GPU User's Guide

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

List of Figures

List of Tables

1 About This Document

This document provides an overview of the PlayStation®Vita GPU – SGX543MP4+, and also provides detailed descriptions of the GPU data formats for primitives, textures, and surfaces.

- Chapters 2-6 present a high-level overview of GPU processing and data flow, including descriptions of each hardware block's primary functions. The purpose of these chapters is to explain the GPU from a hardware architecture perspective, in particular, with a view to aid GPU performance analysis.
- Chapters 7-12 focus on the types, data structures and memory layouts for primitives, vertex formats, texture formats color surface formats, and depth and stencil surface formats.
- Chapters 13-14 provide performance data and basic checkpoints for efficient GPU programming.
- Appendix A contains summary tables of the GPU data formats.
- Appendix B explains the PVRTC compressed texture formats in more detail.

Related Materials

Please refer to the following related materials for a detailed reference:

- *libgxm Overview* – provides an overview of the libgxm graphics library and how the library calls interact with the underlying GPU.
- *libgxm Reference* – provides detailed reference descriptions of each of the libgxm API functions and data structures.
- *Razor User's Guide* – provides detailed descriptions of Razor for PlayStation®Vita GPU performance analysis and debugging features.

2 Introduction

The PlayStation®Vita SoC (system-on-a-chip) contains an SGX543MP4+ GPU. This is a multi-core, tile based deferred rendering GPU, with an advanced unified shader architecture. The key features of the SGX543MP4+ are summarized below.

Key Features

Features	Description
Multi-core	• Four SGX543+ cores • Automatic load-balancing of vertex and fragment processing work
Tile Based Deferred Rendering	• Hidden surface removal – per fragment, before shading - Avoids shading occluded pixels • On-chip tile depth buffer (floating point) • On-chip tile stencil buffer (8-bit) • On-chip tile color buffer (multi-pixel format, max 64 bits) • Significant reduction in color/depth/stencil buffer read + write memory bandwidth
Unified Shader Architecture	• Sixteen USSE – *Universal Scalable Shader Engine* – pipes - Four USSE pipes per SGX543+ core • Advanced feature set - exceeding Shader Model 3.x • Fine-grain instruction level thread switching – zero cost • SIMD instruction set (32-bit / 16-bit floating-point, integer) • Dynamic flow control, zero-cost instruction predication • Arbitrary memory loads and stores
Texture	• 2D (including non-power-of-two), cube map, projective • Formats include: - Integer: 8-bit, 16-bit, 32-bit, packed - Floating-point: 16-bit, 32-bit - Compressed: UBC1-5, PVRTC-I & II - Palette: CLUT4, CLUT8 • Bilinear, trilinear and centroid sampling • sRGB to linear gamma correction • 4096 x 4096 maximum size
Render Targets	• Anti-aliasing, MSAA x4 and x2, programmable sample positions • On-chip MSAA resolve – converts samples to pixels • Formats include: - Integer: 8-bit, 16-bit, 32-bit, packed - Floating-point: 16-bit, 32-bit - Pseudo HDR: F11F11F10, U2F10F10F10, SE5M9M9M9 • Linear to sRGB gamma correction • 4096 x 4096 maximum size

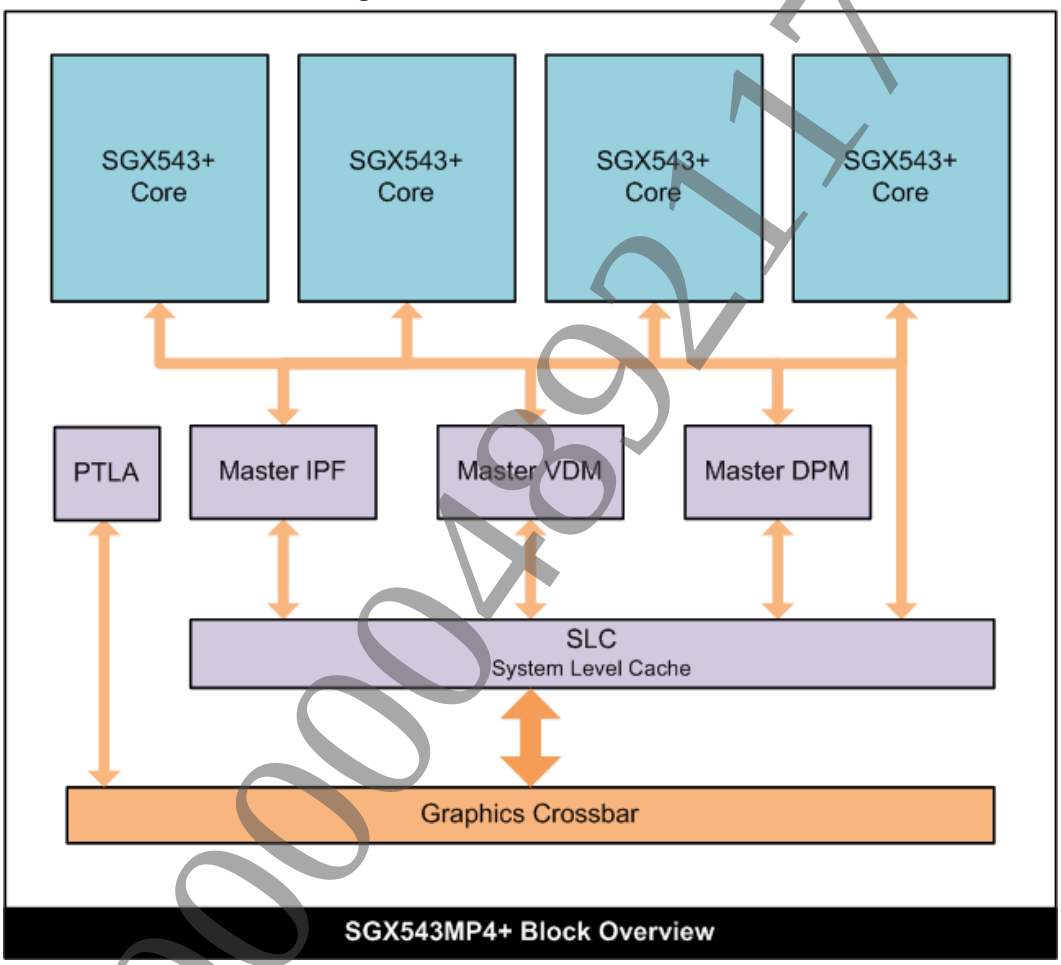
3 GPU Hardware Block Overview

This chapter provides a high-level overview of the SGX543MP4+ in terms of its underlying hardware blocks.

SGX543MP4+ Block Overview

The SGX543MP4+ multi-core GPU contains four SGX543+ cores and several *master* blocks whose role it is to orchestrate and distribute work amongst the cores efficiently. These blocks and their connectivity are illustrated in Figure 1.

Figure 1 GPU Block Overview



The master blocks of SGX543MP4+ are described below.

Master VDM

The Master VDM – *Master Vertex Data Master* – is the front end of the GPU. It is responsible for reading the single command stream constructed by the graphics driver (libgxm) from memory; and for distributing all vertex work amongst the cores.

The Master VDM splits vertex processing work amongst the cores in order to load balance their usage, while at the same time preserving primitive submission order to ensure correct rasterization. Vertex processing work is basically split according to a pre-defined split threshold (in number of primitives). These primitive chunks are the fundamental unit of vertex processing work at master level. Consequently, it is possible for individual *Draw* commands to be split over multiple cores. Once the cores accept their

requested vertex processing work they operate independently, generating their own subset of the Parameter Buffer.

Master IPF

The Master IPF – *Master ISP Parameter Fetch* – is responsible for processing the Parameter Buffer produced by the multiple cores and initiating the rasterization process. As Master IPF reads the Parameter Buffer, it combines the individual subsets produced by the cores, such that each tile has a single tile command stream. A tile command stream contains references to primitive data (output by vertex programs) and state data required for rasterization and fragment processing.

Using information provided by the Master VDM, the Master IPF also ensures primitives binned in each tile are rasterized in submission order. In parallel to the Parameter Buffer read, the Master IPF will also distribute rasterization and fragment processing amongst the cores; and ensure they are efficiently load balanced. Since each tile is completely independent, distribution across the cores is naturally achieved in terms of tiles; this is the fundamental unit of rasterization work at master level. Each tile assigned to a core is rasterized and processed entirely on that core.

Master DPM

The Master DPM – *Master Dynamic Parameter Management* – (block) is responsible for managing Parameter Buffer memory for all GPU cores. No CPU or user intervention is required to manage the Parameter Buffer; it is handled entirely in hardware.

Parameter Buffer memory is managed as a linked list of pages. As vertex processing proceeds individual cores request memory pages from the Master DPM, which are allocated from a free list. As fragment processing proceeds and sections of the scene are rendered, the Master DPM receives lists of memory pages back from the cores. These page allocation and free operations are performed in parallel to help ensure the maximum amount of free Parameter Buffer memory is maintained.

PTLA

The PTLA - *Present and Texture Load Accelerator* - (block) is a fixed function transfer unit that operates asynchronously to the GPU cores. The PTLA block supports various forms of format conversion, memory layout conversion, downscaling, copying, and filling of 2D images. See “Transfers using PTLA“ for additional information.

SLC

The SLC – *System Level Cache* – is the highest level cache within the GPU; sitting between all cores, the master-level blocks and memory. All GPU memory requests go through the SLC in order to avoid re-fetching of memory *between* and *within* the cores.

In addition, to avoid thrashing the SLC, not all memory requests are cached. This depends on the type of data and the hardware block issuing the request. For example, texture and vertex attribute data is likely to be read many times and so is cached; whereas the VDM Command Stream is read only once and so is un-cached. Similarly, triangle primitive data in the Parameter Buffer is written only once per *macro tile* (group of tiles) and so these writes are un-cached, however, this same data is read many times and so reads are cached.

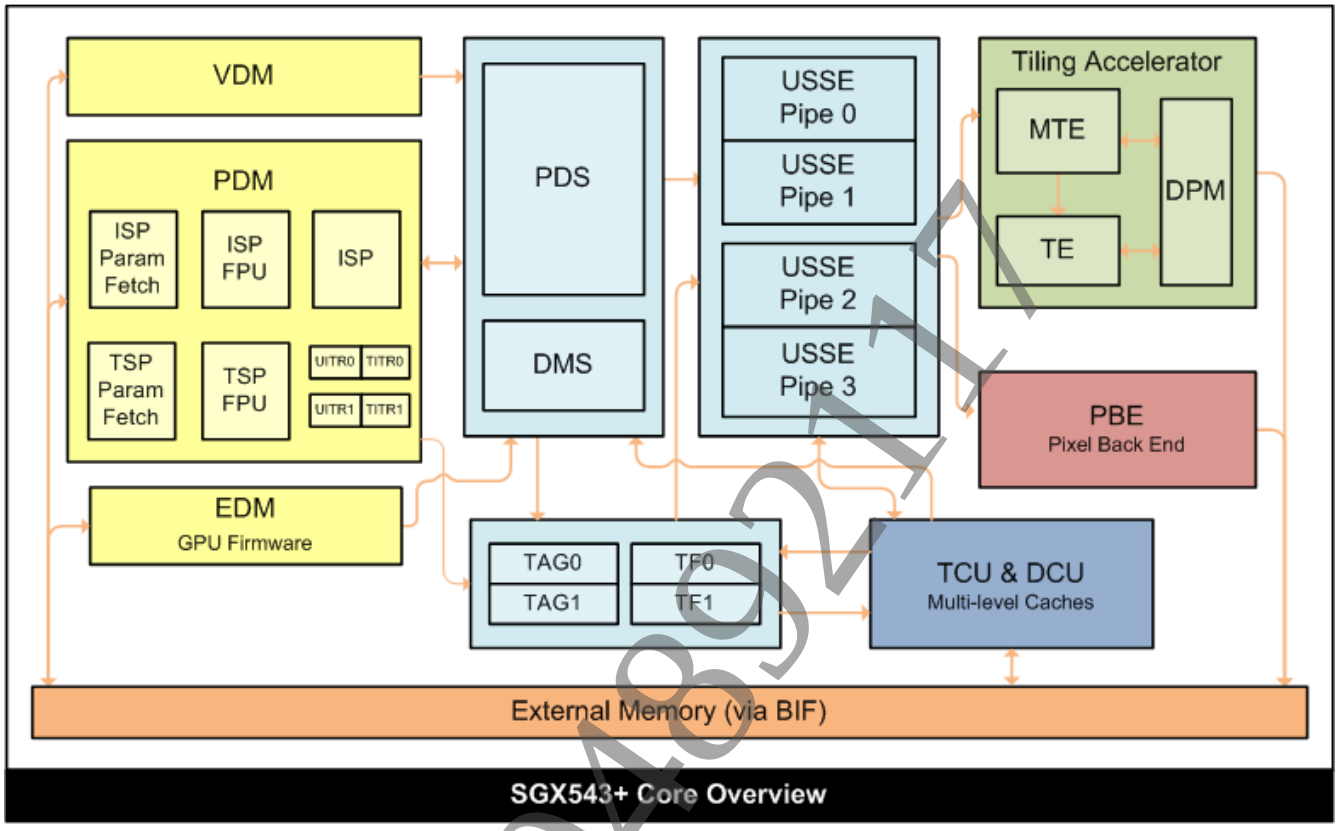
The SLC is implemented in four separate cache banks, one per memory access channel, with a crossbar directing the requests based on address. This allows all four cores to access different SLC banks in parallel.

- Features
  - 256K total cache size
    64K cache bank - per core
  - 16-way set associative
  - Pseudo-LRU replacement policy

Single SGX543+ Core Block Overview

This section provides a high-level overview of a single SGX543+ core in terms of its underlying hardware blocks. There are four such cores within the SGX543MP4+ multi-core GPU, each of them identical. The hardware blocks within a core and their connectivity are shown in Figure 2.

Figure 2 SGX543+ Core Overview



The hardware blocks of a single SGX543+ core are described below.

VDM

The VDM – *Vertex Data Master* – is responsible for starting vertex processing within the core. The VDM accepts VDM command stream segments from the Master VDM via a FIFO. The VDM command stream contains high-level commands such as *Draw Index List* and *Set Vertex Processing State*, which in turn reference primitive (vertex) indices and state data. The state data is used to setup the PDS, USSE pipes and Tiling Accelerator for vertex processing.

Additionally, the VDM parses the index data to determine unique indices; forwarding only these indices to the PDS. This is done to help reduce redundant vertex shading computation.

PDM

The PDM – *Pixel Data Master* – is responsible for starting rasterization and fragment processing within the core. The PDM accepts tile command streams, stored in the Parameter Buffer, from the Master IPF. One tile command stream is received per tile, containing references to primitive data (output by vertex programs) and state data. The tile command streams contain all the information needed to setup and execute all subsequent fragment processing stages.

The PDM comprises the hidden surface removal unit (also known as the Image Synthesis Processor or ISP) and the texture and shader setup unit (known as TSP).

PDS

The PDS – *Programmable Data Sequencer* – controls how vertices and fragments are processed on the USSE pipes; including the fetching of input data and allocation of USSE shared resources.

During vertex processing the VDM issues commands to the PDS, which is responsible for de-indexing each vertex and DMA-fetching its input vertex attribute data. The PDS then issues a command to the USSE pipes so that the associated vertex program is executed.

During fragment processing, the PDS receives groups of 2x2 pixel blocks, known as *spans,* following hidden surface removal. All fragments in a span are visible and use the same fragment program.

The PDS then issues a series of commands to fetch interpolants, issue non-dependent texture reads and instruct the USSE pipes to execute the associated fragment program for visible fragments.

DMS

The DMS - *Data Master Selector* - arbitrates between the data masters (VDM and PDM) so that vertex and fragment processing can share the single PDS unit on a core.

USSE

The USSE – *Universal Scalable Shader Engine* – is a fully programmable processing unit, primarily used to execute vertex and fragment programs. There are four USSE pipes within a single SGX543+ core, each accepting work (in the form of *tasks*) from the PDS.

The USSE units have a powerful and highly optimized instruction set for processing vertices and fragments efficiently; the instruction set is also general in nature. Consequently the USSE units are also responsible for executing the GPU Firmware (on core 0 only).

The USSE also has instructions for interfacing other hardware blocks, including instructions to emit shaded vertices to the Tiling Accelerator or entire rendered tiles (on *end of tile*) to the Pixel Back End (PBE).

Each USSE pipe has its own control unit (for scheduling tasks), pipeline controller (for decoding instructions) and pipeline data path (for executing instructions). Additionally, each USSE pipe has its own local memory, known as *unified store*, for holding input, temporary and output registers.

Each USSE pipe is responsible for scheduling its own work; and it does so at fine-level to ensure maximum usage of its resources and improve performance. It is therefore possible for a single USSE pipe to execute multiple vertex and fragment programs in parallel.

Tiling Accelerator

The TA - *Tiling Accelerator* – is the collective name for the group of hardware blocks responsible for implementing the *tiling process*; that is, the binning of primitives following vertex processing into tiles, ready for rasterization and deferred rendering. The Tiling Accelerator comprises of the MTE, TE and DPM blocks described below.

MTE

The MTE – *Macro Tiling Engine* – performs the first stage of the tiling process. It accepts shaded vertices from the USSE pipes and primitive index data from the VDM (via the IDX FIFO) and subsequently generates blocks of vertex and index data, known as *primitive blocks*, binned by *macro tiles* (a high-level rectangular group of tiles).

During this process, after the viewport transform has been applied, the MTE will cull as many primitives as possible, in order to reduce the amount of vertex and index data written to memory; and subsequently reduce the number of primitives that need to be rasterized and fragment shaded. The MTE's culling methods include: back-face culling, off-screen culling and small-primitive culling.

Primitives culled by the MTE are rejected from all further processing. The remaining accepted primitives then undergo clipping (if necessary), before being forwarded to the TE.

TE

The TE – *Tiling Engine* – performs the second stage of the tiling process. It accepts primitive data from the MTE and performs two incremental tiling algorithms in order to compute the minimal list of tiles that intersects each primitive. The TE then uses this information to create lists of primitives that are contained within each tile. During the tiling process, these lists are written to memory, one per tile, in the form of tile command streams.

Once the MTE indicates the completion of the scene, the TE will then terminate the tile control streams and write their list headers (known as *region headers*) to memory. Together, the primitive blocks of the MTE, and region headers and tile command streams of the TE make up the Parameter Buffer.

DPM

The DPM – *Dynamic Parameter Management* – (block) manages Parameter Buffer memory page allocations and de-allocations. It works closely with the Master DPM so a single Parameter Buffer can be managed entirely in hardware; and without CPU or user intervention.

During the tiling process the DPM will receive page allocation requests from the MTE and TE.

During fragment processing the DPM will receive page de-allocation requests from the Master DPM once each macro tile has been rendered and its associated list of memory pages can be freed.

The DPM, together with the Master DPM, is also responsible for detecting when the Parameter Buffer memory heap has been exhausted; and subsequently signaling when a Partial Render needs to be executed.

ISP

The ISP – *Image Synthesis Processor* – is the first stage in the fragment processing pipeline and is responsible for performing pixel(and sample) accurate hidden surface removal. The ISP does this on a tile basis ahead of fragment shading; a key feature of the SGX architecture known as tile based deferred rendering. This ensures only visible fragments are shaded and USSE cycles are not wasted on occluded fragments, which are discarded at the start of the fragment processing pipeline.

The ISP consists of the following three sub-blocks, which are described in processing stage order:

- **ISP Parameter Fetch** block parses tile command streams received from the Master IPF, forwarding ISP state information downstream and fetching primitive vertex positional (XYZ) data from Parameter Buffer primitive blocks in memory.
- **ISP FPU** block then converts all incoming primitives into triangles (including lines and point sprites), before generating the necessary plane equations required for rasterization and depth comparison.
- **ISP** block will conduct *hidden surface removal* for the tile, using dedicated on-chip tile depth / stencil /mask memory. This greatly reduces the memory bandwidth required for depth/stencil tests; even reducing it to zero if the depth/stencil values are no longer needed in future passes or the current scene. The ISP block is additionally responsible for initializing the tile (loading depth/stencil /mask values from memory if required), multi-sampling, visibility tests and updating the depth /stencil /mask buffer in memory (if required).

TSP

The TSP – *Texture and Shader setup Processor* – accepts groups of visible 2x2 pixel blocks (known as *spans*) from the ISP and performs the necessary setup for texturing and fragment shading to proceed.

The TSP consists of the following two sub-blocks, described in processing stage order:

- **TSP Parameter Fetch** block first fetches vertex position, color, and texcoord attributes from memory for visible primitives; these are the vertex program outputs stored in primitive blocks during tiling. Once read from memory, this vertex data is stored in a dedicated cache to reduce memory bandwidth (that is, when vertices are shared amongst primitives). Additionally the TSP Parameter Fetch block fetches state information for visible primitives, which is forwarded to the PDS to setup fragment processing of spans.
- **TSP FPU** block then uses the vertex data fetched by the TSP Parameter Fetch block to perform *triangle setup.* Like the ISP FPU, it first converts all primitives (including lines and point sprites) into triangles and then generates plane equations (per required vertex attribute), which are forwarded to the iterator (TITR and UITR) blocks.

UITR

The UITR – *USSE Iterator* – (blocks) generate per-fragment color and texcoord data for visible primitives. These values are computed at 32-bit precision using fragment X, Y positions from the PDS, with the vertex attributes fetched by the TSP Parameter Fetch block and the plane equations generated by the TSP FPU.

These per-fragment colors and texcoords (and position, if required) are then forwarded to the USSE pipes as fragment program inputs.

There are two UITR blocks within a core; each shared by two USSE pipes.

