction 4

2. What is PowerVR? 5

2.1. What is Tiling? 6
2.2. What is Deferred Rendering? 6

3. Graphics Architecture Comparison 7

3.1. Immediate Mode Rendering (IMR) 7
3.1.1. Obscured Fragments 7
3.1.2. Expensive Read-Modify-Write Operations 7
3.2. Tile Based Rendering (TBR) 8
3.2.1. On-chip Buffers 8
3.2.2. Obscured Fragments 8
3.2.3. Intermediate Buffer 8
3.3. PowerVR: Tile Based Deferred Rendering (TBDR) 9
3.3.1. Hidden Surface Removal (HSR) 9
3.3.2. Optimal Use of Memory Bandwidth 9

4. SGX Overview 10

4.1. Universal Scalable Shader Engine (USSE) 10
4.1.1. Coarse Grain Scheduler (CGS) 10
4.1.2. Thread scheduling 11
4.2. Parameter Buffer (PB) 12
4.2.1. Primitive Blocks 12
4.2.2. Display Lists 12
4.3. Tile Accelerator (TA) 13
4.4. Image Synthesis Processor (ISP) 14
4.5. Texture and Shading Processor (TSP) 16
4.6. SGX Micro Kernel 17
4.7. SGX-MP 18

5. SGX Hardware Schematic 19

6. Other considerations 20

6.1. Alpha Test/Fragment Discard 20
6.2. Blending 21
6.3. Parameter Buffer (PB) Management 21

7. Notable features 22

7.1. Internal True Colour™ 22
7.2. Full Screen MSAA 22
7.3. PVRTC Texture Compression 22

8. Related Materials 23

9. Contact Details 24

List of Figures

Figure 3-1 - IMR Rendering Pipeline 7
Figure 3-2 - TBR Rendering Pipeline 8
Figure 3-3 - TBDR Rendering Pipeline 9
Figure 4-1 - High Level Hardware Overview 10
Figure 4-2 - Dedicated Shader Modules 10
Figure 4-3 - Universal Scalable Shader Engine 10
Figure 4-4 – Thread Scheduling 11

Figure 4-5 – Parameter Buffer 12
Figure 4-6 - TA 13
Figure 4-7 - ISP 14
Figure 4-8 - Quake 3 Arena Screenshot 15
Figure 4-9 – Quake 3 Overdraw 15
Figure 4-10 - TSP 16
Figure 4-11 - SGX/SGX-MP Micro Kernel 17
Figure 4-12 - SGX-MP Example Tile Distribution 18
Figure 5-1 - SGX Hardware Schematic 19
Figure 6-1 - Alpha Test/Discard 20

1. Introduction

The purpose of this document is to provide graphics programmers with an overview of the PowerVR Series 5 (SGX & SGX-MP) graphics hardware architecture, while also highlighting why some performance recommendations make such a significant difference to graphics rendering on PowerVR platforms. Furthermore, the information in this document outlines the purpose of each hardware module for which PVRTune provides performance counters.

The PowerVR Series 5 architecture is covered by a broad portfolio of patents, the result of more than 15 years research and development by Imagination Technologies. More than 500 million devices incorporating PowerVR graphics have been shipped (as of July 2011) and hundreds of thousands of applications are running on PowerVR graphics-powered platforms across every major operating system and CPU architecture.

2. What is PowerVR?

PowerVR™ graphics is the brand name of the family of graphics IP cores from Imagination Technologies that use Imagination's unique "Tile Based Deferred Rendering" (TBDR) architecture. The core design principle behind the TBDR architecture is to reduce the system memory bandwidth required by the GPU to a bare minimum. As transfer of data between system memory and the GPU is one of the biggest causes of GPU power consumption, any reduction that can be made in this area will allow the GPU to operate at a lower power. Additionally, the reduction in system memory bandwidth use and the hardware optimizations associated with it (such as using on-chip buffers) can boost application performance. Because of this development strategy, PowerVR graphics cores have become the dominant series of graphics cores in the mobile and embedded devices market.

Whereas a traditional Immediate Mode Renderer (IMR) renders all objects within the screen's boundaries and relies on a Z-Buffer to sort the end results, the PowerVR TBDR approach determines up-front what is and isn't visible, allowing the hardware to only render what is necessary. Although current day IMRs incorporate advanced techniques to reduce some of the issues that are inherent within the architecture's design, such as early Z testing to reduce overdraw, there are still many ways in which the TBDR architecture provides a more efficient solution to these problems.

Understanding the differences between these architectures is crucial when optimizing applications for PowerVR graphics hardware.

2.1. What is Tiling?

Tiling is a technique that can be implemented in graphics hardware to process subsections of a render at a time instead of the entire scene. The main benefit of this approach is that fast, on-chip memory can be used during the render for colour, depth and stencil buffer operations, which allows a significant reduction in system memory bandwidth over traditional IMR architectures.

Tiling involves two key rendering phases; geometry processing and rasterization. Once submitted geometry has been transformed into screen space coordinates, and the hardware has determined which geometry has fallen within the bounds of a tile, the hardware renders each tile and flushes the resultant colour buffer out to a frame buffer in system memory until the entire scene has been rendered.

2.2. What is Deferred Rendering?

Deferred rending splits the per-tile rendering process into two stages; Hidden Surface Removal (HSR) and shading. Pixel-perfect, submission order independent HSR is performed within each tile so that the only fragments processed are those that will contribute to the final rendered image. In an entirely opaque scene, overdraw will be removed completely by the HSR of TBDR hardware.

3. Graphics Architecture Comparison

3.1. Immediate Mode Rendering (IMR)

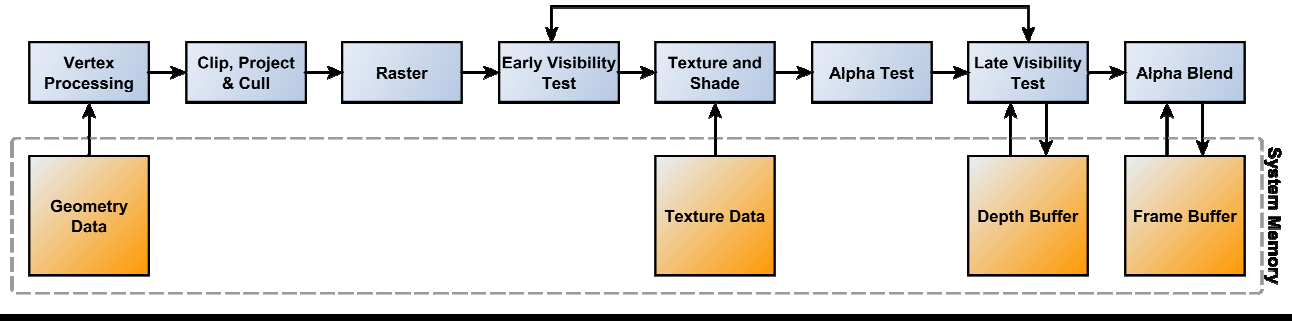

Figure 3-1 - IMR Rendering Pipeline

A traditional Immediate Mode Rendering (IMR) architecture is given its name because each submitted object travels through the entire pipeline immediately. Due to the brute force approach of the design, there are a number of weaknesses that result in inefficient use of the available processing power and memory bandwidth.

3.1.1. Obscured Fragments

As an IMR processes each object as it's submitted, any fragment that has been written to the frame buffer can be later overwritten by other processed objects that cover the fragment. Unnecessarily performing texturing and shading for fragments that will not affect the render is commonly referred to as overdraw. Overdraw can quickly cause bottlenecks in graphics applications running on IMR hardware because a large portion of the hardware's processing time and system memory bandwidth (particularly for texture fetches) is wasted calculating the colour of fragments that will not affect the rendered frame. Most modern IMR architectures utilise Early-Z techniques to perform depth tests early in the graphics pipeline to reduce the amount of overdraw in a render (as shown in Figure 3-1 – Early Visibility Test/Late Visibility Test), but applications can only fully benefit from this optimization if geometry is always submitted to the hardware in front to back order (this requires per frame sorting for scenes with moving cameras and/or geometry). Even when an application attempts to submit all objects from front to back, it is still unlikely that overdraw will be eliminated completely.