TITR

The TITR – *TAG Iterator* – (blocks) generate per-fragment texcoord data for visible primitives. These values are computed first at 32-bit precision using fragment X, Y positions from the PDS, with vertex texture coordinate attributes fetched by the TSP Parameter Fetch block and the plane equations generated by the TSP FPU, but are subsequently quantized to 24-bit precision before perspective correction takes place.

These per fragment texcoords are then forwarded to the TAG blocks to allow texture data to be pre-fetched.

There are two TITR blocks within a core; each shared by two USSE pipes.

TAG

The TAG – *Texture Address Generator* – (blocks) receive texture requests, in the form of texture coordinates, state, and LOD information; these inputs are then used to generate one or more addresses for texture look-ups and coefficients for filtering.

The TAG blocks receive texture requests from two sources:

- From the iterator TITR blocks, when texture coordinates are not modified by the fragment program and so the texture data can be pre-fetched early; (known as *non-dependent texture reads*).
- From the USSE pipes, when texture coordinates are modified by the fragment program and so the texture data cannot be pre-fetched early; (known as *dependent texture reads*).

Vertex texture requests always originate from the USSE pipes and so are always dependent texture reads.

The texture addresses generated by the TAG blocks are forwarded to the Texture Cache Unit (TCU) so the texel data can be read.

There are two TAG blocks within a core; each shared by two USSE pipes.

TF

The TF – *Texture Filter* – (blocks) are responsible for texture filtering. They receive read texel data from the Texture Cache Unit (TCU); and texture state and filter coefficients from TAG. These inputs are then used to format expand and filter the texture lookup. sRGB to linear gamma correction is also applied if required.

The results are finally forwarded to the USSE pipes for use in fragment (and vertex) programs; they are stored in input or temporary registers.

There are two TF blocks within a core; each shared by two USSE pipes.

PBE

The PBE – *Pixel Back End* – (block) is responsible for the final stage of fragment processing; it receives completely rendered tiles from the USSE pipes, applying numerous conversion operations to fragments before writing the final tile pixel data to memory.

These conversion operations take place in the PBE on-chip color buffer.

PBE conversion operations include: linear to sRGB gamma correction, downscaling for x2 and x4 MSAA resolve and pixel packing format conversion (including dithering).

The PBE is also responsible for address translation, such that color surfaces can be written to memory in linear, tiled, or swizzled layouts.

DCU

The DCU – *Data Cache Unit* – is a multi-level data cache providing DMA translation and memory caching for the four USSE pipes and PDS.

The PDS can instruct the DCU to fetch data from memory and write it directly back to the USSE pipes.

The USSE pipes may also directly instruct the DCU to read memory, as well as write data to the DCU and/or memory.

The levels of the DCU are as follows:

- L0 - DMA
  - Provides DMA translation; converting burst memory requests into cache lines requests and burst write back to the USSE pipes
  - One L0 DMA is shared per two USSE pipes + the PDS
  - There are two L0 DMA units within the DCU
- L1 - Data cache
  - 512B size (16 lines x 256-bit)
  - Pseudo-LRU replacement policy
  - One L1 data cache per L0 DMA
  - There are two L1 data caches within the DCU
- L2 – Data cache
  - 1K size (32 lines x 256-bit)
  - Pseudo-LRU replacement policy
  - Serves all four USSE pipes + the PDS
  - There is one DCU L2 data cache within a core

Any memory not cached by the DCU is fetched from external memory via the BIF.

TCU

The TCU – *Texture Cache Unit* – is a dedicated data cache, responsible for reducing the memory bandwidth and latency of texture lookup operations.

The TCU has two levels:

- L1 – Data cache
  - 512B size (16 lines x 256-bit)
  - 4-way set associative, pseudo-LRU replacement policy

- One L1 data cache per two USSE pipes + one TAG unit
- There are two L1 data caches within the TCU

- L2 – Data cache
  - 8K size (256 lines x 256-bit)
  - 4-way set associative, pseudo-LRU replacement policy
  - Serves all four USSE pipes + the two TAG units
  - There is one TCU L2 data cache within a core

Any memory not cached by the TCU is fetched from external memory via the BIF.

BIF

The BIF – *Bus Interface* – is the interface through which the core accesses external memory in the PlayStation®Vita SoC, specifically, main memory and video memory.

4 USSE - Universal Scalable Shader Engine

This chapter provides an overview of the USSE – *Universal Scalable Shader Engine* – architecture; with particular emphasis on data flow, interfaces and how work is partitioned and scheduled for execution.

There are four USSE pipes within a single SGX543+ core; sixteen USSE pipes within the SGX543MP4+ in total. Unless specifically stated, the details that follow are explained in the context of a single USSE pipe. All USSE pipes within the SGX543MP4+ are identical.

The USSE is a fully programmable processing unit, primarily used to execute vertex and fragment programs. The USSE has a highly optimized instruction set for processing vertices and fragments efficiently. Vertex operations executed on the USSE include: vertex attribute unpacking, transform and lighting, deformations and skinning. Fragment operations include: lighting, texturing, blending, alpha-test and component masking. Since all of these operations execute on the USSE, they are fully programmable; even those operations that are traditionally handled by fixed-function units.

Although highly optimized, the USSE instruction set is also general in nature. Consequently USSE pipes (on core 0 only) are also responsible for executing the GPU Firmware.

USSE Data Flow

The USSE is able to process work by having its input *attribute registers* setup. This is typically done by external units, such as the PDS ,when it DMA transfers vertex attribute data or forwards 2x2 pixel blocks; or the texture filtering unit TF, when forwarding the results of non-dependent texture reads (see "TAG" unit description for further details on texture reads).

There are two types input attributes:

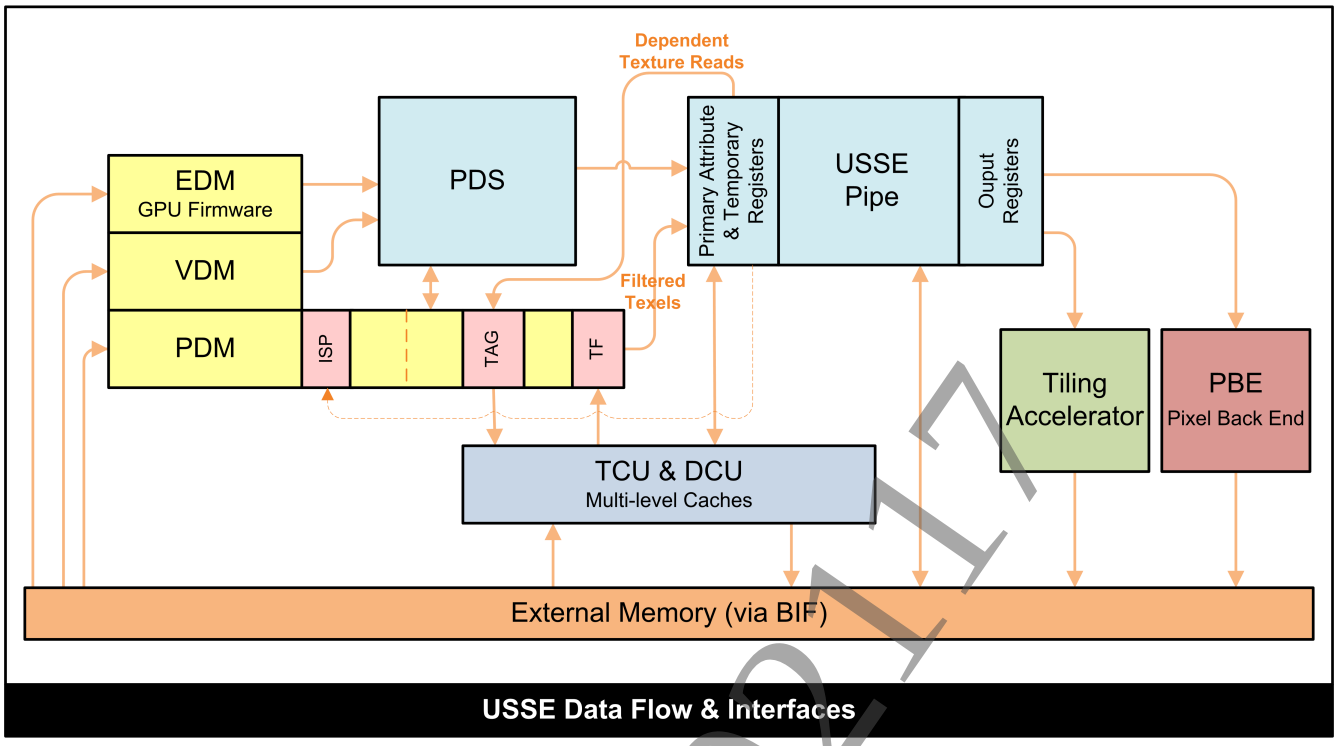
- *Primary attributes* – store each *data instance* that the USSE will process. In the case of vertex processing a data instance is a single vertex; as input into a vertex program. In the case of fragment processing a data instance is a single fragment; as input into a fragment program. Primary attributes are set per data instance. Furthermore data instances are isolated from one another; that is to say, one data instance cannot access the primary attributes of another.
- *Secondary attributes* – store additional input data that is required for processing and remains constant over a number of data instances. Secondary attributes are used to store shader program uniforms, such as, transformation matrix coefficients, lighting constants and so on. Secondary attributes are only set whenever they need updating and are shared across all data instances of a shader program. Each data master (VDM or PDM) has its own secondary attributes.

Once the inputs have been written into the attribute registers, the USSE can then start executing the shader program instructions.

Once the USSE has executed all instructions in the shader program, it can then signal to external units that its results have been written to the output registers and are ready to be read (by the following unit); for example, shaded vertices for the Tiling Accelerator or entire rendered tiles (on end of tile) for the Pixel Back End (PBE).

A more detailed view of USSE data flow and its interfaces with external blocks are illustrated in Figure 3.

Figure 3 USSE Data Flow



USSE Unified Store

The USSE attribute, temporary and output registers are stored in USSE local memory, known as, *unified store*.

The division / partitioning of unified store in terms of primary attribute, secondary attribute and temporary registers is dynamically controlled by the graphics driver (libgxm).

USSE Tasks

The Programmable Data Sequencer (PDS) is responsible for controlling and submitting work to the USSE, it does this in the form of *tasks*. A task comprises of between one and sixteen data instances; and each data instance flows once through the USSE shader program for it to be processed.

For example, during vertex processing, each data instance in a task is a vertex; similarly for fragment data instances and fragment processing tasks.

All data instances in a task use the same USSE shader program.

Whether a USSE shader program can be executed (and so by implication - a task) may depend on an event (signal) from an external unit; such tasks are said to have *external dependencies*.

Additionally, tasks can have *sequential dependencies*; both implicitly (as defined by the hardware) or via software control. Sequential dependencies ensure a task does not start until all previous tasks belonging to the same data master are complete.

A good example of sequentially dependent tasks are fragment processing tasks, which have an implicit sequential dependency ensuring fragment tasks with the same (x, y) positions cannot be executed out of order; this ensures fragments are processed in submission order for correct rendering.

The PDS is responsible for creating USSE tasks, allocating and filling primary and secondary attribute registers; allocating temporary and output registers; and providing additional task control information.

USSE Threads

The USSE executes shader programs on data instances in a multi-threaded fashion. This is achieved using one of two threading modes, which is selected by the graphics driver (libgxm) based on the nature of the compiled shader program. The two threading modes are:

Per-instance mode – each thread contains a single data instance; the thread executes using single instruction, single data (SISD) execution. This mode is specified for shader programs with dynamic flow control.

Parallel mode – each thread can contain up to four data instances; the thread executes using single instruction, multiple data (SIMD) execution. This (more efficient) execution mode is specified for shader programs without dynamic flow control; however, such programs can contain instruction predication and static control flow.

A thread always has a single program counter (PC); this is true even if the thread executes in parallel mode and there are four data instances.

USSE Tasks, Threads and Execution Model

The USSE has a sixteen-deep *task queue*, which allows the PDS to work in advance of the USSE in order to help decouple the two units.

The USSE also has a collection of sixteen threads, which it manages at one time using its *thread manager*. The thread manager has sixteen *slots*; each slot holding one thread.

Together the USSE task queue and thread manager impose three stages through which tasks must flow in order to be executed on the USSE.

When the thread manager contains empty *slots,* the USSE searches the task queue for a task or *sub-task* it can schedule.

A *sub-task* is a task which has already had some of its data instances scheduled. They can arise because there is not a one to one mapping between tasks and threads; since a task can contain up to sixteen data instances and a thread can only contain up to four.

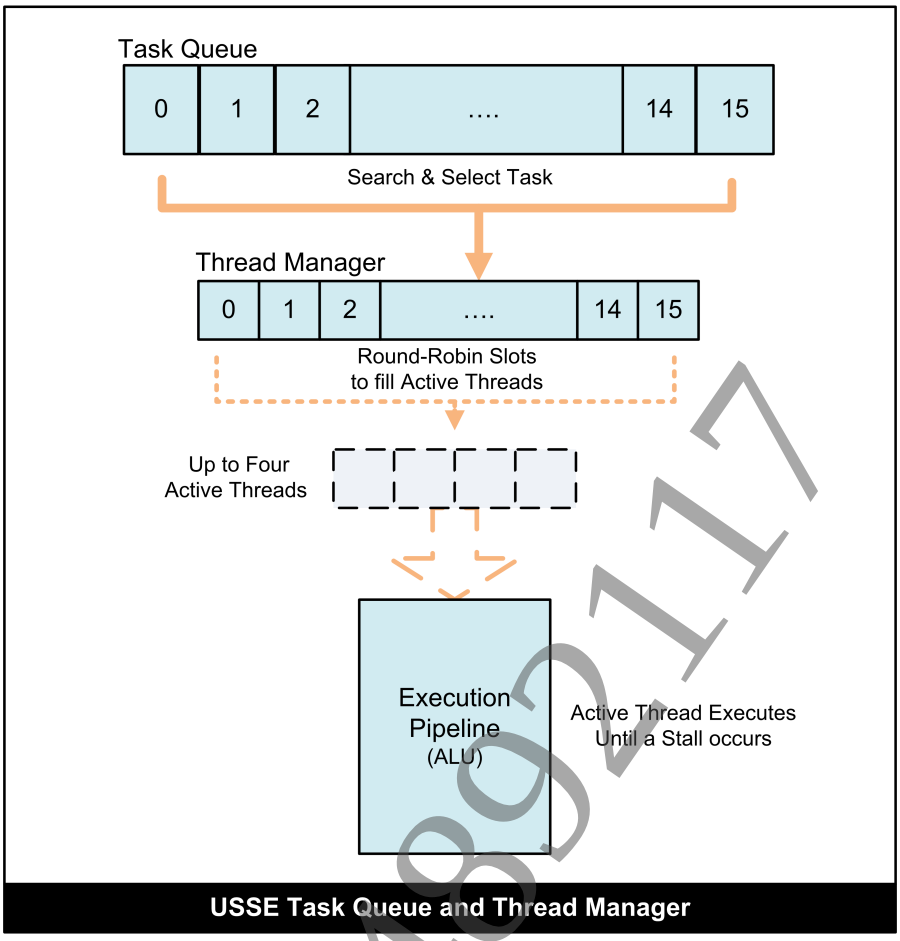
Tasks are selected for execution based on:

- *External dependencies* – if the task has any, have they been satisfied?
- *Sequential dependencies* – if the task has any, have they been satisfied?
- *Resources* – is there sufficient output register space for all scheduled tasks?

Once the USSE has selected a task, it can then schedule its data instances into the empty thread slots; up to four data instances per thread for parallel mode; one data instance per thread for per-instance mode. Data instances flow into the empty threads until the thread manager slots are full. If data instances belonging to the selected task remain, a sub-task is formed. The sub-task will remain in the task queue until thread manager slots become empty again. When this occurs the USSE will conduct a new search through the task queue, again looking for tasks or sub-task that can be selected for scheduling according to the conditions listed above.

This process is repeated continuously, as long as there is work in the task queue; as shown in Figure 4.

Figure 4 USSE Tasks Queue and Thread Manager



Since data instances are independent and do not share data amongst themselves; they can be executed out of (PDS) submission order; specifically, the order of task execution is undefined without the presence of sequential dependencies.

A thread in the thread manager can be:

- *Empty* – slot empty.
- *Active* - currently executing on the USSE.
- *Ready* - can be made active because all scheduling conditions have been met.
- *Waiting* - for hazard to clear.

On every clock cycle the USSE switches between the active threads on a round robin basis.

Of the sixteen threads only up to four are active at one time; this matches the latency of the USSE execution pipeline and so helps ensure internal pipeline data hazards are avoided.

Additionally, recall that in parallel mode, each thread contains up to four data instances. Consequently in parallel mode, four active threads, each containing up to four data instances implies a maximum of sixteen data instances processed at one time. This increased parallelism further improves the ability of the USSE to hide instruction latency.

Generally once threads are made active, they remain active until the program finishes or:

- The instruction cache misses trying to access an instruction.
- The instruction waits for dependent texture read data to return.
- The instruction waits for a branch condition.
- A timeout limit on the thread has been reached.

In other words, threads are deactivated when four cycles of instruction latency absorption is not enough to eliminate (that is, hide) the data hazard. In this case the thread is deactivated (made *waiting*) and a *ready* thread takes its place, so that maximum ALU performance can be achieved.

This fine grain instruction level thread switching has zero cost on the USSE.

Once a thread is deactivated, the USSE automatically detects when the data hazard no longer exists and so can be re-activated; the thread state changes from *waiting* to *ready*.

Due to the USSE scheduling method described, it is possible for a single USSE to switch between different vertex and fragment programs on each cycle; specifically, a maximum of four different shader programs in total; one per active thread.

A key goal of the USSE architecture, in terms of performance, is to keep its arithmetic logic units (ALU) as busy as possible. The USSE does this by using the 16-deep task queue to help keep the sixteen slot thread manager full. The sixteen threads in the thread manager are then used to try to keep up to four threads active at all times. If a thread needs to be deactivated, one of the *ready* threads immediately takes its place. By having three stages; from tasks to threads to active threads, the probability of keeping the USSE execution pipelines busy is dramatically increased.

USSE Tasks, Threads and Data Instance Counts

The number of tasks, threads and data instances within the USSE; and how they scale across a single SGX543+ core and the entire SGX543MP4+ GPU is summarized below:

USSE pipe	• 1 unified store • 1 task manager - 16 deep task queue - Up to 16 data instances - per task • 1 thread manager - 16 thread slots - Up to 4 data instances – per thread (parallel mode) - Up to 4 active threads, 16 data instances
SGX543+ Core – containing four USSE pipes	• 4 unified stores • 4 task managers - 4 x 16 deep task queues (64 entries total) • 4 thread managers - 4 x 16 thread slots (64 slots total) - Up to 16 active threads, 64 data instances
SX543MP4+ GPU – containing four SGX543+ cores	• 16 unified stores • 16 task managers - 16 x 16 deep task queues (256 entries total) • 16 thread managers - 16 x 16 thread slots (256 slots total) - Up to 64 active threads, 256 data instances

5 Vertex Processing

This chapter provides a high-level overview of vertex processing. Explaining how such work is submitted to the GPU and how the data required for vertex processing flows through the main vertex processing stages.

VDM Command Stream

The VDM command stream is the front-end command stream of the SGX543MP4+ GPU, through which vertex processing commands are submitted by the graphics driver (libgxm). The GPU front-end and its VDM command stream is very thin, containing only four high-level commands; they are:

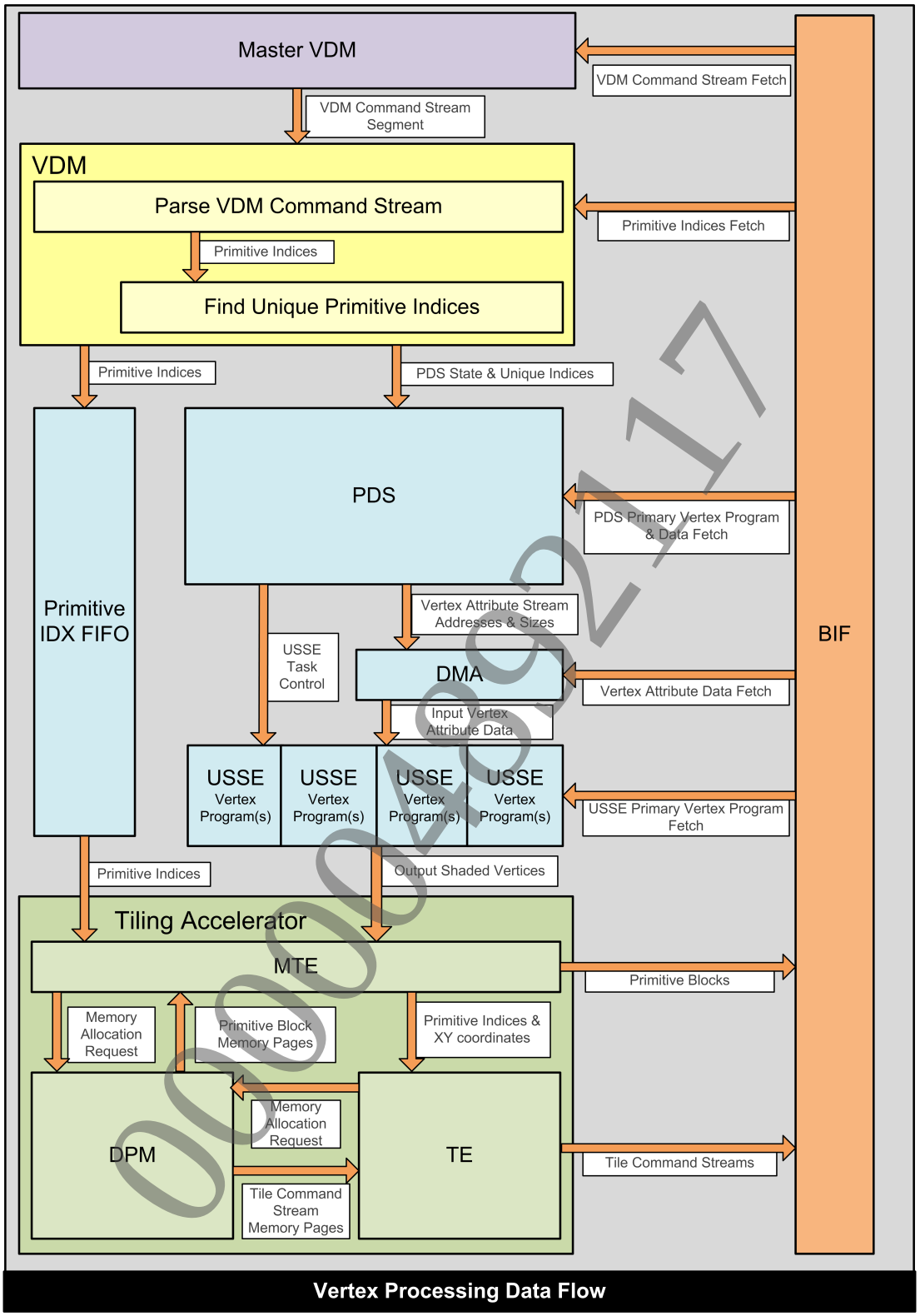
- *Set Vertex Processing State* - a command used for setting GPU state; including information on how the state settings should be fetched from memory (by the PDS) and processed (by the USSE pipes). This one (general) command is used to initiate the setting of all GPU state; including vertex program secondary attributes (such as shader uniforms) and Tiling Accelerator settings responsible for configuring the tiling process. The latter includes fragment processing state, which is propagated into the Parameter Buffer during the tiling process and in order to configure the following fragment processing stage; including hidden surface removal (on the ISP) and fragment shading (on the PDS and USSE pipes).
- *Draw Index List* – a command used to initiate a drawing of an index list; including information on the type of primitives to be processed (e.g. triangles, lines, points, strips, fans etc.) and how vertices should be fetched from memory (by the PDS) and processed (by the USSE pipes).
- *Stream Lin*k – a command which allows non-contiguous blocks of the VDM command stream to be linked together in memory (like a next pointer in a linked list).
- *Terminate* – a command used to signal the end of the VDM command stream and initiate the end of a vertex processing job.

The simplicity of the VDM command stream, in terms of the number of commands, is a consequence of a fundamental SGX543MP4+ design goal; that the setting of all state and vertex and fragment shading (even traditional fixed-function stages) should be fully programmable; from the fetching of state / input data from memory (by the PDS), to the processing of state, vertex and fragment data (by the USSE pipes).

Vertex Processing Data Flow

The main vertex processing stages in terms of hardware blocks and data flow are illustrated in Figure 5.

Figure 5 Vertex Processing Data Flow



Vertex Processing Data Flow

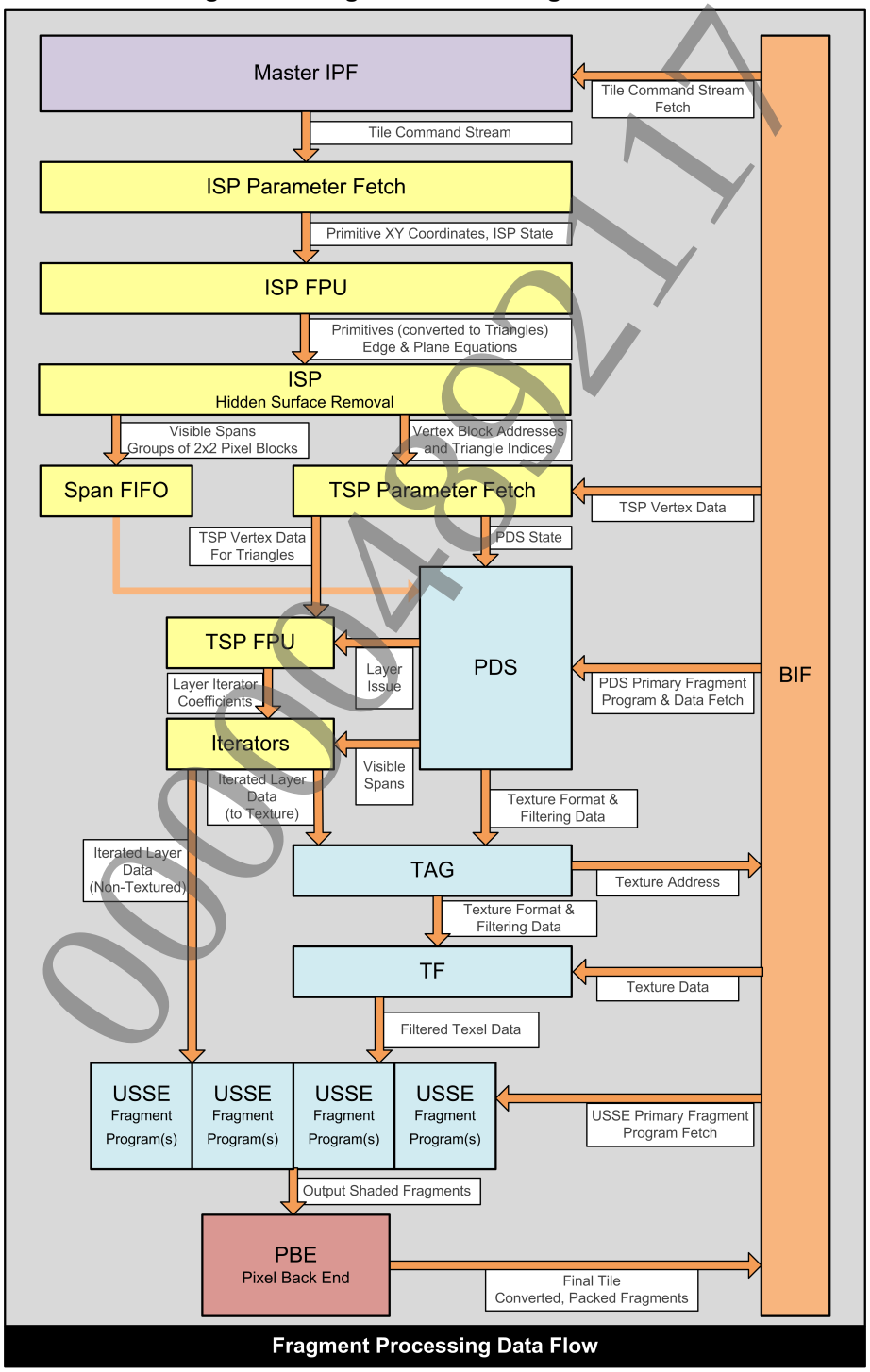
6 Fragment Processing

This chapter provides a high-level overview of fragment processing. Explaining how such work is initiated by the GPU and how data flows through the main fragment processing stages.

Fragment Processing Data Flow

The main fragment processing stages in terms of hardware blocks and data flow are illustrated in Figure 6.

Figure 6 Fragment Processing Data Flow



7 Primitives

The GPU supports the following primitive types:

Table 1 List of Supported Primitive Types

Primitive Type	Description
SCE_GXM_PRIMITIVE_POINTS	Indexed Point List
SCE_GXM_PRIMITIVE_LINES	Indexed Line List
SCE_GXM_PRIMITIVE_TRIANGLES	Indexed Triangle List
SCE_GXM_PRIMITIVE_TRIANGLE_STRIP	Indexed Triangle Strip
SCE_GXM_PRIMITIVE_TRIANGLE_FAN	Indexed Triangle Fan
SCE_GXM_PRIMITIVE_TRIANGLE_EDGES	Indexed Triangle Edge List

All primitives are indexed, the GPU does not support non-indexed primitives. Each index value is used to retrieve attributes from a vertex stream.

The index count for each draw call is stored as a 22-bit unsigned integer, so the maximum number of indices per draw call is $2^{22} - 1$.

Indices can be stored in memory as either 16-bit or 32-bit unsigned integers. When 32-bit integers are used, only the low 24 bits are valid.

Point List

An indexed point (SCE_GXM_PRIMITIVE_POINTS) list contains a single index for each point.

The index values that describe a list of points would be arranged in the index list as shown below.

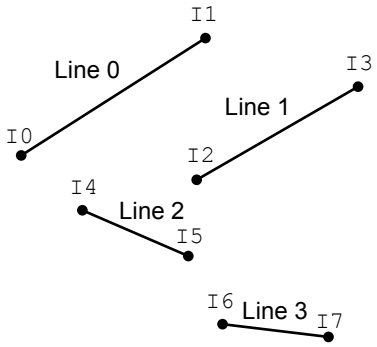
16-bit or 32-bit Value
Index 0 (Point 0)
Index 1 (Point 1)
Index 2 (Point 2)
Index 3 (Point 3)
Index 4 (Point 4)
…
Index N (Point N)

As such, a single indexed point list (index buffer), with values {0, … , N - 1} can be shared between all indexed point draw calls of N primitives or fewer.

Line List

An indexed line list (SCE_GXM_PRIMITIVE_LINES) contains two indices for each line.

Figure 7 Indexed Lines



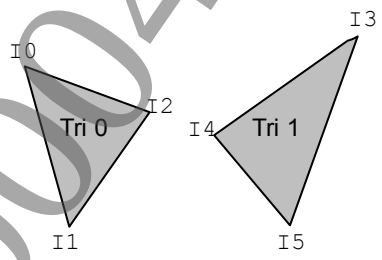
The index values that describe a list of lines would be arranged in the index list as shown below.

16-bit or 32-bit Value
Index 0 (Line 0)
Index 1 (Line 0)
Index 2 (Line 1)
Index 3 (Line 1)
Index 4 (Line 2)
…

Triangle List

An indexed triangle list (`SCE_GXM_PRIMITIVE_TRIANGLES`) contains three indices for each triangle.

Figure 8 Triangle List



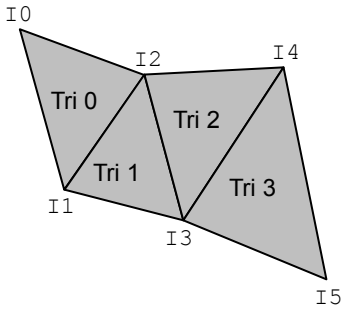
The indices for triangles are arranged as follows:

16-bit or 32-bit Value
Index 0 (Tri 0)
Index 1 (Tri 0)
Index 2 (Tri 0)
Index 3 (Tri 1)
Index 4 (Tri 1)
Index 5 (Tri 1)
…

Triangle Strip

An indexed triangle strip (`SCE_GXM_PRIMITIVE_TRIANGLE_STRIP`) contains a list of indices which are shared between subsequent triangles.

Figure 9 Triangle Strip



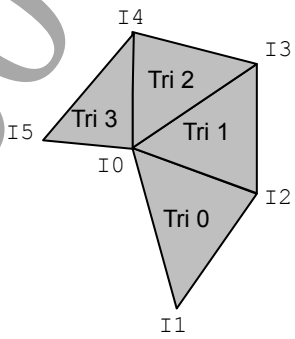
For this example, the indices for triangles strips are arranged as follows:

16-bit or 32-bit Value
Index 0 (Tri 0)
Index 1 (Tri 0, Tri 1)
Index 2 (Tri 0, Tri 1, Tri 2)
Index 3 (Tri 1, Tri 2, Tri 3)
Index 4 (Tri 2, Tri 3)
Index 5 (Tri 3)
…

Triangle Fan

A triangle fan (`SCE_GXM_PRIMITIVE_TRIANGLE_FAN`) contains a list of indices which are shared between subsequent triangles.

Figure 10 Triangle Fan



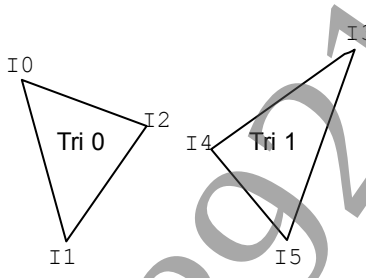
For this example, the indices for triangles fans are arranged as follows:

16-bit or 32-bit Value
Index 0 (Tri 0, Tri 1, Tri 2, Tri 3)
Index 1 (Tri 0)
Index 2 (Tri 0, Tri 1)
Index 3 (Tri 1, Tri 2)
Index 4 (Tri 2, Tri 3)
Index 5 (Tri 3)
…

Triangle Edge List

An indexed triangle edge list (`SCE_GXM_PRIMITIVE_TRIANGLE_EDGES`) contains four indices for each triangle.

Figure 11 Triangles Edge List



Following the indices for each triangle in memory is a single index representing the edge flags for that triangle.

16-bit or 32-bit Value
Index 0 (Tri 0)
Index 1 (Tri 0)
Index 2 (Tri 0)
Edge Flags (Tri 0)
Index 3 (Tri 1)
Index 4 (Tri 1)
Index 5 (Tri 1)
Edge Flags (Tri 1)
…

The edge flags define which triangle edges are visible, built from a bitwise OR of the following values:

Table 2 List of Triangle Edge Flags

Edge Flag	Value	Description
`SCE_GXM_EDGE_ENABLE_01`	0x100	Enable edge 0-1
`SCE_GXM_EDGE_ENABLE_12`	0x200	Enable edge 1-2
`SCE_GXM_EDGE_ENABLE_20`	0x400	Enable edge 2-0

Triangle edge geometry is treated as triangles until rasterization. As such, in order for the edges to be rendered the index values must not be degenerate and the triangle must have non-zero area.

8 Vertex Formats

The following vertex attribute formats are supported:

Table 3 List of Supported Vertex Formats

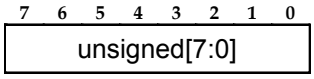
Format	Description
SCE_GXM_ATTRIBUTE_FORMAT_U8	8-bit unsigned integer
SCE_GXM_ATTRIBUTE_FORMAT_S8	8-bit signed integer
SCE_GXM_ATTRIBUTE_FORMAT_U16	16-bit unsigned integer
SCE_GXM_ATTRIBUTE_FORMAT_S16	16-bit signed integer
SCE_GXM_ATTRIBUTE_FORMAT_U8N	8-bit unsigned integer normalized to [0,1] range
SCE_GXM_ATTRIBUTE_FORMAT_S8N	8-bit signed integer normalized to [-1,1] range
SCE_GXM_ATTRIBUTE_FORMAT_U16N	16-bit unsigned integer normalized to [0,1] range
SCE_GXM_ATTRIBUTE_FORMAT_S16N	16-bit signed integer normalized to [-1,1] range
SCE_GXM_ATTRIBUTE_FORMAT_F16	IEEE 754 16-bit half precision floating-point
SCE_GXM_ATTRIBUTE_FORMAT_F32	IEEE 754 32-bit single precision floating-point

The first part of a vertex program will contain instructions to unpack this data into 32-bit floating-point format. The memory layout and conversion to floating point for each format is listed below.

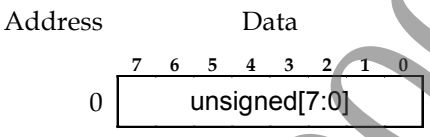
8-bit unsigned

8-bit unsigned format (SCE_GXM_ATTRIBUTE_FORMAT_U8) can represent integers from 0 to 255.

8-bit unsigned occupies 1 byte in memory and has the following 8 bit representation:



This would be stored in memory as follows:

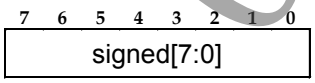


8-bit unsigned values of 0 to 255 will map to values from 0.0 to 255.0 in 32-bit floating-point format.

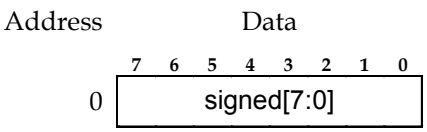
8-bit signed

8-bit signed format (SCE_GXM_ATTRIBUTE_FORMAT_S8) can represent integers from -128 to 127.

8-bit signed occupies 1 byte in memory and has the following 2's complement representation:



This would be stored in memory as follows:

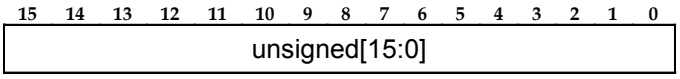


8 bit signed values of -128 to +127 will map to values from -128.0 to 127.0 in 32-bit floating-point format.

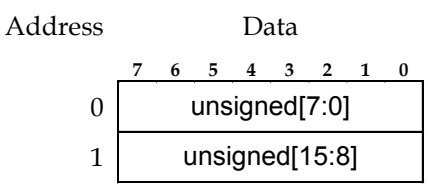
16-bit unsigned

16-bit unsigned format (`SCE_GXM_ATTRIBUTE_FORMAT_U16`) can represent integers from 0 to 65535.

16-bit unsigned occupies 2 bytes in memory and has the following 16 bit representation:



This would be stored in memory as follows:

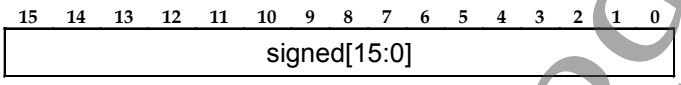


16-bit unsigned values of 0 to 65535 will map to values from 0.0 to 65535.0 in 32-bit floating-point format.

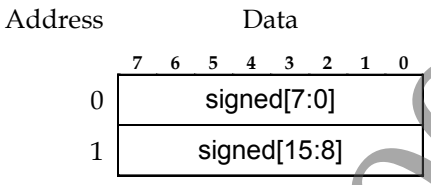
16-bit signed

16-bit signed format (`SCE_GXM_ATTRIBUTE_FORMAT_S16`) can represent integers from -32768 to 32767.

16-bit signed occupies 2 bytes in memory and has the following 2's complement representation:



This would be stored in memory as follows:

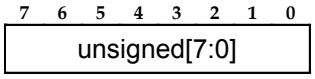


16-bit signed values of -32768 to 32767 will map to values from -32768.0 to 32767.0 in 32-bit floating-point format.

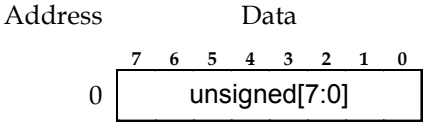
8-bit unsigned normalized

8-bit unsigned normalized format (`SCE_GXM_ATTRIBUTE_FORMAT_U8N`) can represent values from of 0 to 255.

8-bit unsigned occupies 1 byte in memory and has the following 8 bit representation:



This would be stored in memory as follows:



8-bit unsigned normalized values of 0 to 255 will map to values from 0.0 to 1.0 in 32-bit floating-point format using the calculation:

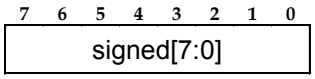
f = x/255.

Where f is the 32-bit floating-point value and x is the 8-bit unsigned normalized value.

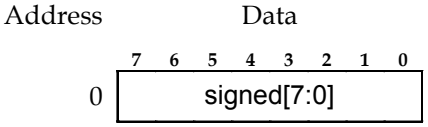
8-bit signed normalized

8-bit signed normalized format (`SCE_GXM_ATTRIBUTE_FORMAT_S8N`) can represent values from -128 to +127.

8-bit signed normalized occupies 1 byte in memory and has the following 8 bit representation:



This would be stored in memory as follows:



8-bit signed normalized values of -128 to +127 will map to values from -1.0 to 1.0 in 32-bit floating-point format using the calculation:

f = clamp (x/127, -1.0, 1.0).

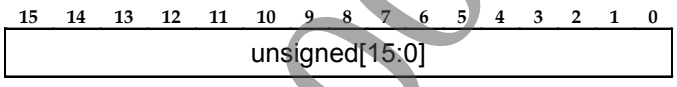
Where f is the 32-bit floating-point value and x is the 8-bit signed normalized value.

Note that 8-bit signed normalized value of -128 and -127 both map to -1.0.

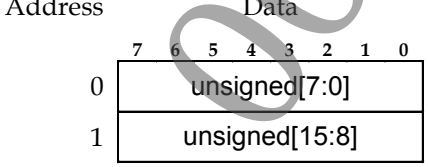
16-bit unsigned normalized

16-bit unsigned normalized format (`SCE_GXM_ATTRIBUTE_FORMAT_U16N`) can represent values from 0 to 65535.

16-bit unsigned normalized occupies 2 bytes in memory and has the following 16 bit representation:



This would be stored in memory as follows:



16-bit unsigned normalized values of 0 to 65535 will map to values from 0.0 to 1.0 in 32-bit floating-point format using the calculation:

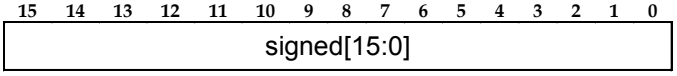
f = x/65535

Where f is the 32-bit floating-point value and x is the 16-bit unsigned normalized value.

16-bit signed normalized

16-bit signed normalized format (`SCE_GXM_ATTRIBUTE_FORMAT_S16N`) can represent values from -32768 to 32767.

16-bit signed normalized occupies 2 bytes in memory and has the following 16 bit representation:



This would be stored in memory as follows:

16-bit signed normalized will map to values from -1.0 to 1.0 in 32-bit floating-point format using the calculation:

F32 = clamp (x/32767, -1.0, 1.0).

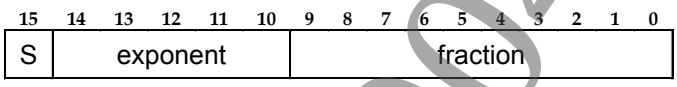
Where f is the 32-bit floating-point value and x is the 16-bit signed normalized value.

Note that 16-bit signed normalized value of -32768 and -32767 both map to -1.0.