3.1.2. Expensive Read-Modify-Write Operations

As IMRs store all colour, depth and stencil buffers in system memory, regular Read-Modify-Write operations to these buffers can quickly induce a large system memory bandwidth overhead. As the memory bandwidth required for these operations is directly related to the precision of these buffers, application developers may find they have to sacrifice the accuracy of their colour, depth and stencil operations to alleviate memory bandwidth bottlenecks. In addition to the time it takes to transfer data over the system's memory bus, regular Read-Modify-Write operations for these buffers will increase the power consumption of the IMR hardware, which will in turn cause the application to have a bigger impact on the life of battery powered devices.

3.2. Tile Based Rendering (TBR)

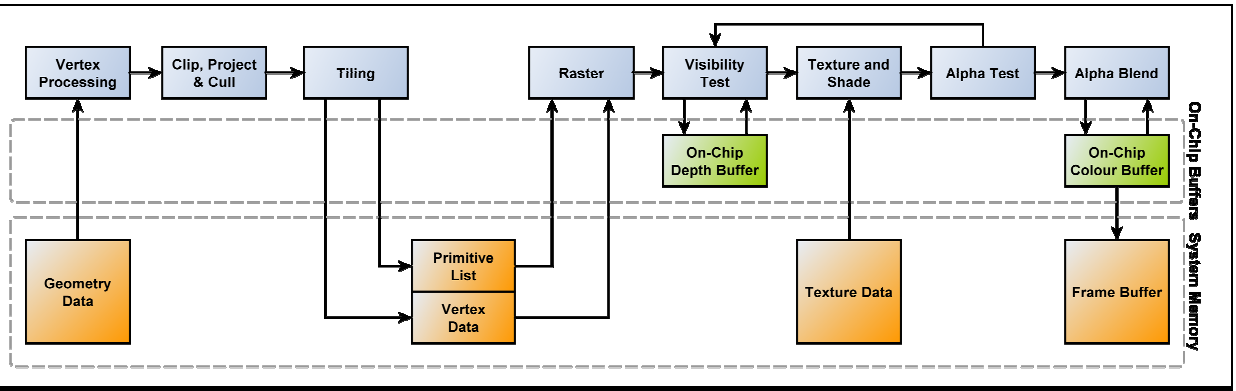

Figure 3-2 - TBR Rendering Pipeline

In tile based rendering hardware, rasterization is performed on a per tile basis instead of rasterizing the entire frame buffer as a traditional IMR would. To facilitate this, all submitted geometry is transformed into screen space coordinates and stored in an intermediate buffer in system memory. Once this has been done, the hardware generates a per-tile list of pointers to geometry that lies within the bounds of the tile. This enables the hardware to only retrieve relevant geometry data when rendering each tile, which keeps memory bandwidth requirements for intermediate buffer access to a minimum. The geometry within each tile is then processed in submission order as per an IMR and the rendered tiles are written out to the frame buffer in system memory on completion.

3.2.1. On-chip Buffers

Read-Modify-Write operations for the colour, depth and stencil buffers are performed using fast on-chip memory instead of relying on repeated system memory access, as IMRs do. Memory bandwidth is required to write the colour buffer into frame buffer memory, but depth and stencil buffer data only needs to be written out if it has to be preserved for later use.

3.2.2. Obscured Fragments

Although the TBR approach improves on the traditional IMR design, it does not attempt to reduce overdraw in the render. When rendering each tile, geometry is processed in submission order, so obscured geometry will still be processed and texture data for invisible fragments will still be fetched. Once again, Early-Z techniques can be used to reduce overdraw, but the full benefit of these can only be utilised if applications sort and submit geometry from front to back.

3.2.3. Intermediate Buffer

As rendering is split into geometry processing and rasterization, the hardware requires buffer space where intermediate data can be stored. Although this storage space is required and the hardware must calculate which geometry is visible in each tile, the benefit of faster on-chip Read-Modify-Write operations and reduced system memory bandwidth use outweighs the additional required workload.

3.3. PowerVR: Tile Based Deferred Rendering (TBDR)

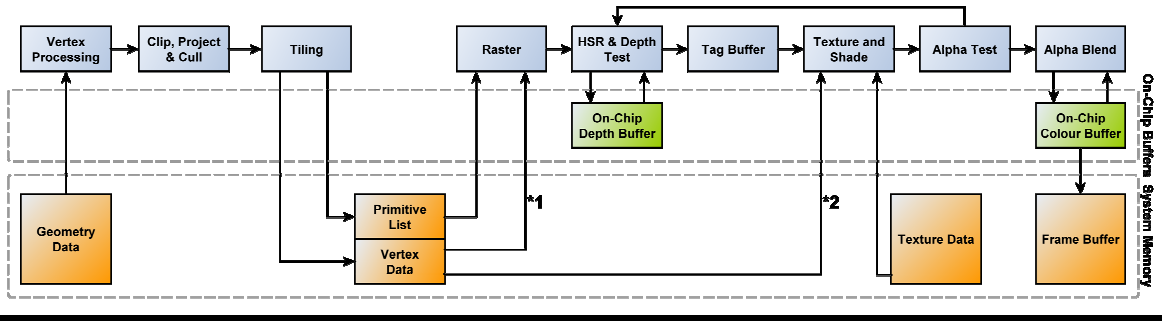

Figure 3-3 - TBDR Rendering Pipeline

The TBDR architecture improves on the memory bandwidth reductions the TBR architecture provides by incorporating pixel-perfect, submission order independent Hidden Surface Removal (HSR). The efficiency of the HSR is such that overdraw can be removed entirely for completely opaque renders, which allows the hardware to significantly reduce its system memory bandwidth requirements for fetching fragment data (for both texture and intermediate buffer data) as well as its fragment processing workload.

3.3.1. Hidden Surface Removal (HSR)

Although Early-Z and similar techniques have been developed to reduce the effects of overdraw in IMR and TBR architectures, overdraw still proves to be a common cause of bottlenecks in many graphics applications running on these architectures. One of the major objectives when developing the TBDR architecture was to provide a sophisticated solution to this problem to remove redundant system memory bandwidth use and fragment processing. Doing so not only improves performance, but also allows the hardware to minimise power consumption.

As the HSR process of the TBDR hardware is performed on a per-pixel basis independently of geometry submission order, overdraw can be removed entirely for completely opaque renders (as well as being hugely beneficial in renders using alpha blend and discard/alpha test). In real-world use cases, this approach is much more efficient than Early-Z techniques used in other architectures as they can only approach the same level of efficiency as the TBDR HSR process if an application perfectly submits all geometry from front to back. Even in these cases, intersecting objects can cause overdraw when Early Z is used, whereas TBDR HSR will still completely remove overdraw in these cases.

As the HSR process ensures the only fragments processed are those that will contribute to the final render, texture memory bandwidth use is reduced to the bare minimum (only relevant texture data is fetched). The HSR achieves this by utilising a block of memory called the Tag Buffer that tracks visible fragments within the tile (see Section 4.4). The HSR process only fetches screen space position data for the geometry within the tile (*1 in Figure 3-3) and all other geometry data fetch operations are deferred until later in the pipeline (*2 in Figure 3-3). By doing so the hardware can ensure that the only additional geometry data that is fetched (beyond screen-space position data required by the HSR) is the data required to texture and shade visible fragments.

From an application developer's point of view, the TBDR HSR technique has the additional benefit of removing the need to sort geometry into front to back order on a per frame basis. This reduces CPU workload and gives the developer more freedom to submit geometry in other ways that will benefit the hardware (such as batching draw calls and minimizing state changes to remove redundant API calls and process objects more efficiently).