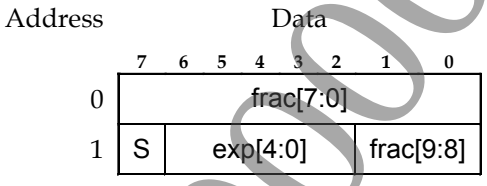
IEEE 754 16-bit half precision floating-point

Half precision floating-point (`SCE_GXM_ATTRIBUTE_FORMAT_F16`) can represent floating-point numbers with an exponent of 5 bits and significant precision on 11 bits (10 bits explicitly stored in the value).

Half precision floating-point occupies 2 bytes in memory and has the following 16 bit representation:



This would be stored in memory as follows:

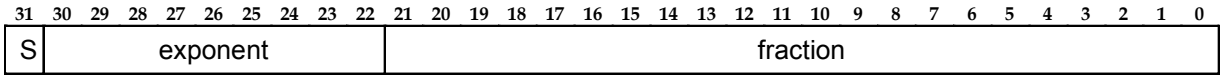


Half precision floating-point values will map to 32-bit floating format using the standard IEEE conversion.

IEEE 754 32-bit single precision floating-point

Single precision floating-point (`SCE_GXM_ATTRIBUTE_FORMAT_F32`) can represent floating-point numbers with an exponent of 8 bits and significant precision on 24 bits (23 bits explicitly stored in the value).

Single precision floating-point occupies 4 bytes in memory and has the following 32 bit representation:



This would be stored in memory as follows:

Address Data

	7	6 5 4 3 2 1 0
0	frac[7:0]	
1	frac[15:8]	
2	exp0	fract[22:16]
3	S	exp[7:1]

9 Textures

Types

Table 4 lists the supported texture types.

Table 4 Supported Texture Types

Type	Notes
SCE_GXM_TEXTURE_LINEAR	2D texture with linear memory layout. Stride is implicit, defined by rounding width up to the next multiple of 8 texels.
SCE_GXM_TEXTURE_LINEAR_STRIDED	2D texture with linear memory layout and explicit stride. This texture type may only have a single mip level. Only the SCE_GXM_TEXTURE_ADDR_CLAMP texture addressing mode is supported for textures of this type, and the min filter cannot be set independently from the mag filter (it always take the value from mag filter).
SCE_GXM_TEXTURE_SWIZZLED	2D texture with swizzled memory layout. Width and height must be a power of 2. Texture addressing modes that use border data can be used with textures of this type.
SCE_GXM_TEXTURE_SWIZZLED_ARBITRARY	2D texture with swizzled memory layout, and arbitrary width and height. Texture addressing modes that use border data can be used with textures of this type.
SCE_GXM_TEXTURE_TILED	2D texture with tiled memory layout.
SCE_GXM_TEXTURE_CUBE	Cubic environment map texture. Each face uses the swizzled memory layout, and must be a power of 2 in width and height.
SCE_GXM_TEXTURE_CUBE_ARBITRARY	Cubic environment map texture. Each face uses the swizzled memory layout, with an arbitrary width and height.

Texture Size

The maximum texture size is 4096 texels in width and height.

Memory Layouts

Memory Alignment

The 16-byte block compressed formats SCE_GXM_TEXTURE_BASE_FORMAT_UBC2 and SCE_GXM_TEXTURE_BASE_FORMAT_UBC3 have the largest alignment restriction for texture data, which is 16 bytes.

The remaining texture formats have less strict alignment constraints, as indicated in each format description.

Linear Textures

Linear texture types are stored in a scan line sequential order as shown in Figure 12 (left figure). When stride is implicit, this is always a multiple of 8 texels.

Figure 12 Linear Texture Memory Layout

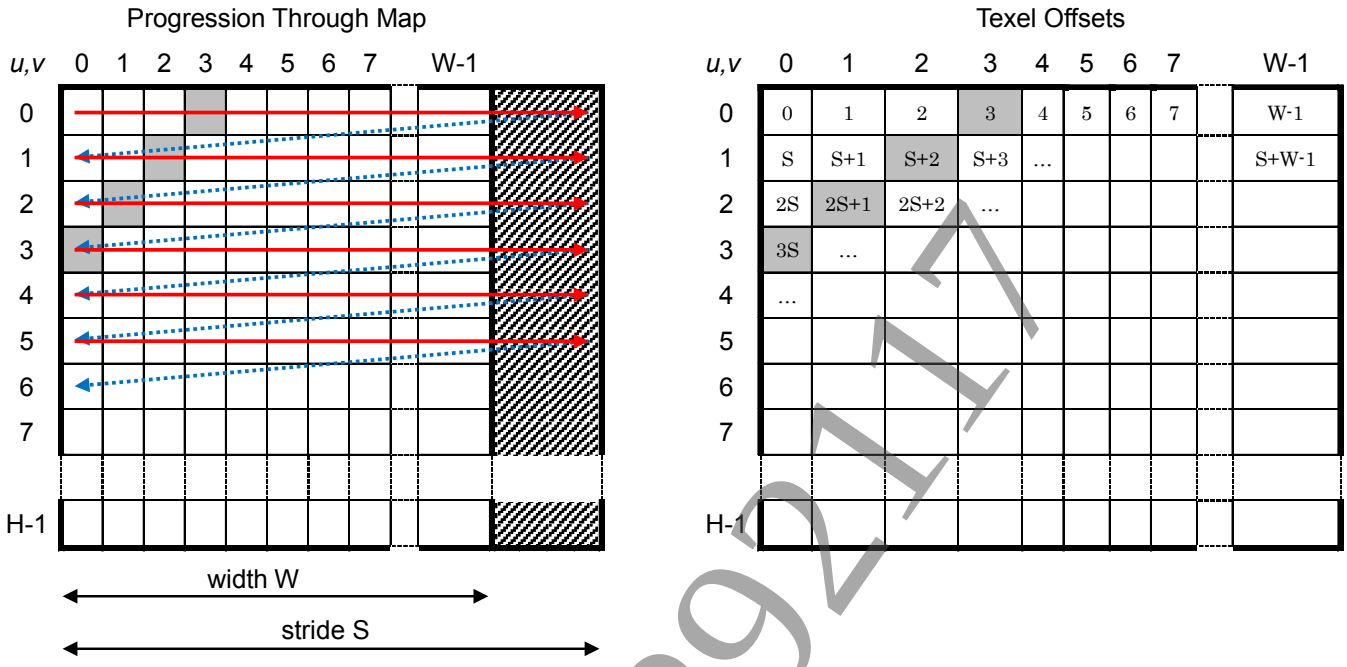
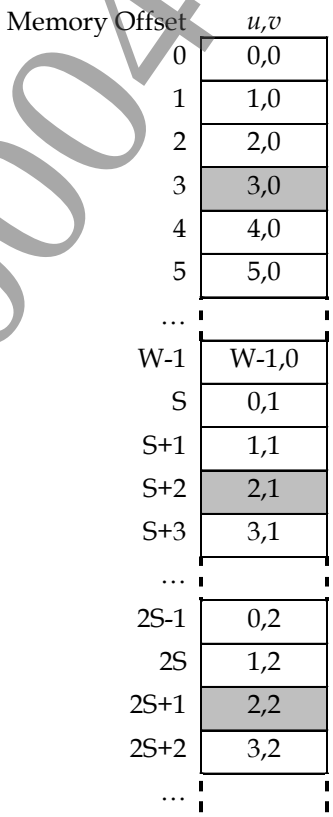


Figure 13 Linear Texture Memory Map



Where W is the width of the texture map, H is the height of the texture map, and S is the stride.

Linear textures with explicit stride are not supported for the 24-bit packed formats `SCE_GXM_TEXTURE_BASE_FORMAT_U8U8U8` or `SCE_GXM_TEXTURE_BASE_FORMAT_S8S8S8`.

Where explicit stride is used with 64-bit formats, the stride must be aligned to 8 bytes. For all other formats the stride must be aligned to 4 bytes.

Mip Maps

When a linear texture contains mipmaps, the offset between the first and second mip levels is based on the smallest enclosing power of 2 texture for the top mip level. The offset between subsequent mip levels is computed by halving these dimensions between each level, clamping the width to at least 8 pixels. Note that only the offsets between mip levels are derived in this way, each mip level internally uses the linear layout for its texel data where stride is derived by rounding the mip width up to the nearest multiple of 8.

For example, a 144x78 linear texture with full mip chain has the offsets between mip levels listed in Table 5:

Table 5 Example Mip Level Offsets for Linear Texture with Full Mip Chain

Mip	Stride (Pixels)	Offset To Next Mip (Pixels)
0 (144x78)	144	32768 (256*128)
1 (72x39)	72	8192 (128*64)
2 (36x19)	40	2048 (64*32)
3 (18x9)	24	512 (32*16)
4 (9x4)	16	128 (16*8)
5 (4x2)	8	32 (8*4)
6 (2x1)	8	16 (8*2)
7 (1x1)	8	8 (8*1)

YUV Linear Textures

When a linear texture has a texture format of **YUV420P2** or **YUV420P3**, the image contains multiple *planes* of data.

Within a YUV texture with a non-zero mip count, the offsets between planes are computed in a similar way to mip level offsets, using the smallest enclosing power of 2 texture for the top mip level. Note that this calculation is only for the offsets, each plane internally uses the linear layout for its texel data where stride is derived by rounding the plane width up to the nearest multiple of 8.

For example a 200x120 YUV420P3 texture with 2 mip levels has the memory layout listed in Table 6:

Table 6 Example Memory Layout for Texture with Two Mip Levels

Mip	Plane	Stride (Pixels)	Offset To Next Plane (Pixels)
0 (200x120)	Y (200x120)	200	32768 (256*128)
	U (100x60)	104	8192 (128*64)
	V (100x60)	104	8192 (128*64)
1 (100x60)	Y (100x60)	104	8192 (128*64)
	U (50x30)	56	2048 (64*32)
	V (50x30)	56	2048 (64*32)

When the mip count of a YUV texture is assigned zero, the texture still contains a single mip level but the memory footprint of each plane is defined only by the linear memory layout, with no extra padding. Additionally, when the mip count is assigned zero for the YUV420P3 format, the width must be a multiple of 16.

For example a 208x120 YUV420P3 texture with 0 mip levels has the memory layout described in Table 7:

Table 7 Example Memory Layout for Texture with Zero Mip Levels

Mip	Plane	Stride (Pixels)	Offset To Next Plane (Pixels)
0 (200x120)	Y (208x120)	208	24960 (208*120)
	U (104x60)	104	6656 (104*60)
	V (104x60)	104	6656 (104*60)

Swizzled Textures

Swizzled textures store texels in Morton order: 2x2 blocks of texels, and 2x2 blocks of blocks and so on.

Figure 14 Swizzled Texture Memory Layout

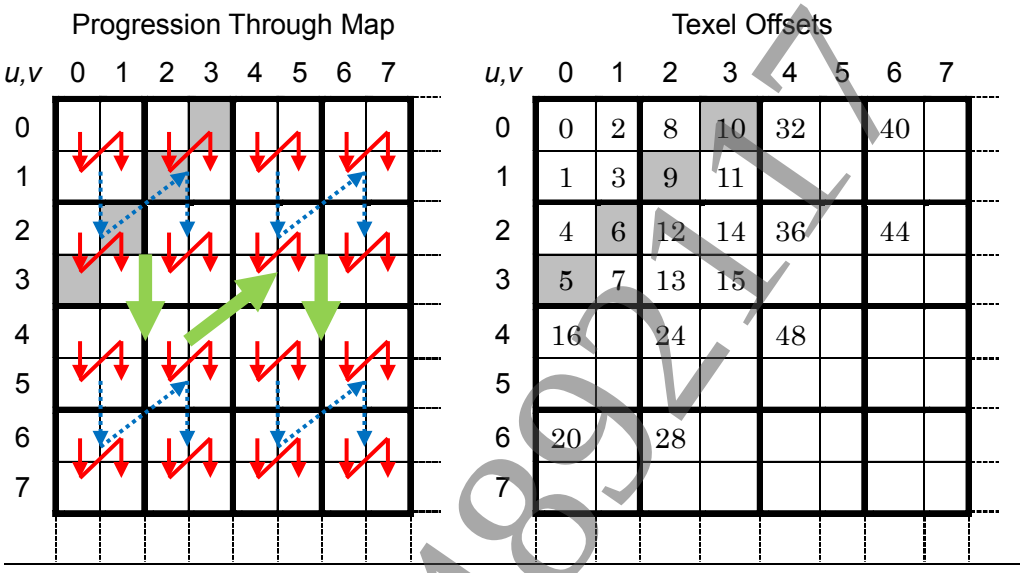


Figure 15 Swizzled Texture Memory Map

Memory Offset	u,v
0	0,0
1	0,1
2	1,0
3	1,1
4	0,2
5	0,3
6	1,2
7	1,3
8	2,0
9	2,1
10	3,0
11	3,1
12	2,2
13	2,3
14	3,2
15	3,3

The address for any given texel is calculated by interleaving the horizontal and vertical texel coordinates as shown in Figure 16:

Figure 16 Swizzled Texture Address Calculation

…	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
and so on	u6	v6	u5	v5	u4	v4	u3	v3	u2	v2	u1	v1	u0	v0

When the texture dimension is not equal on both the vertical and horizontal, the least significant bits of the address are interleaved first. Any remaining bits are appended to the top of the address.

Note that the following block compressed texture formats are also supported in swizzled texture mode:

- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRT2BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRT4BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII2BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII4BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC1`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC2`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC3`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC4`
- `SCE_GXM_TEXTURE_BASE_FORMAT_SBC4`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC5`
- `SCE_GXM_TEXTURE_BASE_FORMAT_SBC5`

When using swizzled block compressed textures, it is the individual blocks that are stored in Morton order in memory. The structure within each block is unchanged.

Mip Maps

When a swizzled texture contains mip maps, they are stored sequentially in memory with no extra padding between mip levels.

Swizzled Arbitrary Textures

Swizzled arbitrary textures allow for the specification of swizzled textures with a width and height that are not a power of two. The memory footprint associated with the texture and any associated mip maps is as detailed for a swizzled texture created with the supplied width and height rounded up to the next power of two.

Tiled Textures

Tiled textures are stored as 32x32 tiles in memory, in scan line sequential order as shown in Figure 17. The texels are also stored in scan line sequential order within each tile.

The width and height of the texture must be at least 32 texels but need not be an exact multiple of 32. However, the memory footprint is always a whole number of tiles. So, for example, it is valid for to have a 48x40 tiled texture, but this will have memory footprint of a 64x64 tiled texture since this is the closest multiple of 32 in width and height.

Figure 17 Tiled Texture Memory Layout

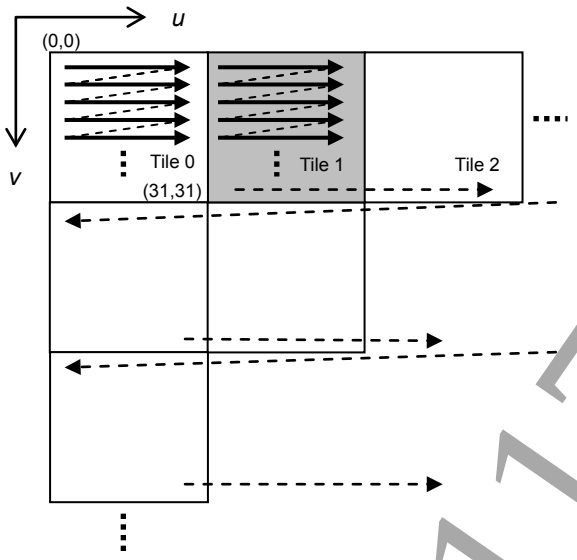
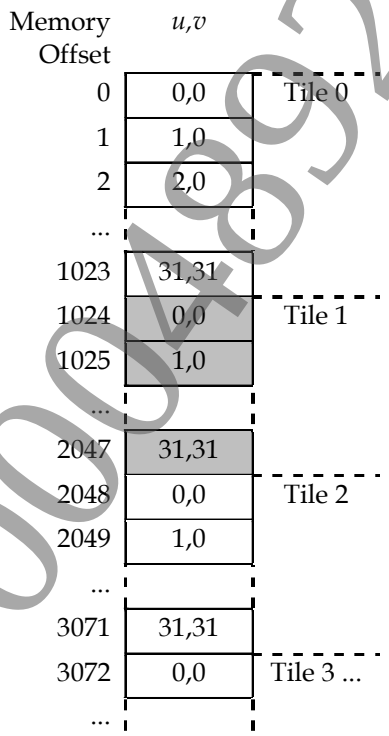
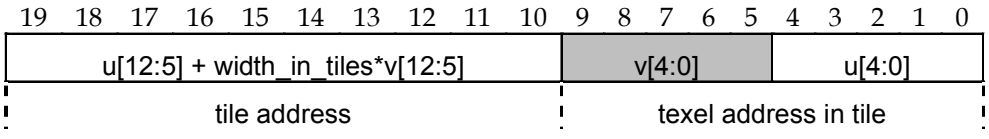


Figure 18 Tiled Texture Memory Map



The address for any given tile and texel is calculated thus:

Figure 19 Tiled Texture Address Calculation



Where:

width_in_tiles = scan line width in tiles

u = horizontal coordinate

v = vertical coordinate

Mip Maps

When a tiled texture contains mip maps, the offset between the first and second mip levels is based on the smallest enclosing power of 2 texture for the top mip level. The offset between subsequent mip levels is computed by halving these dimensions between each level. Note that only the offsets between mip levels are derived in this way, each mip level internally uses the tiled layout for its texel data. The smallest mip level must have dimensions of at least 32x32.

For example, a 129x257 tiled texture with full mip chain has the layout listed in Table 8:

Table 8 Example Layout for Tiled Texture with Full Mip Change

Mip	Tiled Dimensions	Offset To Next Mip (Pixels)
0 (129x257)	160x288	131072 (256*512)
1 (64x128)	64x128	32768 (128*256)
2 (32x64)	32x64	8192 (64*128)

Cubic Environment Maps

Since cube map faces are swizzled, they must be a power of 2 in width and height. For cube maps, the texture data within each face must be swizzled or block-compressed. See section “Swizzled Textures“ for further details on swizzled memory layout.

The following block-compressed texture formats are supported:

- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRT2BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRT4BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII2BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII4BPP`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC1`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC2`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC3`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC4`
- `SCE_GXM_TEXTURE_BASE_FORMAT_SBC4`
- `SCE_GXM_TEXTURE_BASE_FORMAT_UBC5`
- `SCE_GXM_TEXTURE_BASE_FORMAT_SBC5`

YUV texture formats are not supported for cube maps.

If the texture is initialized with a mip count of 0, then only the top mip of each face is present and each face directly follows the previous face in memory with no padding.

If the texture is initialized with a non-zero mip count, then the offset between faces is computed as if all mip levels down to 1x1 are present, even though only a subset of these mip levels may be provided. How the offset between faces must be aligned depends on the following conditions:

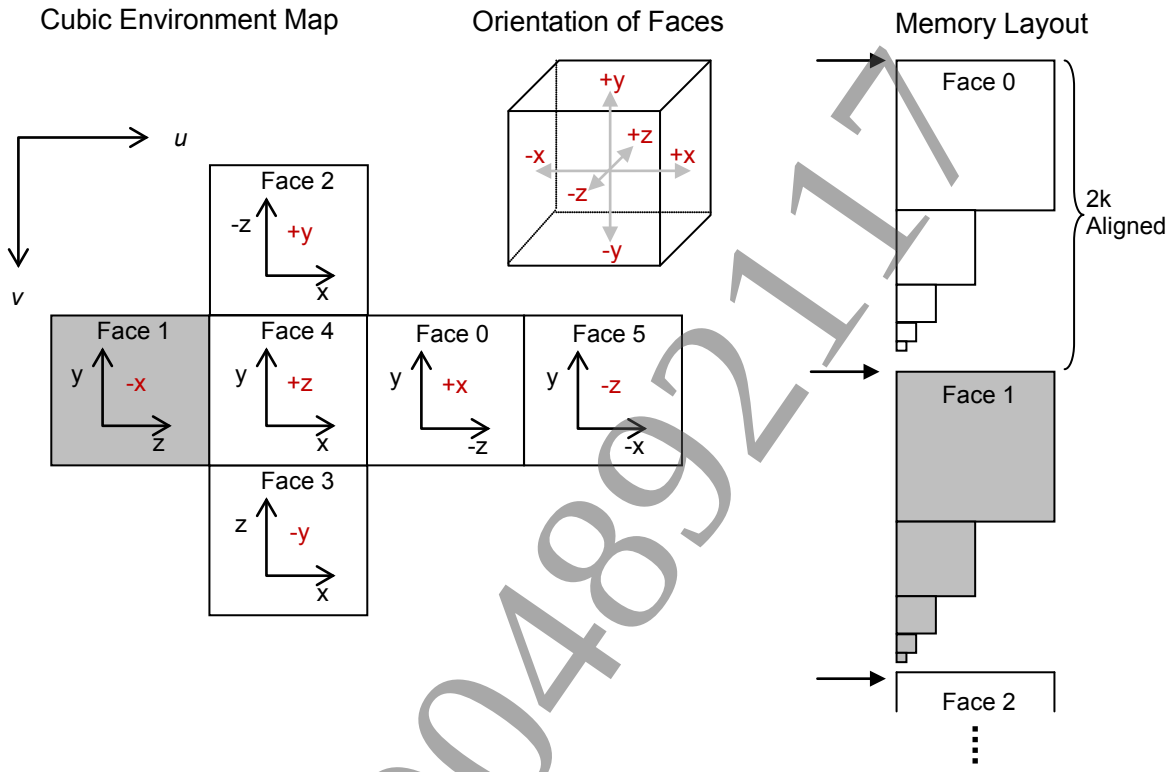
- The texture format is block compressed or 8-bits-per-pixel (8bpp) and the top mip level of each face is 32x32 or larger.
- The texture format is 16-bits-per-pixel (16bpp), 32-bits-per-pixel (32bpp) and the top mip level of each face is 16x16 or larger.
- The texture format is 64-bits-per-pixel (64bpp) and the top mip level of each face is 8x8 or larger.

If none of these conditions are true, then the offset is aligned to 4 bytes. If one of these conditions is true, then the offset is aligned to the next multiple of 2K.

Please note that even when the offset between faces is aligned to the nearest 2K, the start address of the data must still only be aligned to the usual 4 or 8 bytes (depending on the texture format).

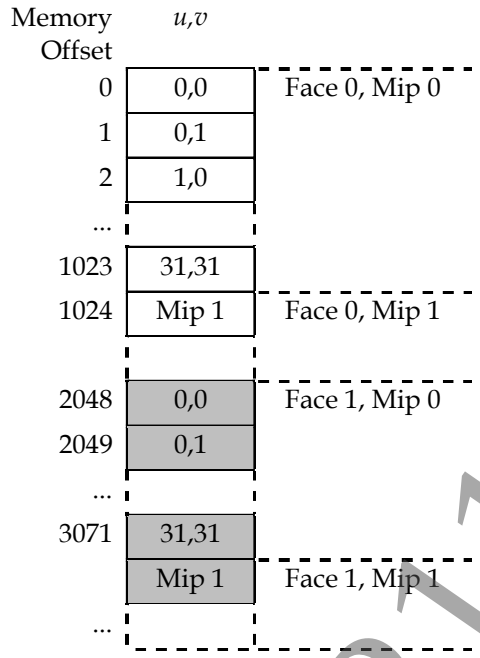
For 2-bits-per-pixel (2bpp) PVRT/PVRTII texture formats, the storage for a mip level is always at least that of an 8x4 texel block. For 4-bits-per-pixel (4bpp) PVRT/PVRTII, all UBC/SBC, and P4 texture formats the storage for a mip level is always at least that of a 4x4 texel block.

Figure 20 Cubic Environment Map Memory Layout



The following example memory map shows a 32x32 cube map with mip mapping, 8-bits-per-pixel and swizzled texture format.

Figure 21 CEM Memory Map



Arbitrary Cubic Environment Maps

Arbitrary cubic environment maps allow for the specification of cubic environment maps with a width and height that are not a power of two. The memory footprint associated with the faces and any associated mip maps is as detailed for a cubic environment map texture created with the supplied width and height rounded up to the next power of two.

Texture Formats

Base Texture Formats

The following texture base formats are supported on the GPU:

Table 9 List of Supported Texture Base Formats

Format	Description	Matching Color Format
SCE_GXM_TEXTURE_BASE_FORMAT_U8	Unsigned 8-bit.	U8
SCE_GXM_TEXTURE_BASE_FORMAT_U16	Unsigned 16-bit.	U16
SCE_GXM_TEXTURE_BASE_FORMAT_F16	16-bit half precision floating-point.	F16
SCE_GXM_TEXTURE_BASE_FORMAT_F32M	32-bit floating-point with masked sign bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_U8U8	Two unsigned 8-bit.	U8U8
SCE_GXM_TEXTURE_BASE_FORMAT_F16F16	Two 16-bit half precision floating-point.	F16F16
SCE_GXM_TEXTURE_BASE_FORMAT_U5U6U5	Unsigned 5-bit, unsigned 6-bit, unsigned 5-bit.	U5U6U5
SCE_GXM_TEXTURE_BASE_FORMAT_U4U4U4U4	Four unsigned 4-bit.	U4U4U4U4
SCE_GXM_TEXTURE_BASE_FORMAT_U8U8U8U8	Four unsigned 8-bit.	U8U8U8U8
SCE_GXM_TEXTURE_BASE_FORMAT_U1U5U5U5	Unsigned 1-bit, unsigned 5-bit, unsigned 5-bit, unsigned 5-bit.	U1U5U5U5

Format	Description	Matching Color Format
SCE_GXM_TEXTURE_BASE_FORMAT_PVRT2BPP	PVRT-I compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_PVRT4BPP	PVRT-I compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII2BPP	PVRT-II compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII4BPP	PVRT-II compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_UBC1	UBC1 compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_UBC2	UBC2 compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_UBC3	UBC3 compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_UBC4	UBC4 compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_SBC4	SBC4 compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_UBC5	UBC5 compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_SBC5	SBC5 compressed format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_YUV422	Interleaved YUV format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_S8	Signed 8-bit.	S8
SCE_GXM_TEXTURE_BASE_FORMAT_S16	Signed 16-bit.	S16
SCE_GXM_TEXTURE_BASE_FORMAT_U32	Unsigned 32-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_S32	Signed 32-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_F32	32-bit floating-point.	F32
SCE_GXM_TEXTURE_BASE_FORMAT_X8U24	Ignored 8-bit, unsigned 24-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_S8S8	Two signed 8-bit.	S8S8
SCE_GXM_TEXTURE_BASE_FORMAT_U16U16	Two unsigned 16-bit.	U16U16
SCE_GXM_TEXTURE_BASE_FORMAT_S16S16	Two signed 16-bit.	S16S16
SCE_GXM_TEXTURE_BASE_FORMAT_F32F32	Two 32-bit floating-point.	F32F32
SCE_GXM_TEXTURE_BASE_FORMAT_U32U32	Two unsigned 32-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_S5S5U6	Signed 5-bit, signed 5-bit, unsigned 6-bit.	S5S5U6
SCE_GXM_TEXTURE_BASE_FORMAT_SE5M9M9M9	Shared 5 bit exponent, three partial-precision floating-point mantissa.	SE5M9M9M9
SCE_GXM_TEXTURE_BASE_FORMAT_F11F11F10	Three partial-precision floating-point.	F11F11F10
SCE_GXM_TEXTURE_BASE_FORMAT_X8S8S8U8	Ignored 8-bit, signed 8-bit, signed 8-bit, unsigned 8-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_U8U8U8	Three unsigned 8-bit.	U8U8U8
SCE_GXM_TEXTURE_BASE_FORMAT_S8S8S8	Three signed 8-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_U8U3U3U2	Unsigned 8-bit, unsigned 3-bit, unsigned 3-bit, unsigned 2-bit.	U8U3U3U2
SCE_GXM_TEXTURE_BASE_FORMAT_S8S8S8S8	Four signed 8-bit.	S8S8S8S8
SCE_GXM_TEXTURE_BASE_FORMAT_U16U16U16U16	Four unsigned 16-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_S16S16S16S16	Four signed 16-bit.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_F16F16F16F16	Four 16-bit half precision floating-point.	F16F16F16F16
SCE_GXM_TEXTURE_BASE_FORMAT_U2U10U10U10	Unsigned 2-bit, unsigned 10-bit, unsigned 10-bit, unsigned 10-bit.	U2U10U10U10
SCE_GXM_TEXTURE_BASE_FORMAT_U2F10F10F10	Unsigned 2-bit and three partial-precision floating-point.	U2F10F10F10
SCE_GXM_TEXTURE_BASE_FORMAT_P4	4-bit index color look up table.	not applicable

Format	Description	Matching Color Format
SCE_GXM_TEXTURE_BASE_FORMAT_P8	8-bit index color look up table.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P2	2-plane YUV format.	not applicable
SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P3	3-plane YUV format.	not applicable

Texture Query Format

The filtered (and potentially gamma corrected) results of a texture read are, by default, passed to the shader code as a half4 value. Most texture formats support conversion to this half4 format as part of the texture read, but not all.

The texture query format can be controlled in shader code by using a Cg syntax extension, and must be used for texture formats that do not support half4.

The full list of Cg types that can be used as texture query formats is:

- float4, float2, float
- half4 (the default), half2, half
- unsigned int4, unsigned int2, unsigned int
- signed int4, signed int2, signed int
- unsigned short4, unsigned short2, unsigned short
- signed short4, signed short2, signed short
- unsigned char4, unsigned char2, unsigned char
- signed char4, signed char2, signed char

This format should be appended to a texture query function in template syntax, such as:

```
unsigned int i = tex2D<unsigned int>(someTex, someUv);
```

Please see the *Shader Compiler User's Guide* document for more details on texture query format syntax. It is the responsibility of the shader author to ensure that the texture formats being used at runtime support the query format used in the shader code.

When 2-component query formats are used, these take the G and R components of the result. When 1-component query formats are used, these take the R component of the result.

For a single table that shows all texture base formats and their support query formats, see the summary table in Appendix A.

Gamma Correction

When gamma correction is enabled the texture unit converts values from the sRGB color space to a linear color space and are passed to the shader. The component values from memory that can be gamma corrected will depend on the texture format. When gamma correcting textures, the texture query format must be half or float precision.

Normalization

As part of the process of converting integer texture components to floating point, the texture unit can normalize the results in the range [0.0, 1.0] for unsigned components and [-1.0, 1.0] for signed components. For block-compressed formats, YUV formats, and other formats containing 8-bit, 16-bit, or 24-bit integer components (as listed in Table 10) that are queried with either half or float precision, this normalization step can be disabled. The results available in shader code are subsequently floating point values in the range associated with the component size. For example, [0.0, 255.0] for U8 data.

Table 10 Texture Formats that Support Normalization Configuration

Format
SCE_GXM_TEXTURE_BASE_FORMAT_U8
SCE_GXM_TEXTURE_BASE_FORMAT_U16
SCE_GXM_TEXTURE_BASE_FORMAT_U8U8
SCE_GXM_TEXTURE_BASE_FORMAT_U5U6U5
SCE_GXM_TEXTURE_BASE_FORMAT_U4U4U4U4
SCE_GXM_TEXTURE_BASE_FORMAT_U8U8U8U8
SCE_GXM_TEXTURE_BASE_FORMAT_U1U5U5U5
SCE_GXM_TEXTURE_BASE_FORMAT_PVRT2BPP
SCE_GXM_TEXTURE_BASE_FORMAT_PVRT4BPP
SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII2BPP
SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII4BPP
SCE_GXM_TEXTURE_BASE_FORMAT_UBC1
SCE_GXM_TEXTURE_BASE_FORMAT_UBC2
SCE_GXM_TEXTURE_BASE_FORMAT_UBC3
SCE_GXM_TEXTURE_BASE_FORMAT_UBC4
SCE_GXM_TEXTURE_BASE_FORMAT_SBC4
SCE_GXM_TEXTURE_BASE_FORMAT_UBC5
SCE_GXM_TEXTURE_BASE_FORMAT_SBC5
SCE_GXM_TEXTURE_BASE_FORMAT_YUV422
SCE_GXM_TEXTURE_BASE_FORMAT_S8
SCE_GXM_TEXTURE_BASE_FORMAT_S16
SCE_GXM_TEXTURE_BASE_FORMAT_X8U24
SCE_GXM_TEXTURE_BASE_FORMAT_S8S8
SCE_GXM_TEXTURE_BASE_FORMAT_U16U16
SCE_GXM_TEXTURE_BASE_FORMAT_S16S16
SCE_GXM_TEXTURE_BASE_FORMAT_S5S5U6
SCE_GXM_TEXTURE_BASE_FORMAT_X8S8S8U8
SCE_GXM_TEXTURE_BASE_FORMAT_U8U8U8
SCE_GXM_TEXTURE_BASE_FORMAT_S8S8S8
SCE_GXM_TEXTURE_BASE_FORMAT_U8U3U3U2
SCE_GXM_TEXTURE_BASE_FORMAT_S8S8S8S8
SCE_GXM_TEXTURE_BASE_FORMAT_U16U16U16U16
SCE_GXM_TEXTURE_BASE_FORMAT_S16S16S16S16
SCE_GXM_TEXTURE_BASE_FORMAT_P4
SCE_GXM_TEXTURE_BASE_FORMAT_P8
SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P2
SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P3

Normalization configuration is ignored under the following conditions:

- When gamma correction is enabled, the texture results are always normalized to the range [0.0, 1.0] regardless of the setting.
- When the texture format is `SCE_GXM_TEXTURE_BASE_FORMAT_U2U10U10U10`, the results are always normalized to the range [0.0, 1.0] regardless of the setting.
- If the texture format does not support half or float precision texture queries, such as `SCE_GXM_TEXTURE_BASE_FORMAT_U32`, the setting is ignored.
- If the texture format is already floating point data, such as `SCE_GXM_TEXTURE_BASE_FORMAT_F16`, the setting is ignored.

Derived Texture Formats

A texture format is defined by combining a base texture format and compatible swizzle mode. The libgxm library provides an enumeration of all texture formats supported by the GPU. For more details, see `SceGxmTextureFormat` in the *libgxm Reference*.

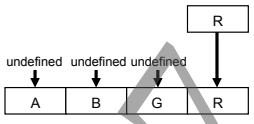
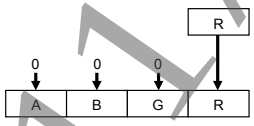
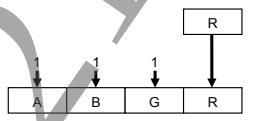
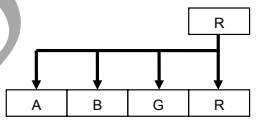
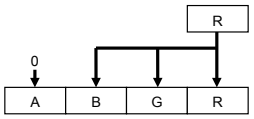
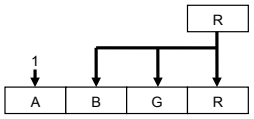
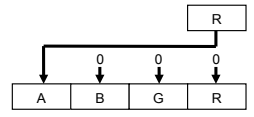
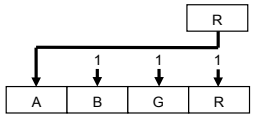
Single Component Base Texture Formats

The single component base texture formats have a single red component value.

Single Component Swizzle Modes

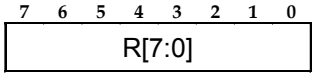
Unless otherwise documented in the texture format description, the value from memory is passed to the shader in ABGR component ordering according to the swizzle mode shown in Table 11.

Table 11 Single Component Memory Swizzle Modes

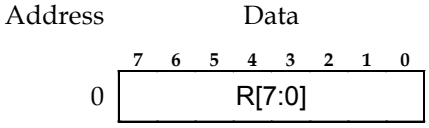
Swizzle Mode	ABGR Representation
SCE_GXM_TEXTURE_SWIZZLE1_R	
SCE_GXM_TEXTURE_SWIZZLE1_000R	
SCE_GXM_TEXTURE_SWIZZLE1_111R	
SCE_GXM_TEXTURE_SWIZZLE1_RRRR	
SCE_GXM_TEXTURE_SWIZZLE1_0RRR	
SCE_GXM_TEXTURE_SWIZZLE1_1RRR	
SCE_GXM_TEXTURE_SWIZZLE1_R000	
SCE_GXM_TEXTURE_SWIZZLE1_R111	

U8

The U8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U8) requires 4-byte alignment. Each texel occupies 1 byte in memory and has the following 8 bit representation:



This would be stored in memory as follows:



Where R is interpreted as the unsigned 8-bit red component value.

The swizzle modes as shown in Table 11 are supported.

The supported query formats are shown in the following table.

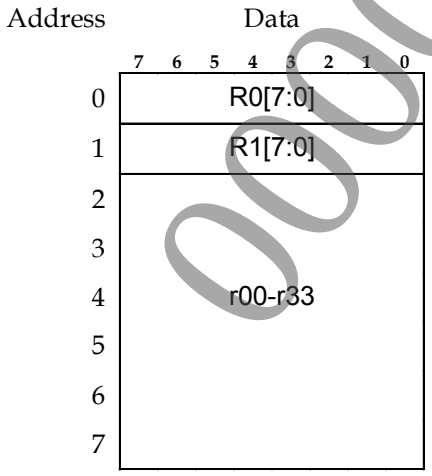
Table 12 U8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• unsigned char4 • half4 - When gamma correction is enabled, the R color component from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R color component from memory will be gamma corrected.
SCE_GXM_TEXTURE_SWIZZLE1_R	• unsigned char • half - When gamma correction is enabled, the R color component from memory will be gamma corrected. • float - When gamma correction is enabled, the R color component from memory will be gamma corrected.

UBC4

The UBC4 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_UBC4) is a block compressed format.

The UBC4 format requires 8-byte alignment. Each 4x4 pixel block occupies 8 bytes in memory and has the following representation in memory:



Where R0 and R1 are the reference values in U8 format, and the block r00-r33 is the 16 3-bit indices.

The swizzle modes as shown in Table 11 are supported.

The supported query formats are shown in the following table.

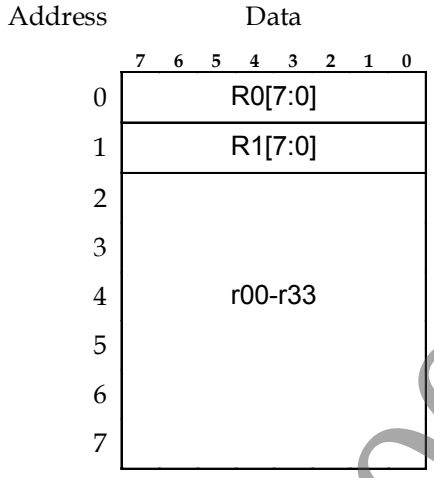
Table 13 UBC4 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• unsigned char4 • half4 - When gamma correction is enabled, the R color component will be gamma corrected before swizzle. • float4 - When gamma correction is enabled, the R color component will be gamma corrected before swizzle.
SCE_GXM_TEXTURE_SWIZZLE1_R	• unsigned char • half - When gamma correction is enabled, the R color component will be gamma corrected before swizzle. • float - When gamma correction is enabled, the R color component will be gamma corrected before swizzle.

SBC4

The SBC4 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_SBC4) is a block compressed format.

The SBC4 format requires 8-byte alignment. Each 4x4 pixel block occupies 8 bytes in memory and has the following representation in memory:



Where R0 and R1 are the reference values in S8 format and the block r00-r33 is the 16 3-bit indices.

The swizzle modes as shown in Table 11 are supported.

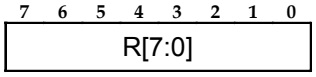
The supported query formats are shown in the following table.

Table 14 SBC4 Query Formats

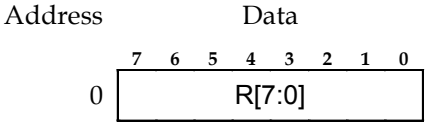
Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• signed char4 • half4 • float4
SCE_GXM_TEXTURE_SWIZZLE1_R	• signed char • half • float

S8

The S8 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_S8`) requires 4-byte alignment. Each texel occupies 1 byte in memory and has the following 8 bit representation:



This would be stored in memory as follows:



Where R is interpreted as the signed 8-bit red component value.

The swizzle modes as shown in Table 11 are supported.

The supported result formats are shown in the following table.

Table 15 S8 Query Formats

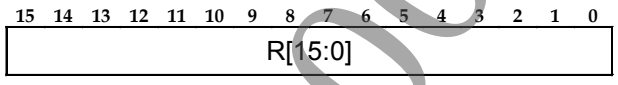
Swizzle Mode	Supported Query Formats

`SCE_GXM_TEXTURE_SWIZZLE1_000R`
`SCE_GXM_TEXTURE_SWIZZLE1_111R`
`SCE_GXM_TEXTURE_SWIZZLE1_RRRR`
`SCE_GXM_TEXTURE_SWIZZLE1_0RRR`
`SCE_GXM_TEXTURE_SWIZZLE1_1RRR`
`SCE_GXM_TEXTURE_SWIZZLE1_R000`
`SCE_GXM_TEXTURE_SWIZZLE1_R111`
	• signed char4
	• half4
	• float4

`SCE_GXM_TEXTURE_SWIZZLE1_R`	• signed char • half • float

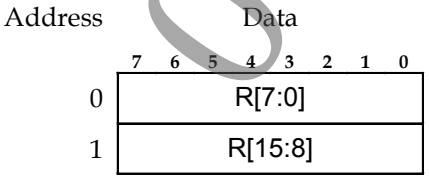
U16

The U16 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_U16`) requires 4-byte alignment. Each texel occupies 2 bytes in memory and has the following 16-bit representation:



Where R is the unsigned 16-bit red component value.

This would be stored in memory as follows:



The swizzle modes as shown in Table 11 are supported.

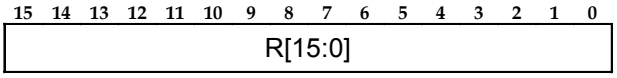
The supported query formats are shown in the following table.

Table 16 U16 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• unsigned short4 • float4
SCE_GXM_TEXTURE_SWIZZLE1_R	• unsigned short • float

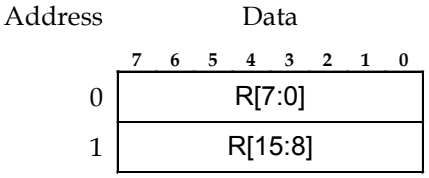
S16

The S16 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_S16) requires 4-byte alignment. Each texel occupies 2 bytes in memory and has the following 16-bit representation:



Where R is the signed 16-bit red component value.

This would be stored in memory as follows:



The swizzle modes as shown in Table 11 are supported.

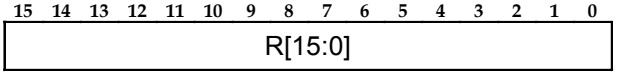
The supported query formats are shown in the following table.

Table 17 S16 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• signed short4 • float4
SCE_GXM_TEXTURE_SWIZZLE1_R	• signed short • float

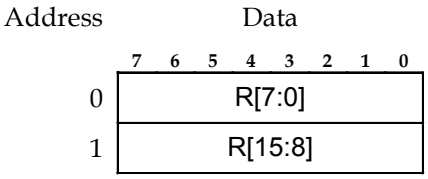
F16

The F16 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_F16) requires 4-byte alignment. Each texel occupies 2 bytes in memory and has the following 16-bit representation:



Where R is the 16-bit half precision floating-point red component value.