3.3.2. Optimal Use of Memory Bandwidth

As with TBRs, TBDR hardware uses on-chip buffers for colour, depth and stencil buffer Read-Modify-Write operations. This allows the hardware to perform these operations faster than a traditional IMR would allow, while also significantly reducing the memory bandwidth overhead.

4. SGX Overview

The PowerVR architecture consists of several core modules that convert application submitted 3D data into a rendered image (as shown in Figure 4-1). These modules are the Tile Accelerator (TA), Image Synthesis Processor (ISP) and the Texture & Shading Processor (TSP). The core modules make use of the Universal Scalable Shader Engine (USSE) and Parameter Buffer (PB) shared components to keep the design as flexible and scalable as possible.

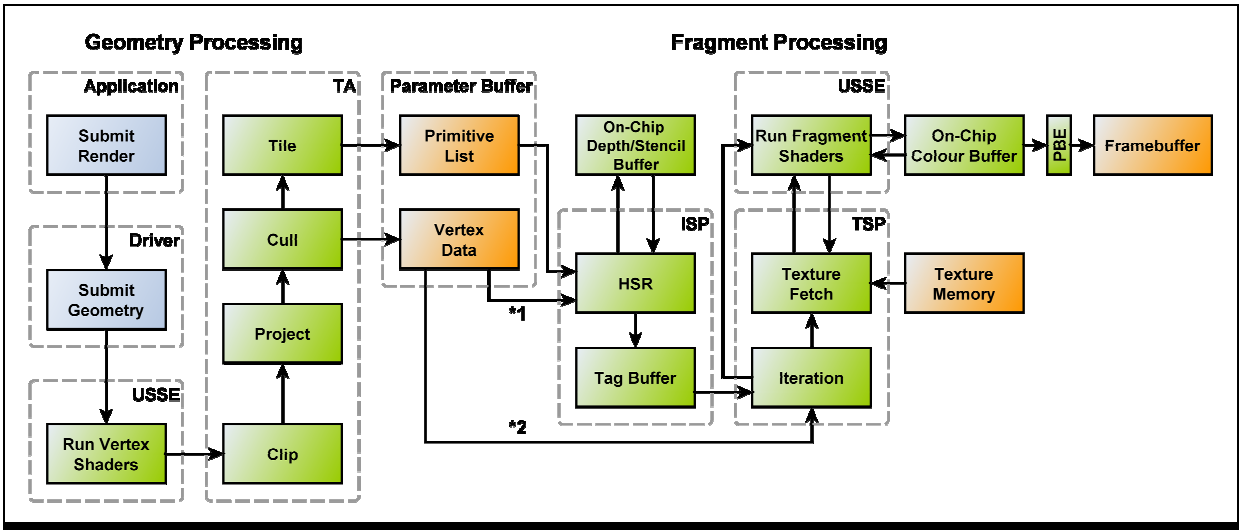

Figure 4-1 - High Level Hardware Overview

4.1. Universal Scalable Shader Engine (USSE)

The USSE is a flexible, multi-threaded processor capable of executing vertex, fragment and GP-GPU instructions. As vertex and fragment processing tasks are completely decoupled (all geometry processing is done, then rasterization begins), the USSE's thread scheduler can automatically load balance a queue of tasks, which ensures idle time is kept to a minimum and latency is hidden as much as possible.

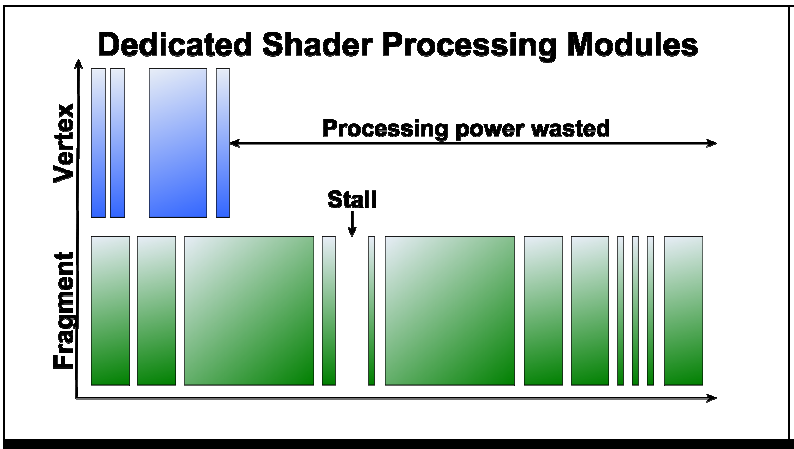

Figure 4-2 - Dedicated Shader Modules

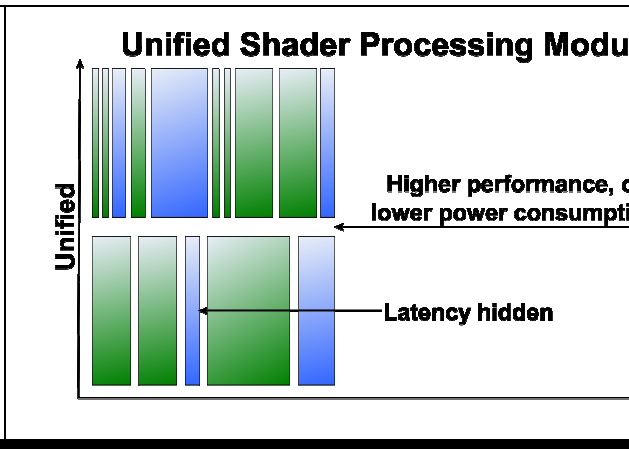

Figure 4-3 - Universal Scalable Shader Eng

4.1.1. Coarse Grain Scheduler (CGS)

The CGS takes vertex, fragment and GP-GPU jobs as input and breaks them down into tasks (smaller units of work) that can be submitted to available USSE execution units.

4.1.2. Thread scheduling

Each USSE execution unit in a given SGX graphics core has its own thread scheduler. Each scheduler manages 16 threads, 4 of which can be active at a given time (Figure 4-4). The benefit of managing this many threads simultaneously is that when an active thread stalls, the scheduler can suspend it and schedule in a previously suspended thread with a zero cycle scheduling overhead. This approach keeps latency to a minimum as threads will still be processed while a stalled thread resolves. When an active thread completes a task, the thread scheduler will retrieve a new task from the CGS.

This efficient, hardware based, data driven thread scheduling can benefit applications by hiding the latency induced by any stalls, such as dependent texture reads and branching in shaders. Applications can take advantage of this by placing as much work as possible ahead of the point at which the shader is likely to stall, which will increase the number of instructions the hardware can use to mask latency induced by the stalls.