This would be stored in memory as follows:



The swizzle modes as shown in Table 11 are supported.

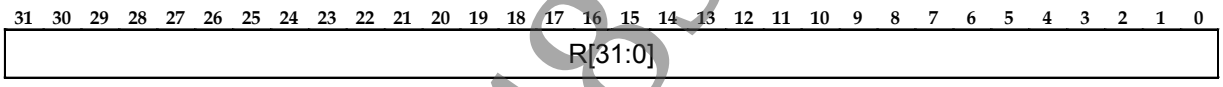
The supported query formats are shown in the following table.

Table 18 F16 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• half4 • float4
SCE_GXM_TEXTURE_SWIZZLE1_R	• half • float

U32

The U32 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U32) requires 4-byte alignment. Each texel occupies 4 bytes in memory and has the following 32-bit representation:



Where R is the unsigned 32-bit red component value.

This would be stored in memory as follows:

Linear filtering is not supported for this texture format.

The swizzle modes as shown in Table 11 are supported.

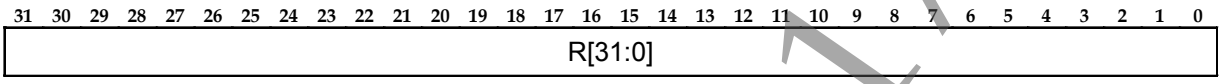
The supported query formats are shown in the following table.

Table 19 U32 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• unsigned int4
SCE_GXM_TEXTURE_SWIZZLE1_R	• unsigned int

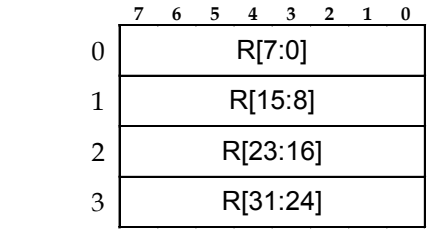
S32

The S32 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_S32) requires 4-byte alignment. Each texel occupies 4 bytes in memory and has the following 32-bit representation:



Where R is the unsigned 32-bit red component value.

This would be stored in memory as follows:



Linear filtering is not supported for this texture format.

The swizzle modes as shown in Table 11 are supported.

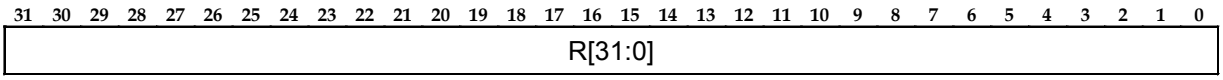
The supported query formats are shown in the following table.

Table 20 S32 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• int4
SCE_GXM_TEXTURE_SWIZZLE1_R	• int

F32

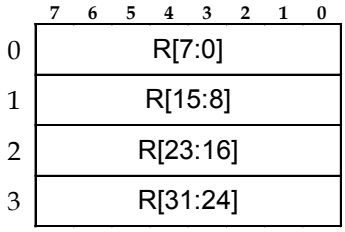
The F32 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_F32) requires 4-byte alignment. Each texel occupies 4 bytes in memory and has the following 32-bit representation:



Where R is the 32-bit floating-point red component value.

This would be stored in memory as follows:

Address Data



The GPU only supports linear filtering of this format when used with a shadow map compare in shader code. Linear filtering of the raw texture data is not supported.

The swizzle modes as shown in Table 11 are supported.

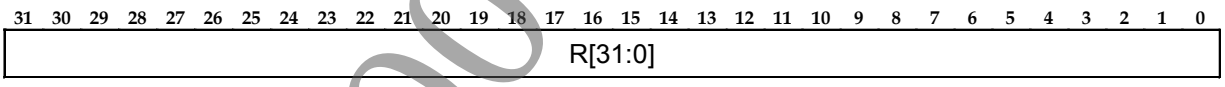
The supported query formats are shown in the following table.

Table 21 F32 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• float4
SCE_GXM_TEXTURE_SWIZZLE1_R	• float

F32M

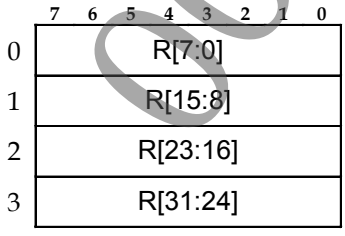
The F32M texture format (SCE_GXM_TEXTURE_BASE_FORMAT_F32M) requires 4-byte alignment. Each texel occupies 4 bytes in memory and has the following 32-bit representation:



Where R is the 32-bit floating-point red component value with masked sign bit. The resulting value is always positive.

This would be stored in memory as follows:

Address Data



The GPU only supports linear filtering of this format when used with a shadow map compare in shader code. Linear filtering of the raw texture data is not supported.

The swizzle modes as shown in Table 11 are supported.

The supported query formats are shown in the following table.

Table 22 F32M Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE1_000R SCE_GXM_TEXTURE_SWIZZLE1_111R SCE_GXM_TEXTURE_SWIZZLE1_RRRR SCE_GXM_TEXTURE_SWIZZLE1_0RRR SCE_GXM_TEXTURE_SWIZZLE1_1RRR SCE_GXM_TEXTURE_SWIZZLE1_R000 SCE_GXM_TEXTURE_SWIZZLE1_R111	• float4
SCE_GXM_TEXTURE_SWIZZLE1_R	• float

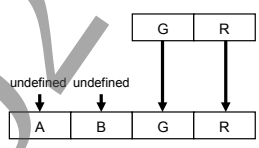
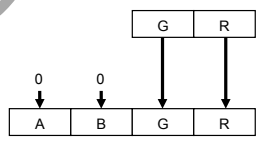
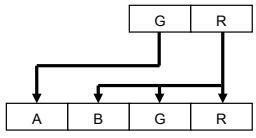
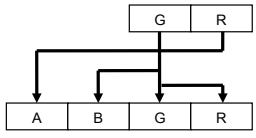
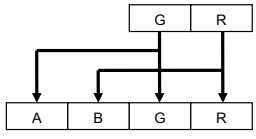
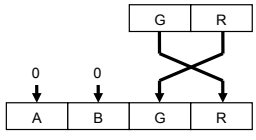
Two Component Base Texture Formats

The two component base texture formats only have red and green component values.

Two Component Swizzle Modes

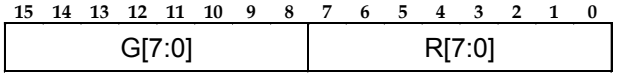
Unless otherwise documented in the texture format description, the values from memory are passed to the shader in ABGR component ordering according to the swizzle mode shown in Table 23.

Table 23 Two Component Swizzle Modes

Swizzle Mode	ABGR Representation
SCE_GXM_TEXTURE_SWIZZLE2_GR	
SCE_GXM_TEXTURE_SWIZZLE2_00GR	
SCE_GXM_TEXTURE_SWIZZLE2_GRRR	
SCE_GXM_TEXTURE_SWIZZLE2_RGGG	
SCE_GXM_TEXTURE_SWIZZLE2_GRGR	
SCE_GXM_TEXTURE_SWIZZLE2_00RG	

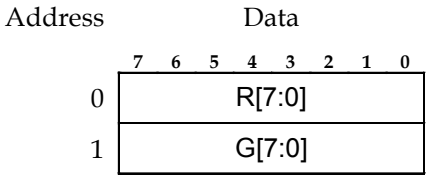
U8U8

The U8U8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U8U8) requires 4-byte alignment. Each texel occupies 2 bytes in memory and has the following 16-bit representation:



Where R is the unsigned 8-bit red, and G is the unsigned 8-bit green component values.

This would be stored in memory as follows:



The swizzle modes as shown in Table 23 are supported.

The supported query formats are shown in the following table.

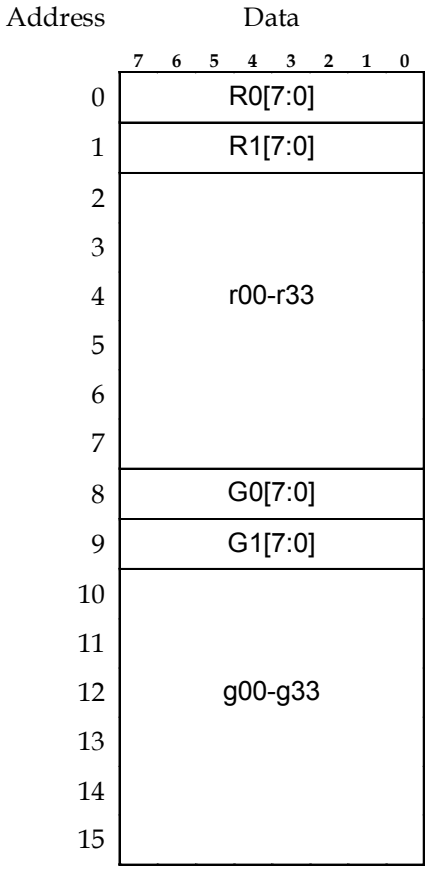
Table 24 U8U8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE2_00GR SCE_GXM_TEXTURE_SWIZZLE2_GRRR SCE_GXM_TEXTURE_SWIZZLE2_RGGG SCE_GXM_TEXTURE_SWIZZLE2_GRGR SCE_GXM_TEXTURE_SWIZZLE2_00RG	• unsigned char4 • half4 - Gamma correction can be enabled on just the R color component from memory, or both the R and G color components from memory. • float4 - Gamma correction can be enabled on just the R color component from memory, or both the R and G color components from memory.
SCE_GXM_TEXTURE_SWIZZLE2_GR	• unsigned char2 • half2 - Gamma correction can be enabled on just the R color component from memory, or both the R and G color components from memory. • float2 - Gamma correction can be enabled on just the R color component from memory, or both the R and G color components from memory.

UBC5

The UBC5 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_UBC5) is a block compressed format.

The UBC5 format requires 16-byte alignment. Each 4x4 pixel block occupies 16 bytes in memory and has the following representation in memory:



Where R0, R1, G0 and G1 are the reference values in U8 format, and the blocks r00-r33 and g00-g33 are blocks of 16 3-bit indices.

The swizzle modes as shown in Table 23 are supported.

The supported query formats are shown in the following table.

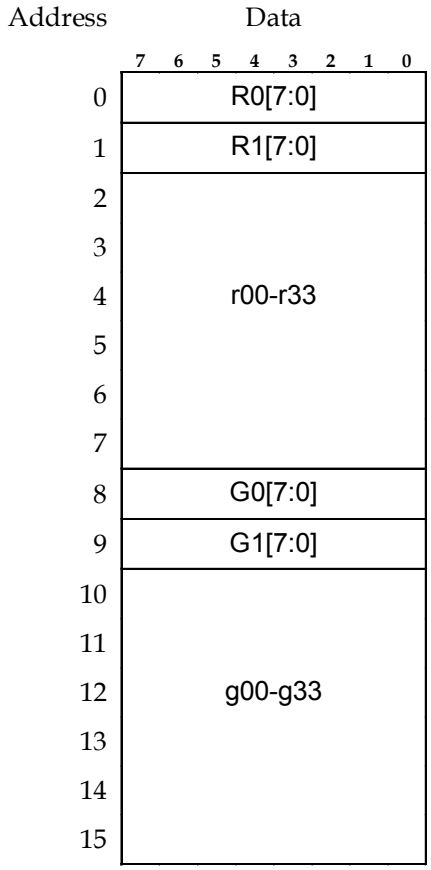
Table 25 UBC5 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE2_00GR SCE_GXM_TEXTURE_SWIZZLE2_GRRR SCE_GXM_TEXTURE_SWIZZLE2_RGGG SCE_GXM_TEXTURE_SWIZZLE2_GRGR SCE_GXM_TEXTURE_SWIZZLE2_00RG	• unsigned char4 • half4 - Gamma correction can be enabled on just the R color component before swizzle, or both the R and G color components before swizzle. • float4 - Gamma correction can be enabled on just the R color component before swizzle, or both the R and G color components before swizzle.
SCE_GXM_TEXTURE_SWIZZLE2_GR	• unsigned char2 • half2 - Gamma correction can be enabled on just the R color component before swizzle, or both the R and G color components before swizzle. • float2 - Gamma correction can be enabled on just the R color component before swizzle, or both the R and G color components before swizzle.

SBC5

The SBC5 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_SBC5) is a block compressed format.

The SBC5 format requires 16-byte alignment. Each 4x4 pixel block occupies 16 bytes in memory and has the following representation in memory:



Where R0, R1, G0 and G1 are the reference values in S8 format, and the blocks r00-r33 and g00-g33 are blocks of 16 3-bit indices.

The swizzle modes as shown in Table 23 are supported.

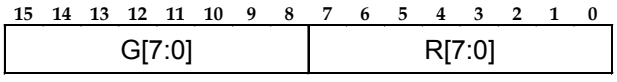
The supported query formats are shown in the following table.

Table 26 SBC5 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE2_00GR SCE_GXM_TEXTURE_SWIZZLE2_GRRR SCE_GXM_TEXTURE_SWIZZLE2_RGGG SCE_GXM_TEXTURE_SWIZZLE2_GRGR SCE_GXM_TEXTURE_SWIZZLE2_00RG	• signed char4 • half4 • float4
SCE_GXM_TEXTURE_SWIZZLE2_GR	• signed char2 • half2 • float2

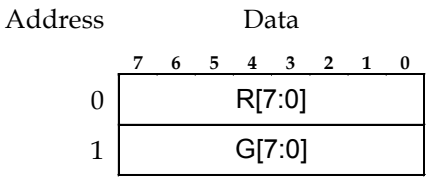
S8S8

The S8S8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_S8S8) requires 4-byte alignment. Each texel occupies 2 bytes in memory and has the following 16-bit representation:



Where R is the signed 8-bit red, and G is the signed 8-bit green component values.

This would be stored in memory as follows:



The swizzle modes as shown in Table 23 are supported.

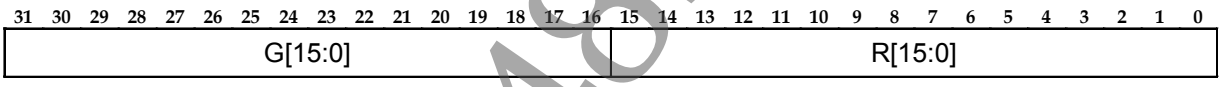
The supported query formats are shown in the following table.

Table 27 S8S8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE2_00GR SCE_GXM_TEXTURE_SWIZZLE2_GRRR SCE_GXM_TEXTURE_SWIZZLE2_RGGG SCE_GXM_TEXTURE_SWIZZLE2_GRGR SCE_GXM_TEXTURE_SWIZZLE2_00RG	• signed char4 • half4 • float4
SCE_GXM_TEXTURE_SWIZZLE2_GR	• signed char2 • half2 • float2

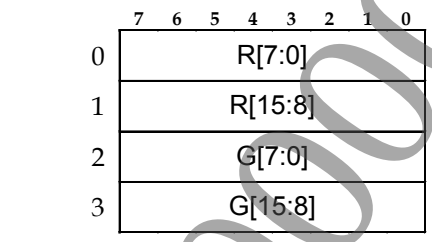
U16U16

The U16U16 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U16U16) requires 4-byte alignment. Each texel occupies 4 bytes in memory and has the following 32-bit representation:



Where R is the unsigned 16-bit red, and G is the unsigned 16-bit green component values.

This would be stored in memory as follows:



The swizzle modes as shown in Table 23 are supported.

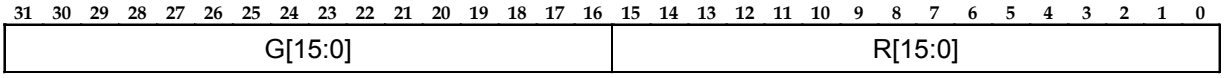
The supported query formats are shown in the following table.

Table 28 U16U16 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE2_00GR SCE_GXM_TEXTURE_SWIZZLE2_GRRR SCE_GXM_TEXTURE_SWIZZLE2_RGGG SCE_GXM_TEXTURE_SWIZZLE2_GRGR SCE_GXM_TEXTURE_SWIZZLE2_00RG	• unsigned short4 • float4
SCE_GXM_TEXTURE_SWIZZLE2_GR	• unsigned short2 • float2

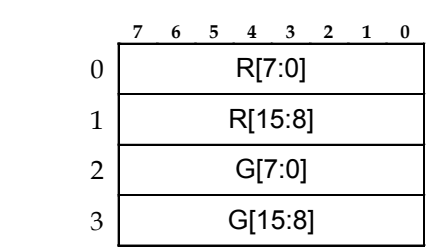
S16S16

The S16S16 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_S16S16`) requires 4-byte alignment. Each texel occupies 4 bytes in memory and has the following 32-bit representation:



Where R is the signed 16-bit red, and G is the signed 16-bit green component values.

This would be stored in memory as follows:



The swizzle modes as shown in Table 23 are supported.

The supported query formats are shown in the following table.

Table 29 S16S16 Query Formats

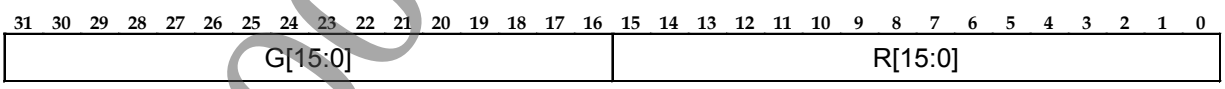
Swizzle Mode	Supported Query Formats

`SCE_GXM_TEXTURE_SWIZZLE2_00GR`
`SCE_GXM_TEXTURE_SWIZZLE2_GRRR`
`SCE_GXM_TEXTURE_SWIZZLE2_RGGG`
`SCE_GXM_TEXTURE_SWIZZLE2_GRGR`
`SCE_GXM_TEXTURE_SWIZZLE2_00RG`
	• signed short4
	• float4

`SCE_GXM_TEXTURE_SWIZZLE2_GR`	• signed short2 • float2

F16F16

The F16F16 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_F16F16`) requires 4-byte alignment. Each texel occupies 4 bytes in memory and has the following 32-bit representation:



Where R is the 16-bit half precision floating-point red, and G is the 16-bit half precision floating-point green component values.

This would be stored in memory as follows:

The swizzle modes as shown in Table 23 are supported.

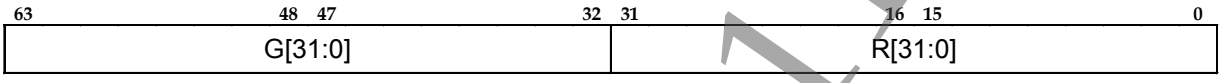
The supported query formats are shown in the following table.

Table 30 F16F16 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE2_00GR SCE_GXM_TEXTURE_SWIZZLE2_GRRR SCE_GXM_TEXTURE_SWIZZLE2_RGGG SCE_GXM_TEXTURE_SWIZZLE2_GRGR SCE_GXM_TEXTURE_SWIZZLE2_00RG	• half4 • float4
SCE_GXM_TEXTURE_SWIZZLE2_GR	• half2 • float2

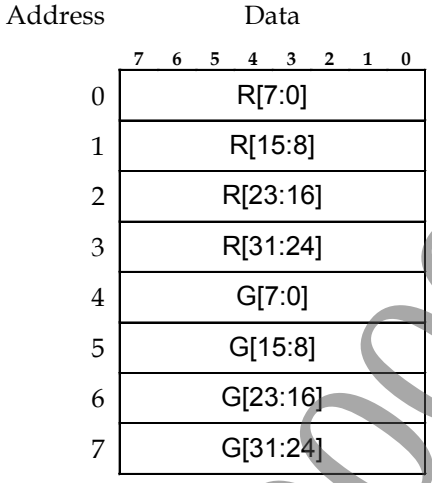
F32F32

The F32F32 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_F32F32) requires 8-byte alignment. Each texel occupies 8 bytes in memory and has the following 64-bit representation.



Where R is the 32-bit floating-point red, and G is the 32-bit floating-point green component values.

This would be stored in memory as follows:



Linear filtering is not supported for this texture format.

The swizzle modes as shown in Table 23 are supported.

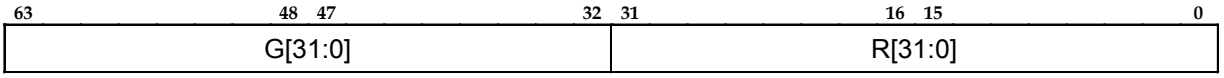
The supported query formats are shown in the following table.

Table 31 F32F32 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE2_00GR SCE_GXM_TEXTURE_SWIZZLE2_GRRR SCE_GXM_TEXTURE_SWIZZLE2_RGGG SCE_GXM_TEXTURE_SWIZZLE2_GRGR SCE_GXM_TEXTURE_SWIZZLE2_00RG	• float4
SCE_GXM_TEXTURE_SWIZZLE2_GR	• float2

U32U32

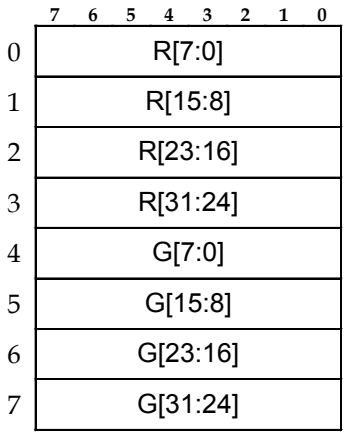
The U32U32 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_U32U32`) requires 8-byte alignment. Each texel occupies 8 bytes in memory and has the following 64-bit representation.



Where R is the unsigned 32-bit red, and G is the unsigned 32-bit green component values.

This would be stored in memory as follows:

Address Data



Linear filtering is not supported for this texture format.

The swizzle modes as shown in Table 23 are supported.

The supported query formats are shown in the following table.