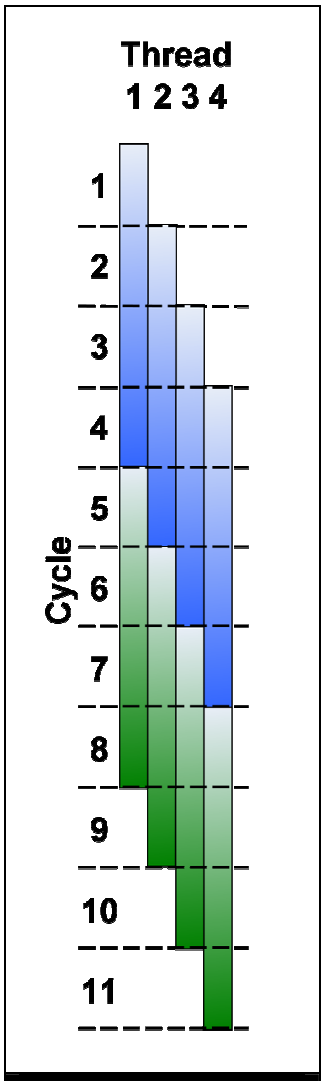

Figure 4-4 – Thread Scheduling

4.2. Parameter Buffer (PB)

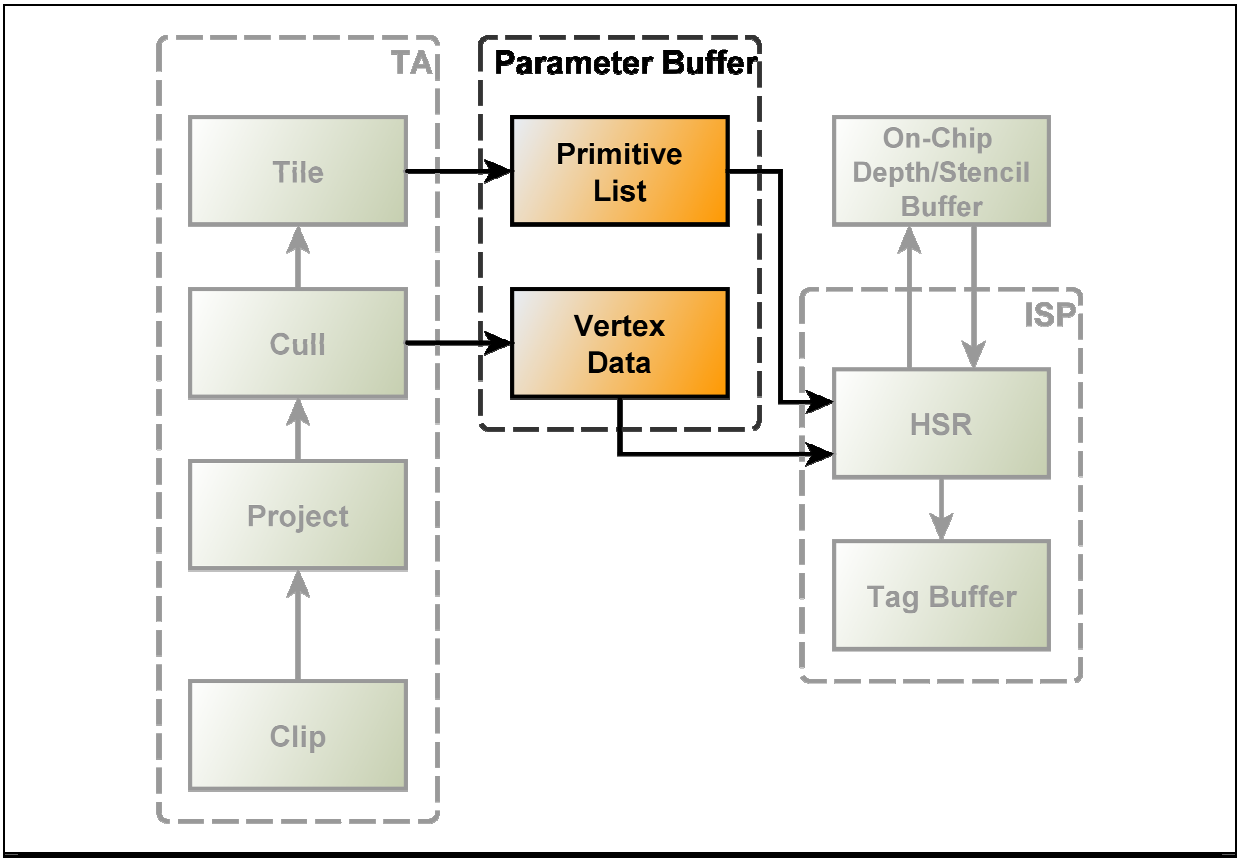

Figure 4-5 – Parameter Buffer

The Parameter Buffer is an area of system memory that is used to store intermediate data, which allows the geometry processing and rasterization stages of rendering to be separated. The hardware and drivers handle this storage space entirely (applications have no control over it).

The Parameter Buffer consists of Primitive Blocks and Display Lists.

4.2.1. Primitive Blocks

A Primitive Block is a list of primitives, where each primitive consists of its indices and the screen space x and y positions of its vertices. The data contained within a Primitive Block is used to reference transformed geometry data that has been clipped, projected and culled by the TA and written into Parameter Buffer memory. Display Lists are per-tile linked lists that reference the Primitive Blocks of objects that lie within the tile's boundaries.

4.2.2. Display Lists

To ensure that tiles minimise the amount of geometric data they have to fetch, Display Lists use primitive and vertex masks (e.g. for a triangle strip that goes beyond the boundaries of the tile, masks are used so the hardware will only fetch data for triangles that cover the tile).

4.3. Tile Accelerator (TA)

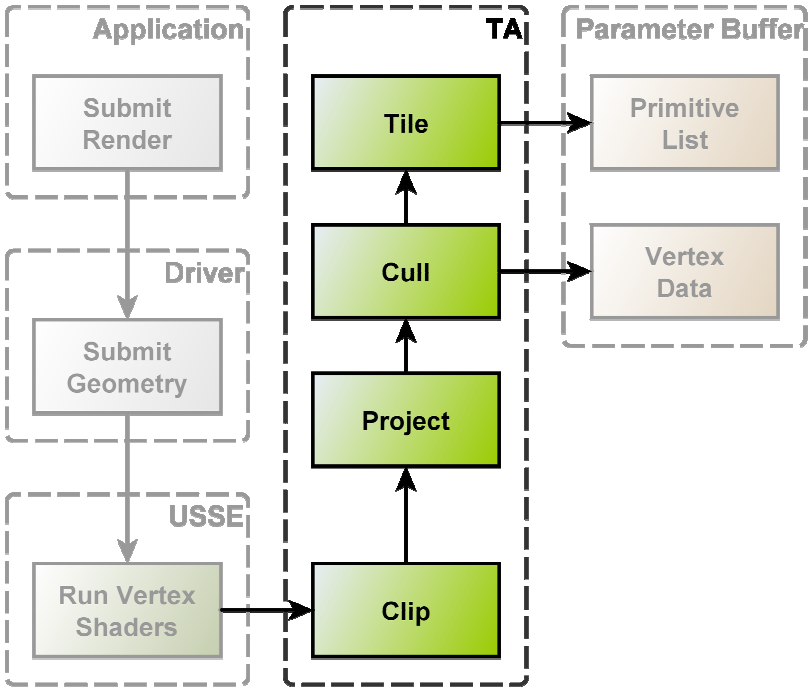

Figure 4-6 - TA

The TA takes geometric data that has been transformed by the USSE as input and clips, projects and culls it. In a shader based graphics API, such as OpenGL ES 2.0, the USSE executes vertex shaders on the geometry to transform and perform other per-vertex operations, such as lighting, before the resultant data is given to the TA. If polygons have been deemed visible after these tests, the TA updates all of the Display Lists of tiles that the object covers to reference it and writes out the transformed data into a Primitive Block (Figure 4-6).

The number of tiles required to complete the render is determined to be the resolution of the render pass divided by the tile size – where the tile size is a fixed value that is hardware specific (e.g. many SGX cores use a 16x16 pixel tile size). Larger tile sizes do improve performance (e.g. there will be fewer tiles to process, fewer Display Lists to update, single cycle scan-line depth tests can yield greater benefit etc.), but they require the on-chip memory requirements of the graphics core to increase. The tile size used in a given graphics core is a balance between required performance and the cost of the additional resources.

4.4. Image Synthesis Processor (ISP)

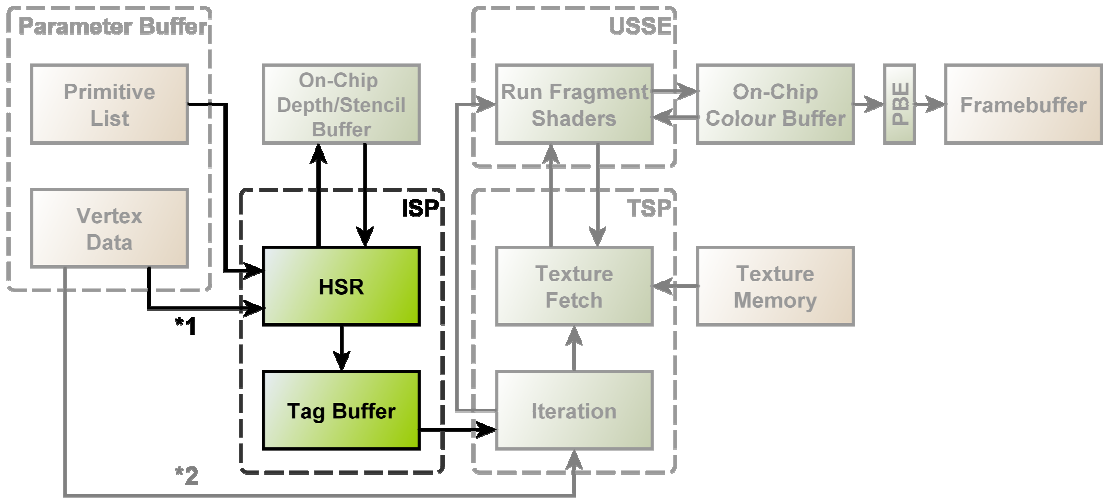

Figure 4-7 - ISP

The ISP is responsible for per-tile HSR (Hidden Surface Removal) to ensure that the only fragments processed by the TSP (Texture and Shading Processor – see section 4.5) are those that will affect the rendered image. To do this, the ISP processes all of the triangles referenced in the tile's Display List one by one, calculating the triangle equation and, if depth testing is enabled, projecting a ray at each position in each triangle to retrieve accurate depth information for each fragment that the primitive covers within the tile. The calculated depth information is then compared with the values in the tile's on-chip depth buffer to determine if the fragments are visible. The ISP uses a Tag Buffer to track all visible fragments and the primitives that they belong to. To minimize data fetch, the ISP only retrieves position information for geometry that is required to render the tile (vertex and primitive masks ensure the smallest possible data set is retrieved). When all of the primitives that the tile's Display List references have been processed, the ISP submits fragments to the TSP in groups of fragments that belong to the same primitive, rather than submitting visible fragments on a per scan line basis. Doing so allows the hardware to improve the efficiency cache access and fragment processing.

Figure 4-8 and Figure 4-9 are two screenshots from Quake 3; the first screenshot is a normal screenshot from a scene in Quake 3 while the second renders everything translucently to get an idea of the overdraw in the scene. Although Quake 3 uses Binary Space Partitioning (BSP) and portal techniques to attempt to reduce overdraw, the scene complexity is such that overdraw is often quite high. As an example we measured overdraw of the Quake 3 demo001 at 3.39 (i.e. each pixel on the screen is drawn an average of 3.39 times). As the ISP can eliminate overdraw entirely for opaque pixels (i.e. those without any blending or alpha test/discard), TBDR hardware can reduce overdraw further than these BSP and portal techniques are capable of alone.

The benefit of the ISP, compared to Early-Z techniques in IMR and TBR architectures, is that overdraw in 3D applications can be significantly reduced independently of the submission order of draw calls, as the hardware has full visibility of the scene within the tile when tests are performed. In entirely opaque render passes, this technique can reduce overdraw to zero, meaning the number of fragments processed is the same as the resolution of the render's viewport. Unlike Early Z techniques used in other architectures, this approach does not require applications to sort and submit geometry in front to back order to get the most out of the technique. Because of this, developers have no need for per-frame CPU sorting algorithms to further reduce overdraw and it also gives developers the freedom to submit their geometry in other ways that can benefit the render (such as batching draw calls by render state).

Figure 4-8 - Quake 3 Arena Screenshot

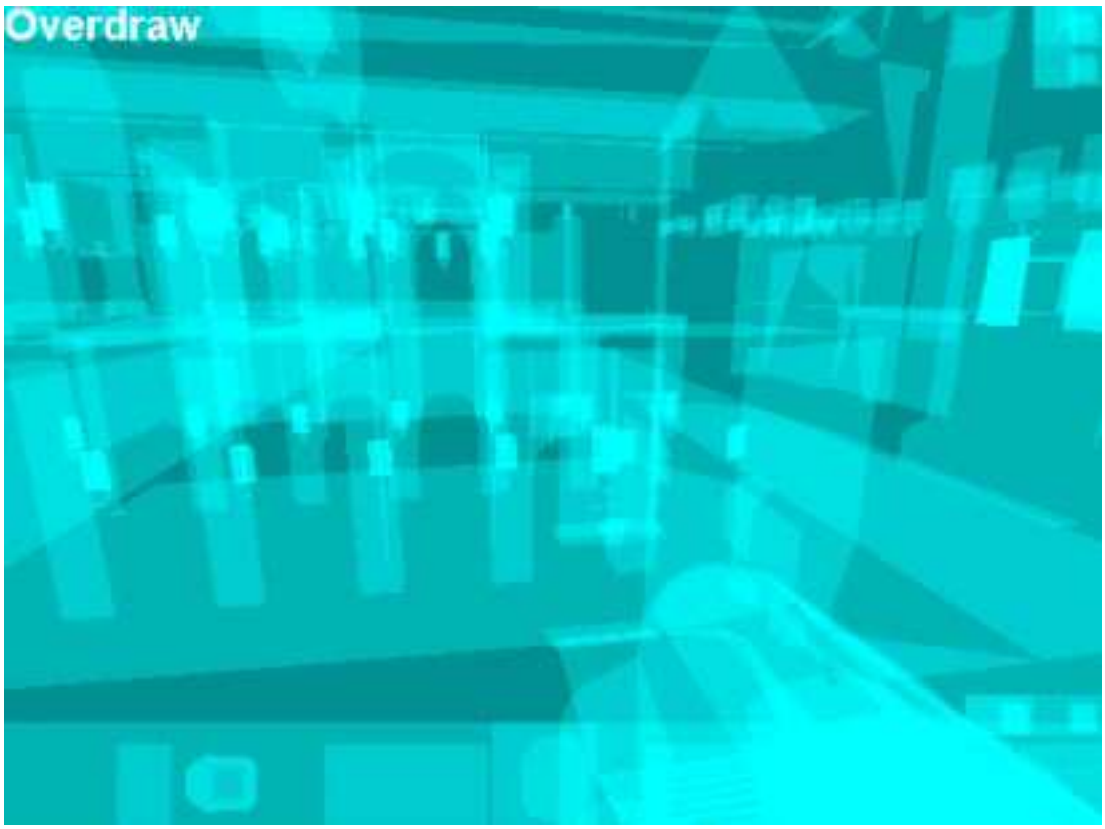

Figure 4-9 – Quake 3 Overdraw

4.5. Texture and Shading Processor (TSP)

Rather than texturing and shading fragments itself, the job of the TSP is to schedule fragment processing tasks (performed by the USSE), iterations, and texture data pre-fetch (Figure 4-10). If texture coordinates are calculated during iteration (e.g. texture coordinate varyings used in GLSL shaders), they can be used to pre-fetch texture data, which ensures the data is available when the USSE comes to process the fragments. Once iteration has been performed, the TSP can submit groups of visible fragments to the USSE via the CGS to be textured and shaded (visible fragments are grouped by the ISP during the HSR process). When using a shader based graphics API, such as OpenGL ES 2.0, fragment shaders are executed at this stage to calculate fragment colour.