Table 32 U32U32 Query Formats

Swizzle Mode	Supported Query Formats

`SCE_GXM_TEXTURE_SWIZZLE2_00GR`
`SCE_GXM_TEXTURE_SWIZZLE2_GRRR`
`SCE_GXM_TEXTURE_SWIZZLE2_RGGG`
`SCE_GXM_TEXTURE_SWIZZLE2_GRGR`
`SCE_GXM_TEXTURE_SWIZZLE2_00RG`
	• unsigned int4

`SCE_GXM_TEXTURE_SWIZZLE2_GR`	• unsigned int2

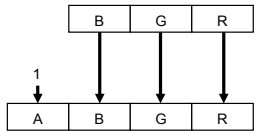
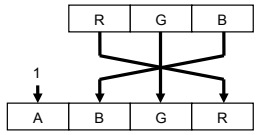
Three Component Base Texture Formats

The three component base texture formats have red and green and blue component values.

Three Component Swizzle Modes

The values from memory are passed to the shader in ABGR component ordering according to the swizzle mode shown in Table 33.

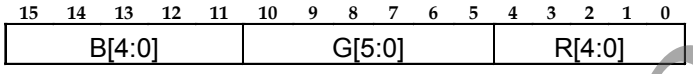
Table 33 Three Component Swizzle Modes

Swizzle Mode	ABGR Representation
SCE_GXM_TEXTURE_SWIZZLE3_BGR	
SCE_GXM_TEXTURE_SWIZZLE3_RGB	

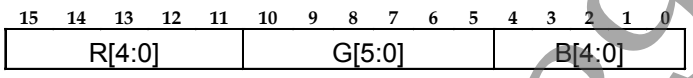
U5U6U5

The U5U6U5 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U5U6U5) requires 4-byte alignment. Each texel occupies 2 bytes in memory and can have the following 16-bit representations:

BGR

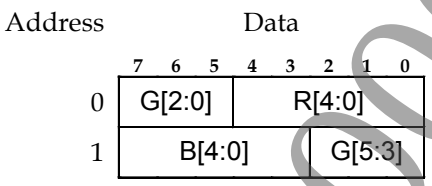


RGB



Where R is the unsigned 5-bit red, G is the unsigned 6-bit green, and B is the unsigned 5-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 33 are supported.

The supported query formats are shown in the following table.

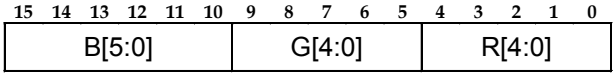
Table 34 U5U6U5 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE3_BGR SCE_GXM_TEXTURE_SWIZZLE3_RGB	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

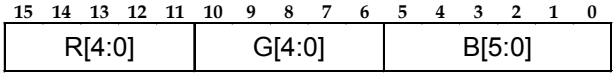
S5S5U6

The S5S5U6 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_S5S5U6) requires 4-byte alignment. Each texel occupies 2 bytes in memory and can have the following 16-bit representations:

BGR

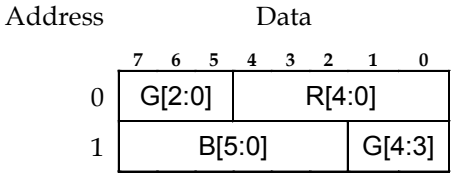


RGB



Where R is the signed 5-bit red, G is the signed 5-bit green, and B is the unsigned 6-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 33 are supported.

The supported query formats are shown in the following table.

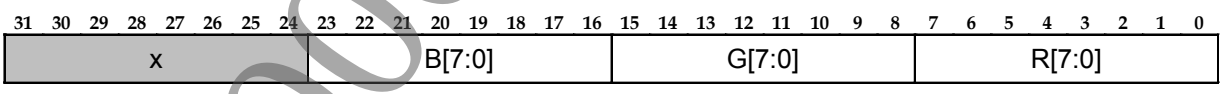
Table 35 S5S5U6 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE3_BGR SCE_GXM_TEXTURE_SWIZZLE3_RGB	• half4 • float4

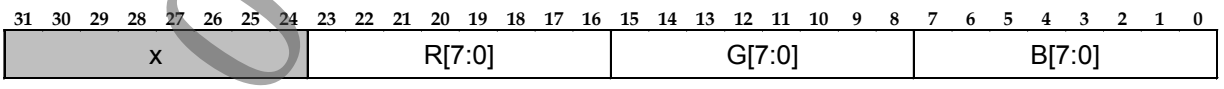
X8S8S8U8

The X8S8S8U8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_X8S8S8U8) requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations.

BGR



RGB



Where R is the unsigned 8-bit red, G is the signed 8-bit green, and B is the signed 8-bit blue component values. The value x is ignored.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:

The swizzle modes as shown in Table 33 are supported.

The supported query formats are shown in the following table.

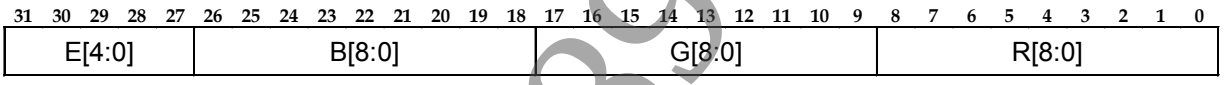
Table 36 X8S8S8U8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE3_BGR SCE_GXM_TEXTURE_SWIZZLE3_RGB	• half4 • float4

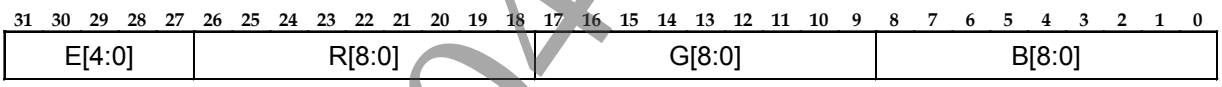
SE5M9M9M9

The SE5M9M9M9 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_SE5M9M9M9) requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations:

BGR

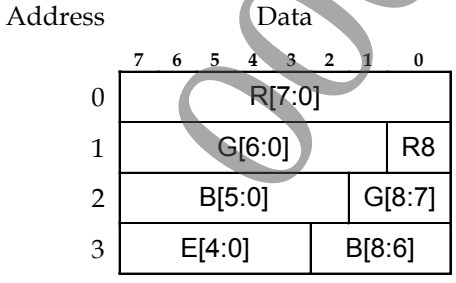


RGB



Where E is the shared 5 bit exponent, R is the partial-precision floating-point 9-bit red mantissa, G is the partial-precision floating-point 9-bit green mantissa, and B is the partial-precision floating-point 9-bit blue mantissa values. There is no sign bit.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 33 are supported.

The supported query formats are shown in the following table.

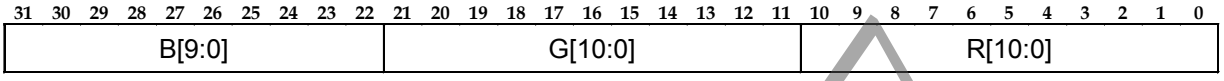
Table 37 SE5M9M9M9 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE3_BGR SCE_GXM_TEXTURE_SWIZZLE3_RGB	• half4 • float4

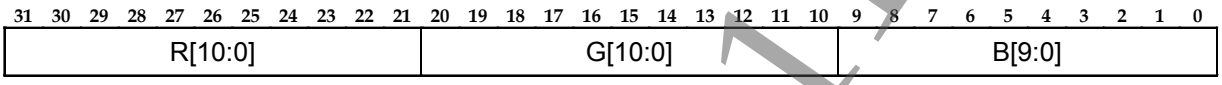
F11F11F10

The F11F11F10 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_F11F11F10) requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations.

BGR

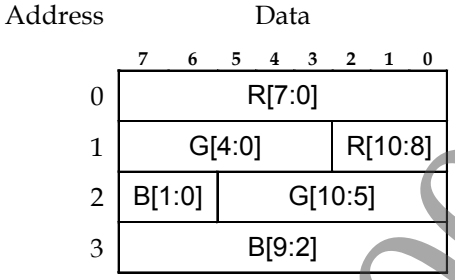


RGB

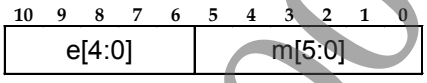


Where R is the partial-precision floating-point 11-bit red, G is the partial-precision floating-point 11-bit green, and B is the partial-precision floating-point 10-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:

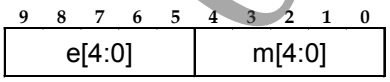


The 11-bit red and green values are interpreted as:



Where e is the 5-bit exponent and m is the 6-bit mantissa. There is no sign bit.

The 10-bit blue value is interpreted as:



Where e is the 5-bit exponent and m is the 5-bit mantissa. There is no sign bit.

The swizzle modes as shown in Table 33 are supported.

The supported query formats are shown in the following table.

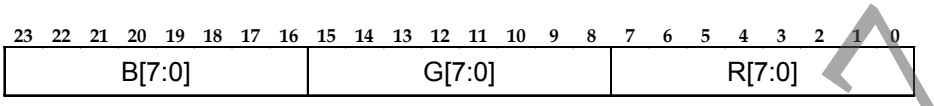
Table 38 F11F11F10 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE3_BGR SCE_GXM_TEXTURE_SWIZZLE3_RGB	• half4 • float4

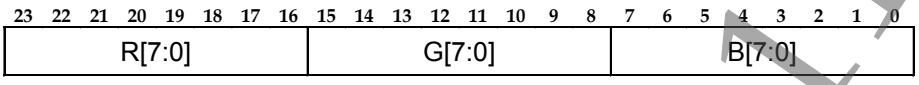
U8U8U8

The U8U8U8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U8U8U8) requires 4-byte alignment. This is a packed texture format, each texel occupies 3 bytes in memory and can have the following 24-bit representations.

BGR

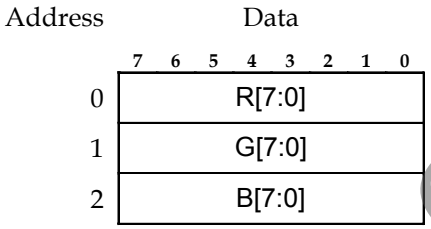


RGB



Where R is the unsigned 8-bit red, G is the unsigned 8-bit green, and B is the unsigned 8-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 33 are supported.

The supported query formats are shown in the following table.

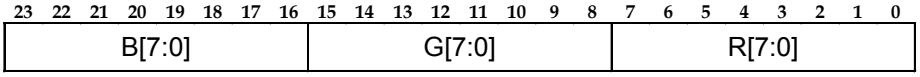
Table 39 U8U8U8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE3_BGR SCE_GXM_TEXTURE_SWIZZLE3_RGB	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

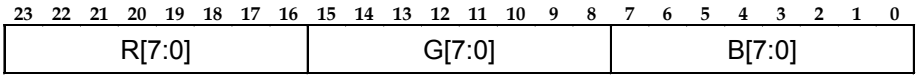
S8S8S8

The S8S8S8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_S8S8S8) requires 4-byte alignment. This is a packed texture format, each texel occupies 3 bytes in memory and can have the following 24-bit representations.

BGR



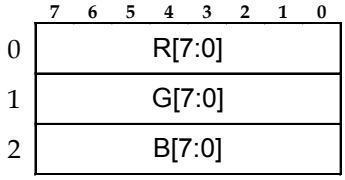
RGB



Where R is the signed 8-bit red, G is the signed 8-bit green, and B is the signed 8-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:

Address Data



The swizzle modes as shown in Table 33 are supported.

The supported query formats are shown in the following table.

Table 40 S8S8S8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE3_BGR SCE_GXM_TEXTURE_SWIZZLE3_RGB	• signed char4 • half4 • float4

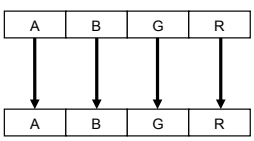
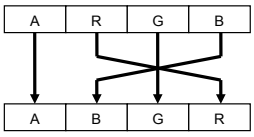
Four Component Base Texture Formats

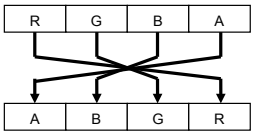
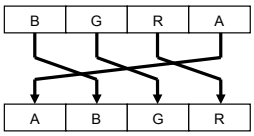
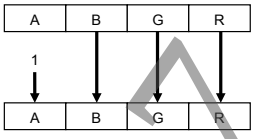
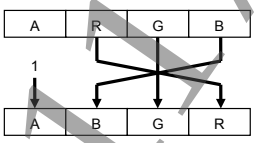
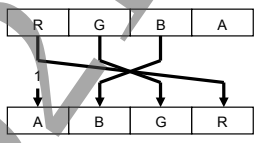
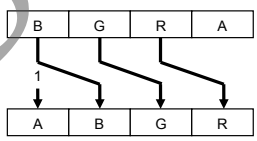
The four component base texture formats have alpha, red and green and blue component values.

Four Component Swizzle Modes

Unless otherwise documented in the texture format description, the values from memory are passed to the shader in ABGR component ordering according to the swizzle mode shown in Table 41.

Table 41 Four Component Swizzle Modes

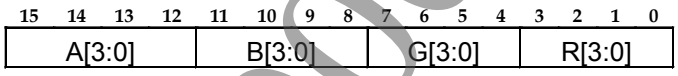
Swizzle Mode	ABGR Representation
SCE_GXM_TEXTURE_SWIZZLE4_ABGR	
SCE_GXM_TEXTURE_SWIZZLE4_ARGB	

Swizzle Mode	ABGR Representation
SCE_GXM_TEXTURE_SWIZZLE4_RGBA	
SCE_GXM_TEXTURE_SWIZZLE4_BGRA	
SCE_GXM_TEXTURE_SWIZZLE4_1BGR	
SCE_GXM_TEXTURE_SWIZZLE4_1RGB	
SCE_GXM_TEXTURE_SWIZZLE4_RGB1	
SCE_GXM_TEXTURE_SWIZZLE4_BGR1	

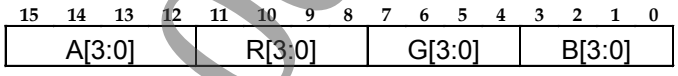
U4U4U4U4

The U4U4U4U4 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U4U4U4U4) requires 4-byte alignment. Each texel occupies 2 bytes in memory and can have the following 16-bit representations.

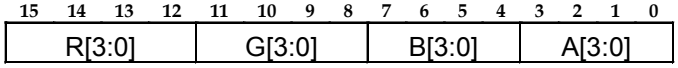
ABGR



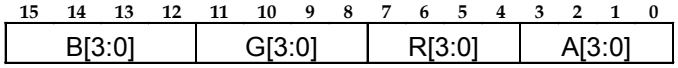
ARGB



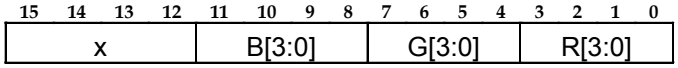
RGBA



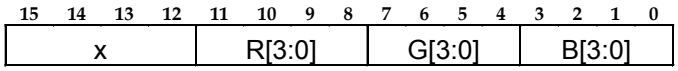
BGRA



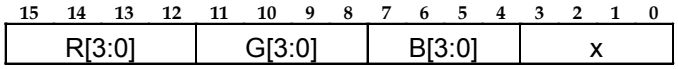
1BGR



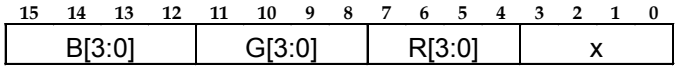
1RGB



RGB1



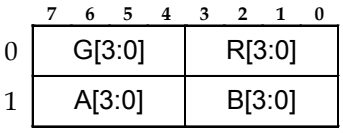
BGR1



Where A is interpreted as the unsigned 4-bit alpha, R is the unsigned 4-bit red, G is the unsigned 4-bit green, and B is the unsigned 4-bit blue component values. The value x is ignored.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

Address Data



The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

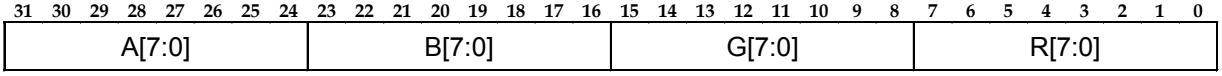
Table 42 U4U4U4U4 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_RGBA SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_RGB1 SCE_GXM_TEXTURE_SWIZZLE4_BGR1	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

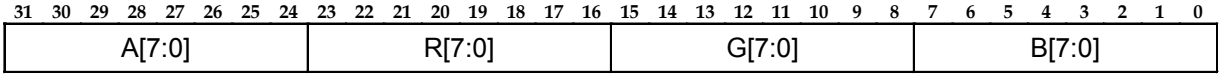
U8U8U8U8

The U8U8U8U8 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_U8U8U8U8`) requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations.

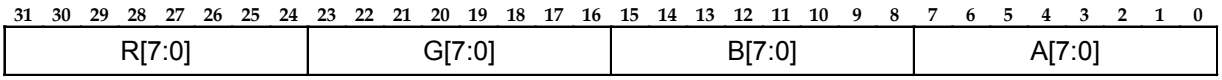
ABGR



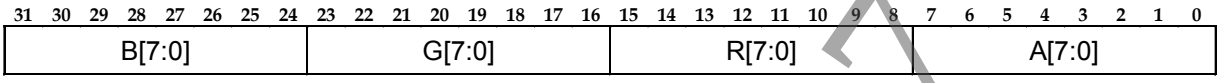
ARGB



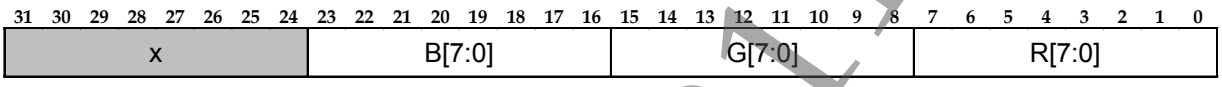
RGBA



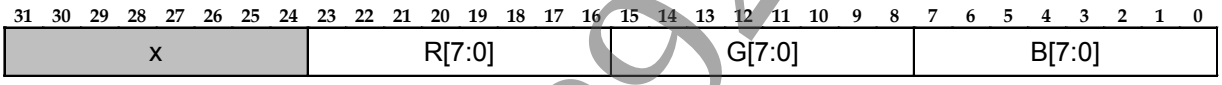
BGRA



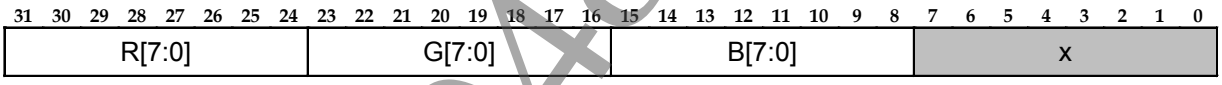
1BGR



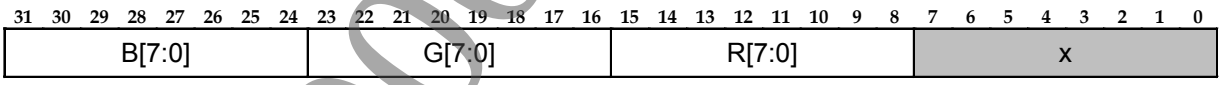
1RGB



RGB1

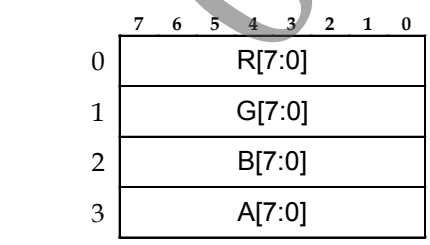


BGR1



Where A is interpreted as the unsigned 8-bit alpha, R is the unsigned 8-bit red, G is the unsigned 8-bit green, and B is the unsigned 8-bit blue component values. The value x is ignored.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:



The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

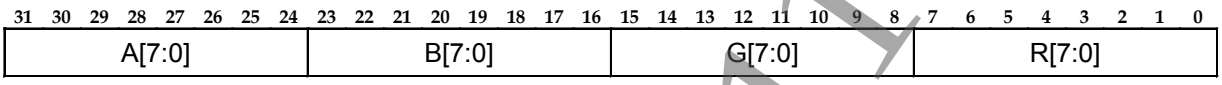
Table 43 U8U8U8U8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_RGBA SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_RGB1 SCE_GXM_TEXTURE_SWIZZLE4_BGR1	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

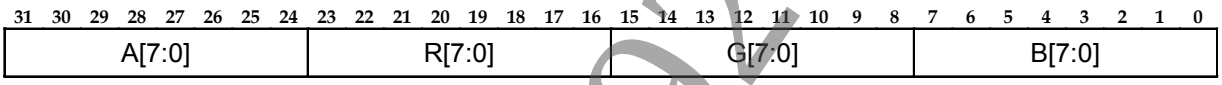
S8S8S8S8

The S8S8S8S8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_S8S8S8S8) requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations.