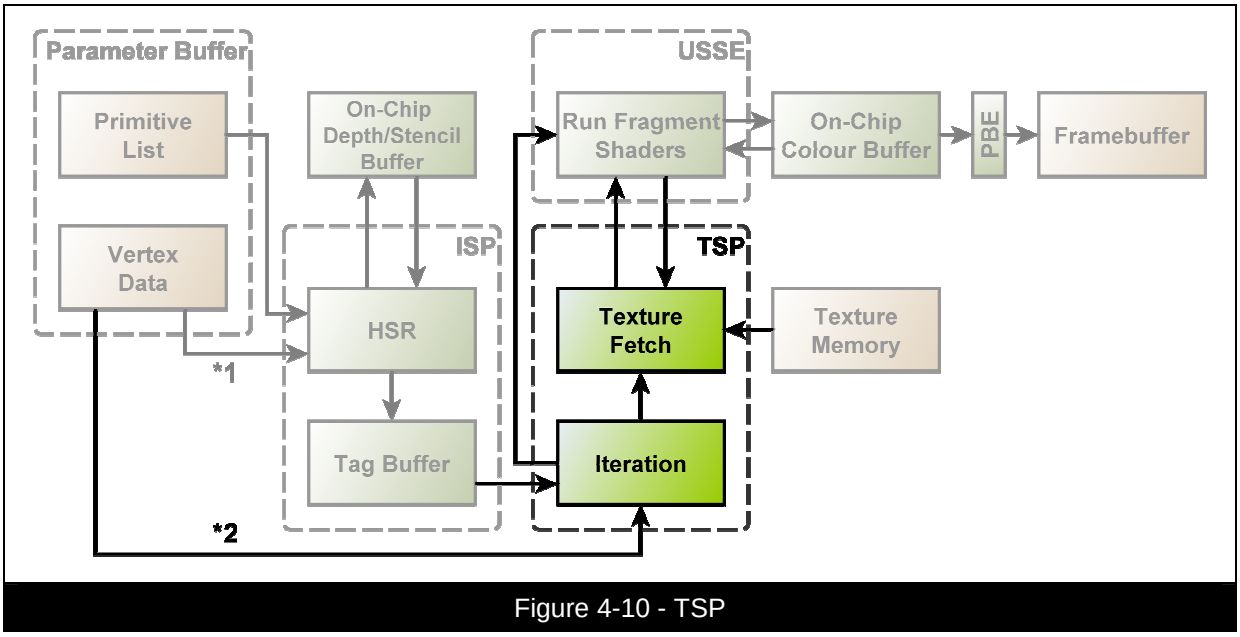

Figure 4-10 - TSP

4.6. SGX Micro Kernel

The Micro Kernel is specialised software that the hardware runs on the available USSE general purpose execution units. Unlike many other graphics architectures, the Micro Kernel enables the GPU to handle internal interrupt events generated by the GPU entirely on the GPU. By utilising the Micro Kernel in this way, the graphics core has minimal impact on CPU load, performance is kept as high as possible (increased parallelism between the CPU and GPU) and synchronisation issues between the CPU and GPU are decreased (as highlighted in Figure 4-11).

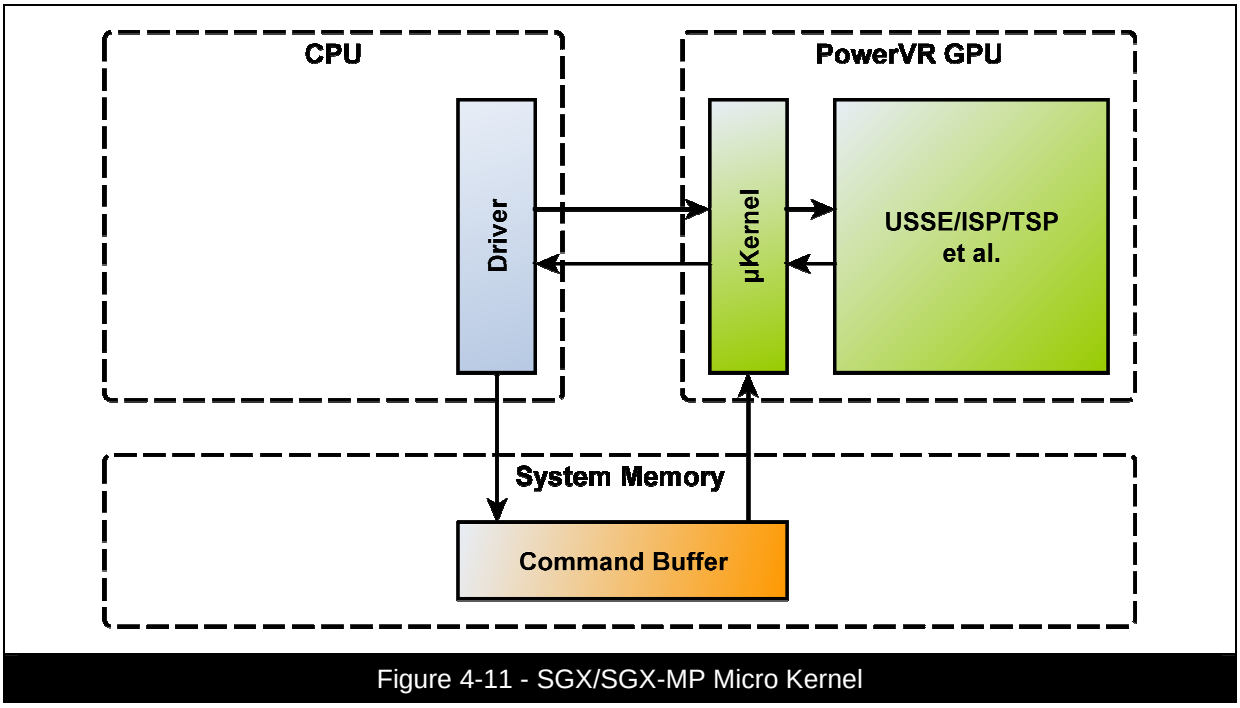

Figure 4-11 - SGX/SGX-MP Micro Kernel

4.7. SGX-MP

The SGX-MP architecture is designed to scale as close to linearly as possible as more graphics cores are used. In typical real-world performance conditions (running well known game engines, graphics applications, benchmarks etc.), each additional core runs at 95%+ the efficiency of a single core. Additionally, adding another core to the system only increases the overall memory bandwidth for a frame by <1%.

In SGX-MP devices, the hardware and drivers have total control over the multi-core logic. For this reason, applications that already run on SGX devices will be able to run without modification on SGX-MP devices.

When processing geometry, the hardware splits objects into chunks of work that can be distributed evenly between all available graphics cores. By doing this, the hardware can ensure that idle time in the cores is kept to a minimum.

During the rasterization, texturing and shading process, each additional graphics core allows another tile to be processed in parallel. To enable this, the hardware manages a queue of all tiles that need to be processed and each graphics core fetches and processes a tile from this queue. By taking this approach, the hardware keeps idle time to a minimum and avoids creating hotspots that could occur with static tile distribution (e.g. one core may have more work than others, which would cause the other cores to go idle). A tile task submitted to a graphics core includes HSR, texturing and shading.

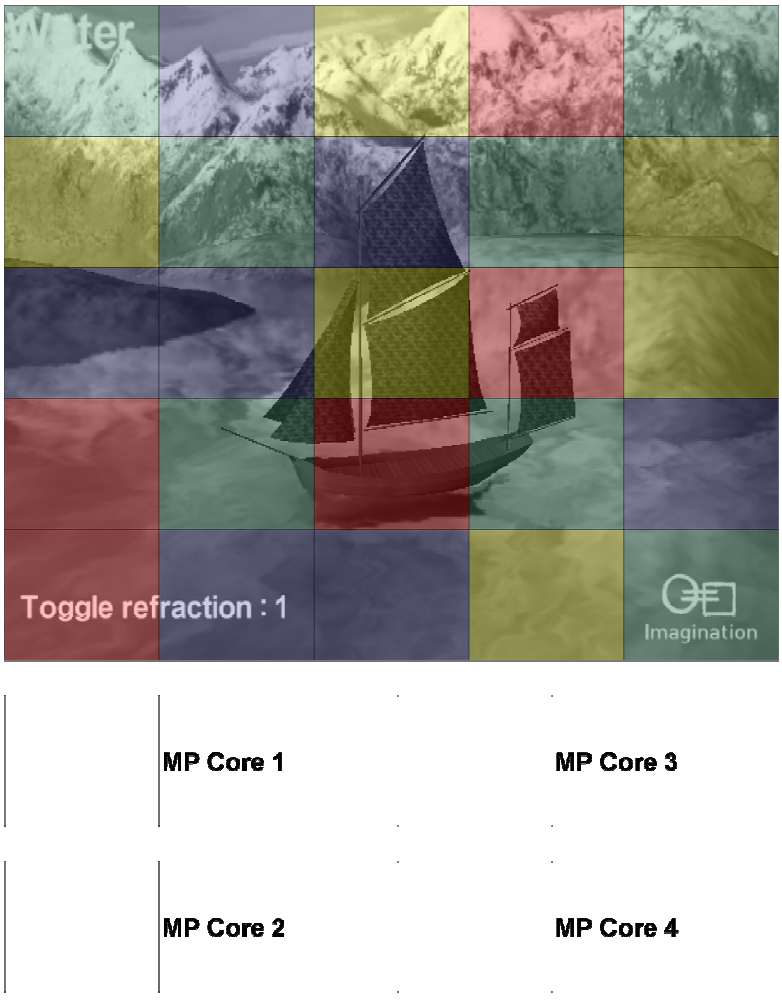

Figure 4-12 - SGX-MP Example Tile Distribution

5. SGX Hardware Schematic

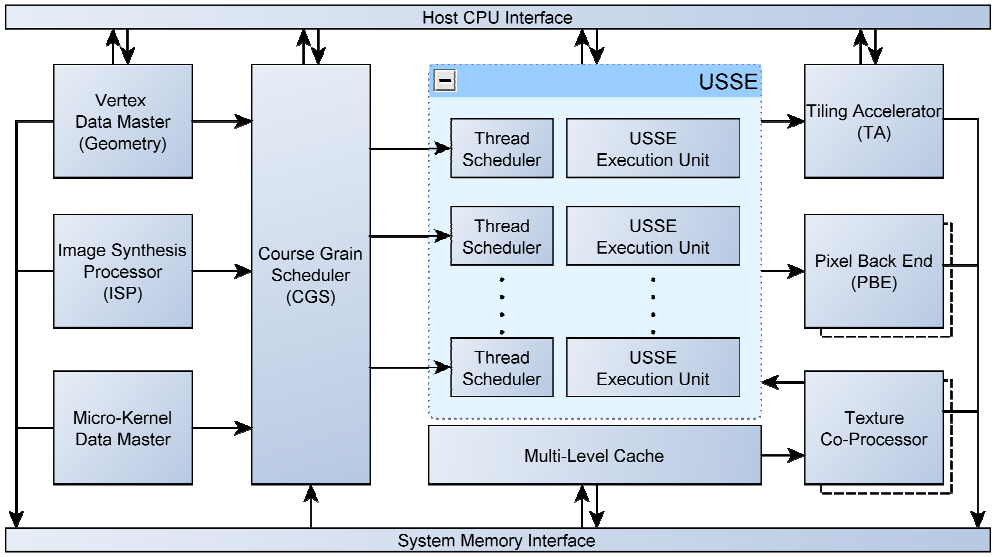

Figure 5-1 - SGX Hardware Schematic

The first stage of the processing is for the driver to submit geometry data from system memory to the graphics core's Vertex Data Master. This module takes the stream of vertex data and submits vertex processing jobs to the CGS, which in turn breaks down the jobs into tasks (smaller units of work) that can be distributed to the Thread Scheduler's of the available USSE Execution Units. The USSE Execution Units then process the vertex tasks that have been given to it and the resultant transformed data is given to the TA. The TA applies clipping, projection and culling to the geometric data and, if it passes these tests, the data will be written into a Primitive Block in Parameter Buffer memory. The TA also updates the Display Lists of all tiles the object covers to reference the object's Primitive Block. As the Display Lists are updated, the TA creates vertex and primitive masks to restrict the amount of geometry data that will be fetched when per-tile operations are performed.

Once all of the scene's geometry has been processed by the TA, per-tile rasterization, texturing and shading can begin. A tile task is first given to the ISP, where HSR, depth and stencil tests are performed. Once this is done, iteration is performed for visible primitives within the tile and a fragment processing job is submitted to the CGS for each group of visible fragments (where the grouping has been done by the ISP during the HSR process – see section 4.4 for more information). When iteration is performed for texture coordinates, the resultant values are sent to the Texturing Co-Processor to pre-fetch texture data.

The CGS breaks each fragment processing job into tasks that can be distributed to the Thread Schedulers of available USSE Execution Units. When dependent texture reads are performed by the USSE Execution Units, the thread will be suspended and a texture fetch task will be issued to the Texturing Co-Processor. When the tile's fragment processing is complete, the PBE (Pixel Back End) accesses the tile's buffers and applies final operations to the rendered image before it is flushed to the frame buffer in system memory, e.g. applying dither patterns when required, combining MSAA sub-sampled etc.

6. Other considerations

6.1. Alpha Test/Fragment Discard

Performing alpha test/discard in an application removes some of the advantage of overdraw reducing techniques such as HSR and Early Z. The reason for this is that fragment visibility is not known until per fragment operations are performed, which means the hardware has to assume that all fragments for that object may contribute to the frame buffer colour and process them anyway. As more objects using discard overlap a given pixel, more overdraw will be introduced (all layers using discard have to be processed as the hardware doesn't know which, if any, of the fragments will contribute to the frame buffer colour).

For an object using discard, the ISP assumes all fragments will be visible and submits them all to the TSP. The TSP then submits all of the fragments to the USSE to be processed and, potentially, discarded. As fragments are discarded, the USSE sends fragment visibility information back to the ISP so that depth and stencil buffers can be updated accordingly (as highlighted in Figure 6-1). To improve the performance of discard, the hardware utilises a number of optimizations to reduce the workload as early as possible in the pipeline.

Applications should avoid the use of alpha test/discard where possible as objects using this feature cannot fully benefit from the HSR process. When an application needs to use discard, all opaque objects should be rendered before objects using discard so that obscured fragments can be removed from the render, e.g. if a wall of a building is partially obscuring a tree object using discard, rendering the opaque building first will ensure the hardware only has to process the area of the tree that isn't obscured.

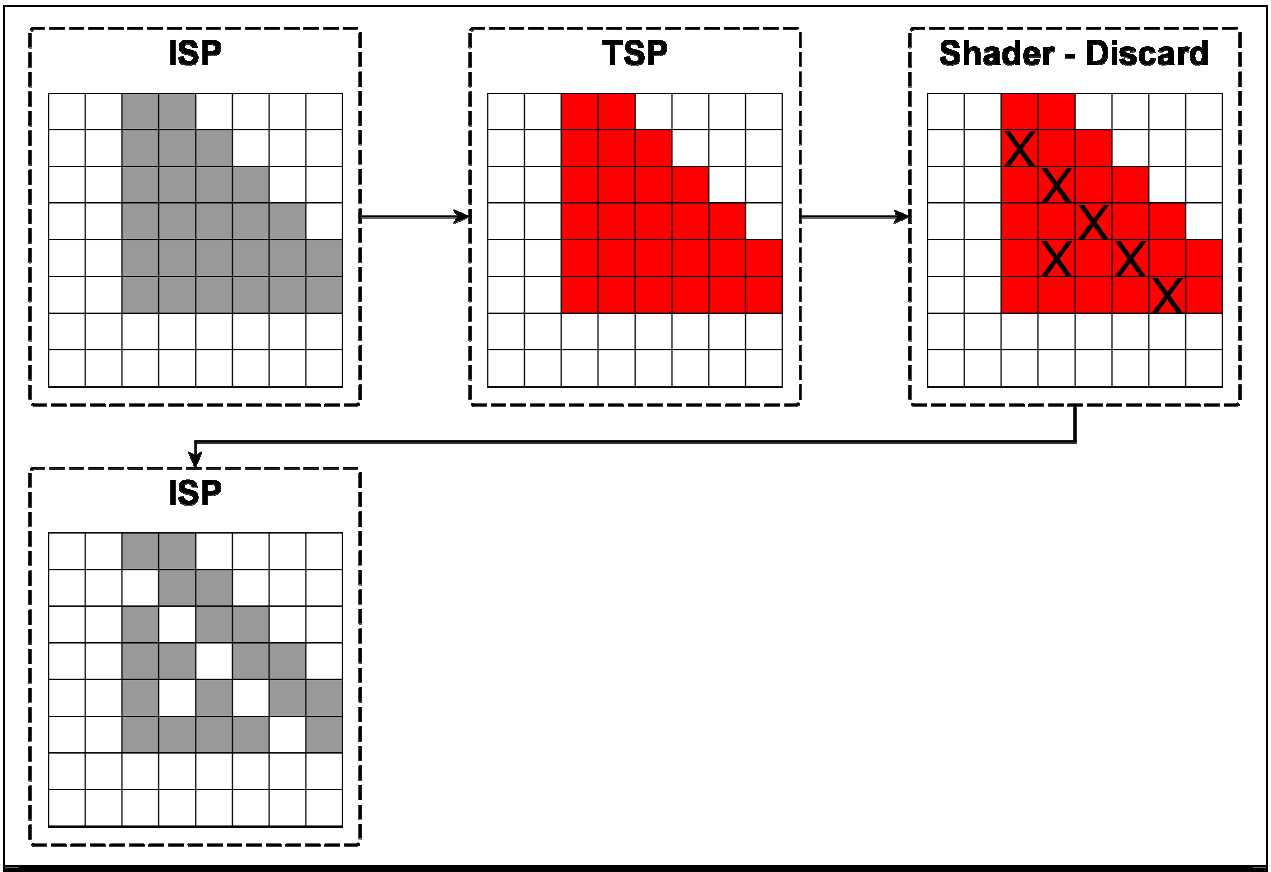

Figure 6-1 - Alpha Test/Discard

6.2. Blending

To render blended objects correctly, the hardware has to process each object individually as they may all contribute to the frame buffer's colour. Similarly to discard operations, the hardware utilises a number of optimizations to reduce the workload as early as possible in the pipeline.

All opaque objects should be drawn before blended objects to achieve the maximum benefit from the HSR process. When an application needs to use discard and blending, opaque objects should be submitted first, then discard objects, then blended objects.

As all blending operations are performed using on-chip memory, they can be executed very quickly and, unlike many other architectures, do not waste system memory bandwidth.

6.3. Parameter Buffer (PB) Management

Although the TBDR architecture provides a very efficient solution for processing 3D graphics, where overdraw and memory bandwidth use can be significantly reduced compared to other architectures, it does require system memory to be reserved for intermediate data stored in the PB. As a finite amount of memory is allocated for the PB the hardware has to cope when this memory has been filled but the render is still incomplete. The TBDR solution to this is to flush the render when the buffer fills; this allows the hardware to free memory in the PB associated with objects that are rendered during the flush, the freed space can then be used to render other objects in the scene allowing the hardware to complete the render. The downside of this is that objects rendered during the flush will not benefit from HSR performed on objects later in the render, which means overdraw may be introduced.

Additionally, the hardware has to flush all buffers associated with the render, which means depth and stencil buffers will have to be flushed in addition to the colour buffer. This needs to be done so that successive renders will execute correctly. Although there is some cost associated with this mode of operation, the technique has been used for years in PowerVR graphics hardware and, as such, has benefitted from much optimization that allows it to have a minimal effect on performance.

The amount of memory allocated for the PB on a given device is done in such a way that the memory footprint is kept as low as possible, but the majority of graphics applications will never encounter this mode of operation (only extremely complex scenes may induce it). On closed platforms using PowerVR graphics hardware, the OEM may choose to expose the ability for an application to choose the amount of memory that is allocated for the PB. By doing so, application developers can opt to allocate more or less memory for the PB than the default for the device, which allows them to raise the threshold before PB management is encountered.

7. Notable features

7.1. Internal True Colour™

Internal True Colour™ is a term that refers to the hardware's ability to perform all blending operations at 32 bits colour precision. This is possible because a 32 bit colour buffer is always used when processing each tile, regardless of the target frame buffer precision. In traditional IMR architectures, image quality is directly related to the frame buffer colour depth; when a 16 bit colour buffer is used in these architectures, multiple reads and writes of the frame buffer will result in a loss of precision, this inaccuracy tends to create "banding" artefacts in the rendered image. While dithering is often used to reduce the effect of banding, it can potentially create worse results again, as the dither pattern applied to multiple blended layers accumulates and strong dither pattern artefacts are introduced (this gives the image a "grainy" appearance. On PowerVR hardware, all blending is performed at 32 bits precision on-chip and each pixel is only written into the frame buffer once, which results in a much higher quality image.

7.2. Full Screen MSAA

Another benefit of the SGX and SGX-MP architecture is the ability to perform efficient 4x Multi-Sample Anti-Aliasing (MSAA). MSAA is performed entirely on-chip, which keeps performance high without introducing a system memory bandwidth overhead (as would be seen when performing anti-aliasing in some other architectures). To achieve this, the tile size is effectively quartered and 4 sample positions are taken for each fragment (e.g. if the tile size is 16x16, an 8x8 tile will be processed when MSAA is enabled). The reduction in tile size ensures the hardware has sufficient memory to process and store colour, depth and stencil data for all of the sample positions. When the ISP operates on each tile, HSR and depth tests are performed for all sample positions. Additionally, the ISP uses a 1 bit flag to indicate if a fragment contains an edge. This flag is used to optimize blending operations later in the render.

When the subsamples are submitted to the TSP, texturing and shading operations are executed on a per-fragment basis, and the resultant colour is set for all visible subsamples. This means that the fragment workload will only slightly increase when MSAA is enabled (there is likely to be some increase in workload as the subsamples within a given fragment may be coloured by different primitives when the fragment contains an edge). When performing blending, the edge flag set by the ISP indicates if the standard blend path needs to be taken, or if the optimized path can be used. If the destination fragment contains an edge, then the blend needs to be performed individually for each visible subsample to give the correct resultant colour (standard blend). If the destination fragment does not contain an edge, then the blend operation is performed once and the colour is set for all visible subsamples (optimized blend).

Once a tile has been rendered, the Pixel Back End (PBE) combines the subsample colours for each fragment into a single colour value that can be written to the frame buffer in system memory. As this combination is done on the hardware before the colour data is sent, the system memory bandwidth required for the tile flush is identical to the amount that would be required when MSAA is not enabled.

7.3. PVRTC Texture Compression

PVRTC is the PowerVR texture compression scheme. PVRTC boasts very high image quality for competitive compression ratios: 4 bits per pixel (PVRTC 4bpp) and 2 bits per pixel (PVRTC 2bpp). These represent savings in memory footprint of 8:1 (PVRTC 4bpp) and 16:1 (PVRTC 2bpp) compared to 32 bit per pixel textures. By using PVRTC compressed textures, an application can significantly reduce its system memory bandwidth overhead without drastically reducing the quality of the texture. More information can be found in the PVRTC & Texture Compression Usage Guide.

8. Related Materials

Software

- PVRTune

Documentation

- OpenGL ES 2.0 Application Development Recommendations
- PVRTC Compression Usage Guide
- PVRTune User Manual

9. Contact Details

For further support contact:

devtech@imgtec.com

PowerVR Developer Technology
Imagination Technologies Ltd.
Home Park Estate
Kings Langley
Herts, WD4 8LZ
United Kingdom

Tel: +44 (0) 1923 260511
Fax: +44 (0) 1923 277463

Alternatively, you can use the PowerVR Insider forums:

www.imgtec.com/forum

For more information about PowerVR or Imagination Technologies Ltd. visit our web pages at:

www.imgtec.com

Imagination Technologies, the Imagination Technologies logo, AMA, Codescape, Ensigma, IMGworks, I2P, PowerVR, PURE, PURE Digital, MeOS, Meta, MBX, MTX, PDP, SGX, UCC, USSE, VXD and VXE are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.