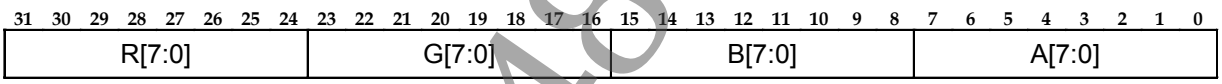
ABGR



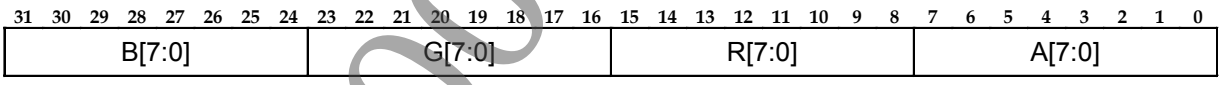
ARGB



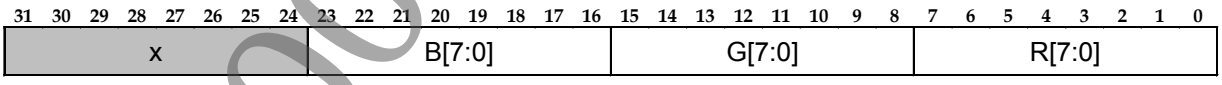
RGBA



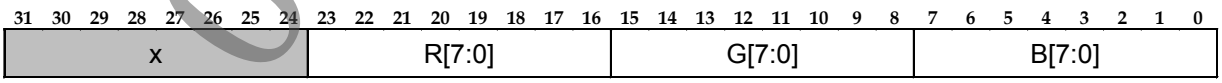
BGRA



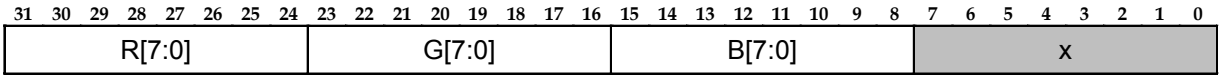
1BGR



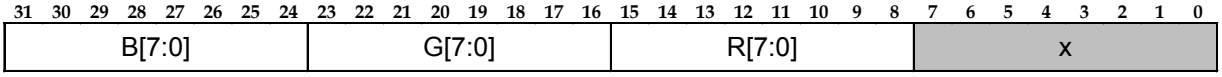
1RGB



RGB1



BGR1



Where A is interpreted as the signed 8-bit alpha, R is the signed 8-bit red, G is the signed 8-bit green, and B is the signed 8-bit blue component values. The value x is ignored.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

The swizzle modes as shown in Table 41 are supported.

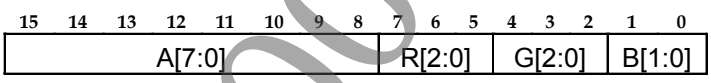
The supported query formats are shown in the following table.

Table 44 S8S8S8S8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_RGBA SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_RGB1 SCE_GXM_TEXTURE_SWIZZLE4_BGR1	• signed char4 • half4 • float4

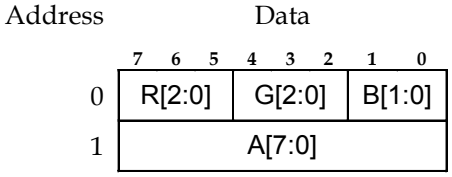
U8U3U3U2

The U8U3U3U2 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_U8U3U3U2`) requires 4-byte alignment. Each texel occupies 2 bytes in memory and has the following 16-bit representation.



Where A is interpreted as the unsigned 8-bit alpha, R is the unsigned 3-bit red, G is the unsigned 3-bit green, and B is the unsigned 2-bit blue component values.

This would be stored in memory as follows:



The U8U3U3U2 format does not support swizzle mode and always uses this fixed bit layout. The following query formats are supported:

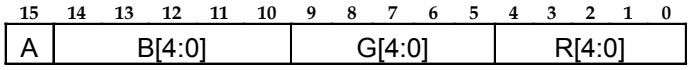
Table 45 U8U3U3U2 Memory to Result Format Mode

Supported Query Formats
• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

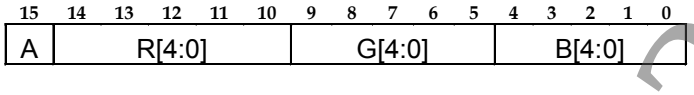
U1U5U5U5

The U1U5U5U5 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_U1U5U5U5`) requires 4-byte alignment. Each texel occupies 2 bytes in memory and can have the following 16-bit representations:

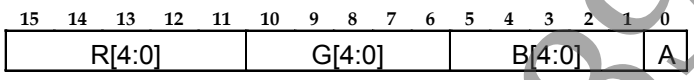
ABGR



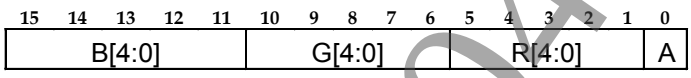
ARGB



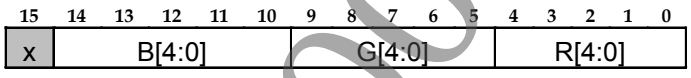
RGBA



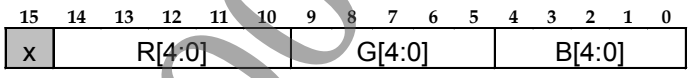
BGRA



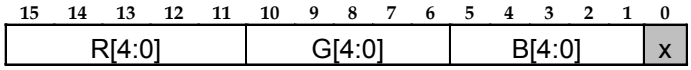
1BGR



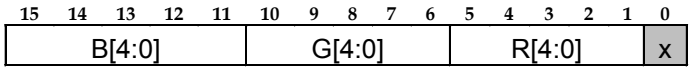
1RGB



RGB1

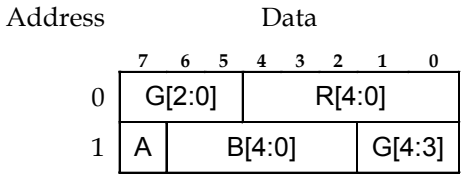


BGR1



Where A is interpreted as the unsigned 1-bit alpha, R is the unsigned 5-bit red, G is the unsigned 5-bit green, and B is the unsigned 5-bit blue component values. The value x is ignored, and alpha is assigned the value 1.0 in this case.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:



The following swizzle modes are supported.

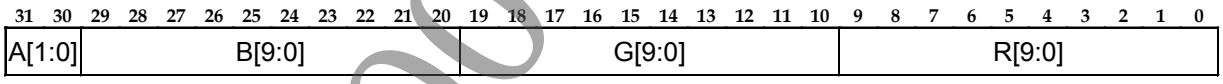
Table 46 U1U5U5U5 Swizzle Modes and Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_RGBA SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_RGB1 SCE_GXM_TEXTURE_SWIZZLE4_BGR1	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

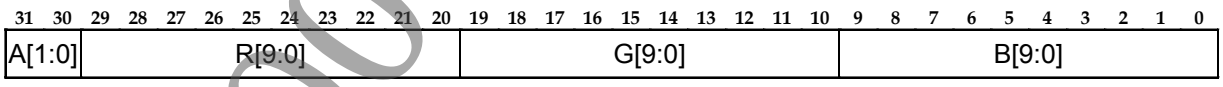
U2U10U10U10

The U2U10U10U10 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_U2U10U10U10`) requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations depending on the swizzle mode:

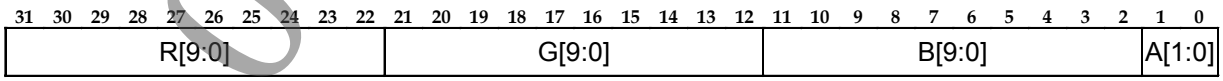
ABGR



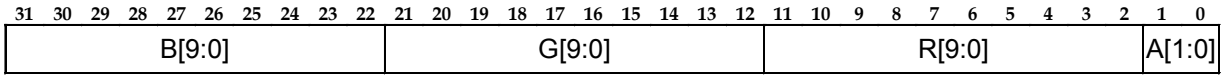
ARGB



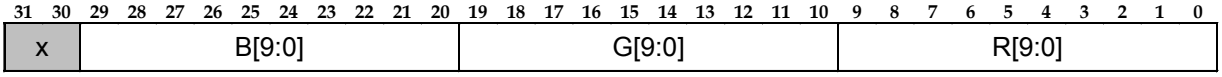
RGBA



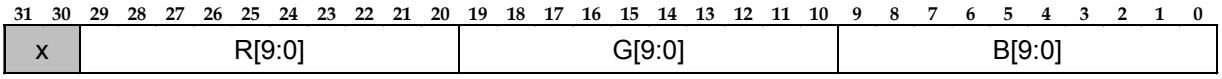
BGRA



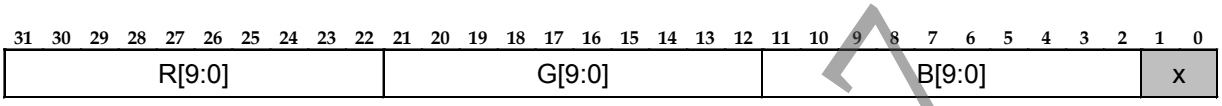
1BGR



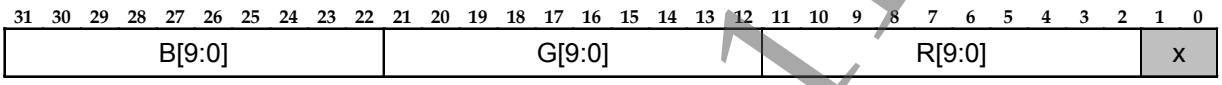
1RGB



RGB1



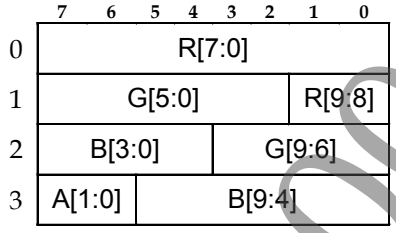
BGR1



Where A is interpreted as the unsigned 2-bit alpha, R is the unsigned 10-bit red, G is the unsigned 10-bit green, and B is the unsigned 10-bit blue component values. The value x is ignored, and alpha is assigned the value 1.0 in this case.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

Address Data



The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

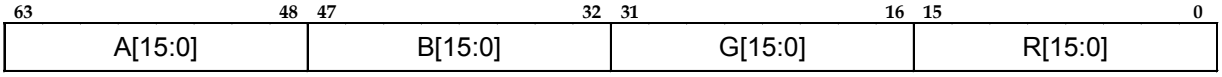
Table 47 U2U10U10U10 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_BRGA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_BGR1 SCE_GXM_TEXTURE_SWIZZLE4_BRG1	• half4 • float4

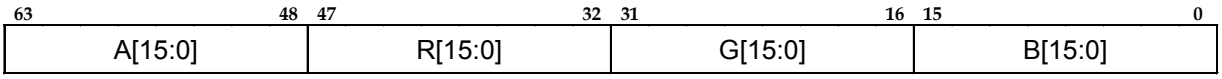
U16U16U16U16

The U16U16U16U16 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_U16U16U16U16`) requires 8-byte alignment. Each texel occupies 8 bytes in memory and can have the following 64-bit representations.

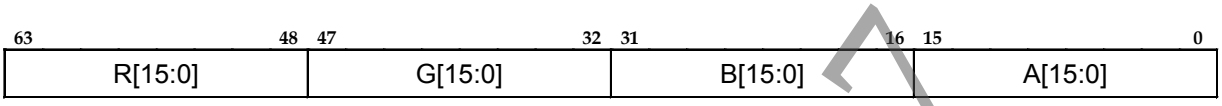
ABGR



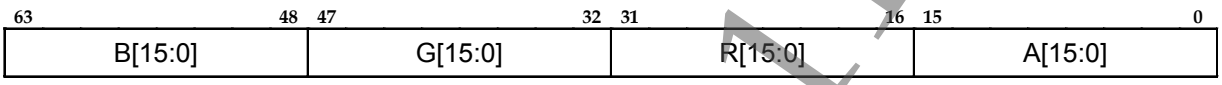
ARGB



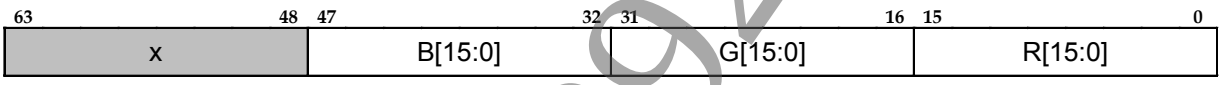
RGBA



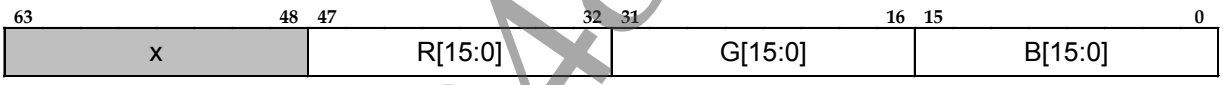
BGRA



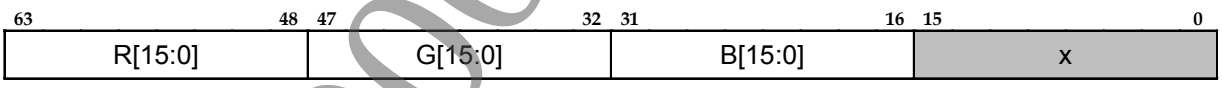
1BGR



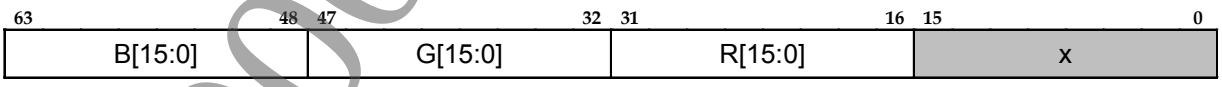
1RGB



RGB1



BGR1



Where A is interpreted as the unsigned 16-bit alpha, R is the unsigned 16-bit red, G is the unsigned 16-bit green, and B is the unsigned 16-bit blue component values. The value x is ignored, and alpha is assigned the value 1.0 in this case.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

Address	Data (7 6 5 4 3 2 1 0)
0	R[7:0]
1	R[15:8]
2	G[7:0]
3	G[15:8]
4	B[7:0]
5	B[15:8]
6	A[7:0]
7	A[15:8]

The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

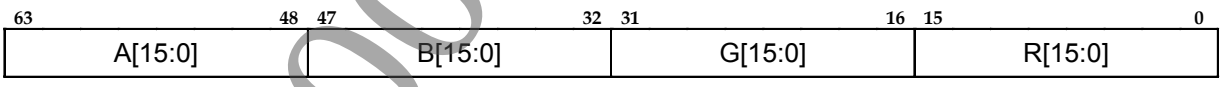
Table 48 U16U16U16U16 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_BRGA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_BGR1 SCE_GXM_TEXTURE_SWIZZLE4_BRG1	• unsigned short4 • float4

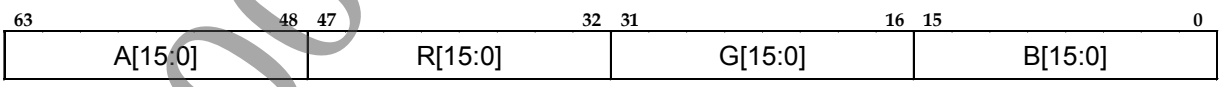
S16S16S16S16

The S16S16S16S16 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_S16S16S16S16`) requires 8-byte alignment. Each texel occupies 8 bytes in memory and can have the following 64-bit representations.

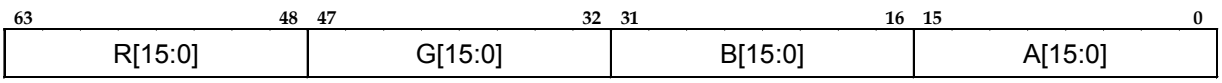
ABGR



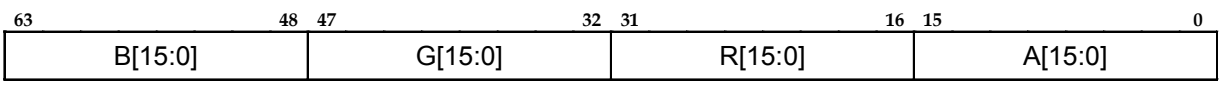
ARGB



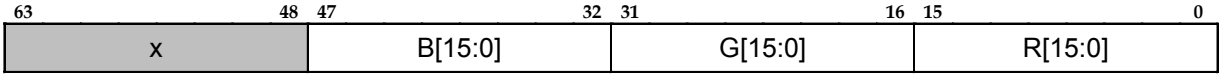
RGBA



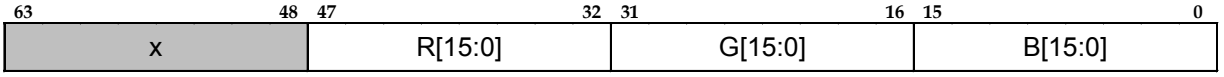
BGRA



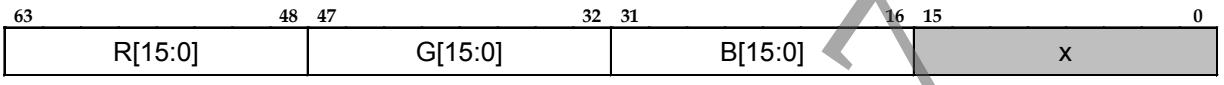
1BGR



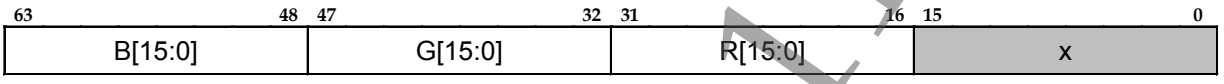
1RGB



RGB1



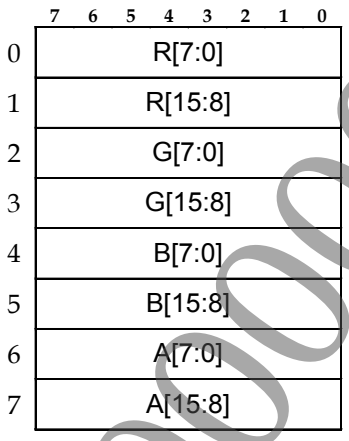
BGR1



Where A is interpreted as the signed 16-bit alpha, R is the signed 16-bit red, G is the signed 16-bit green, and B is the signed 16-bit blue component values. The value x is ignored, and alpha is assigned the value 1.0 in this case.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

Address Data



The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

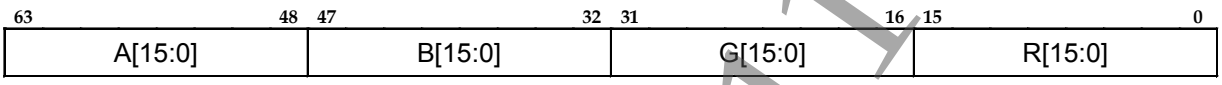
Table 49 S16S16S16S16 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_BRGA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_BGR1 SCE_GXM_TEXTURE_SWIZZLE4_BRG1	• signed short4 • float4

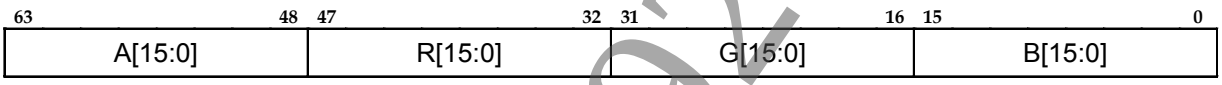
F16F16F16F16

The F16F16F16F16 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_F16F16F16F16) requires 8-byte alignment. Each texel occupies 8 bytes in memory and can have the following 64-bit representations.

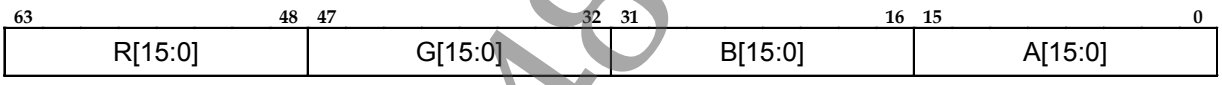
ABGR



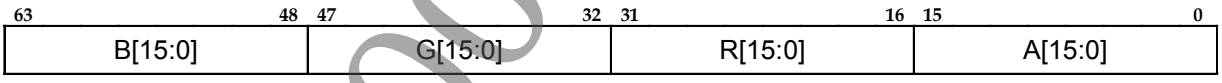
ARGB



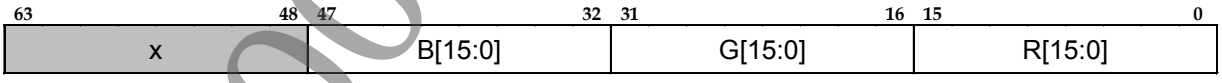
RGBA



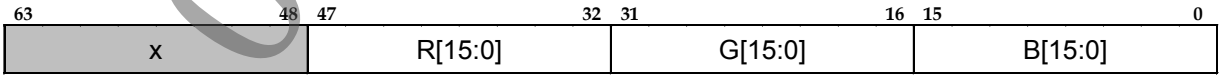
BGRA



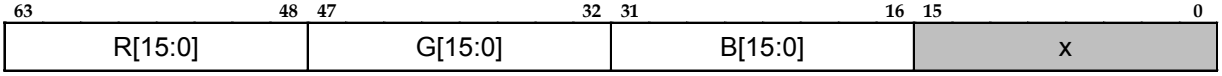
1BGR



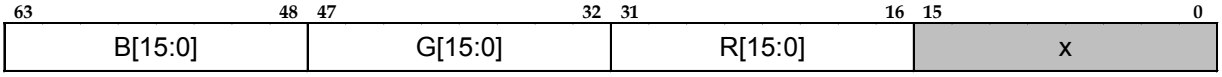
1RGB



RGB1

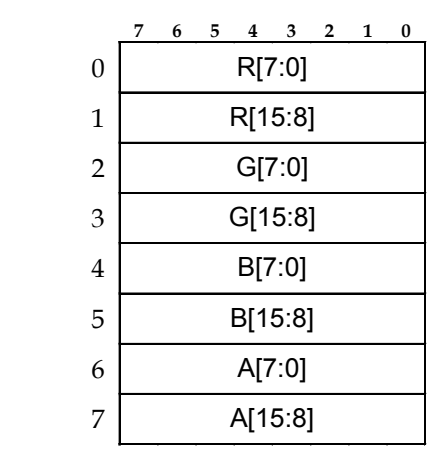


BGR1



Where A is interpreted as the 16-bit half precision floating-point alpha, R is the 16-bit half precision floating-point red, G is the 16-bit half precision floating-point green, and B is the 16-bit half precision floating-point blue component values. The value x is ignored, and alpha is assigned the value 1.0 in this case.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:



The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

Table 50 F16F16F16F16 Query Formats

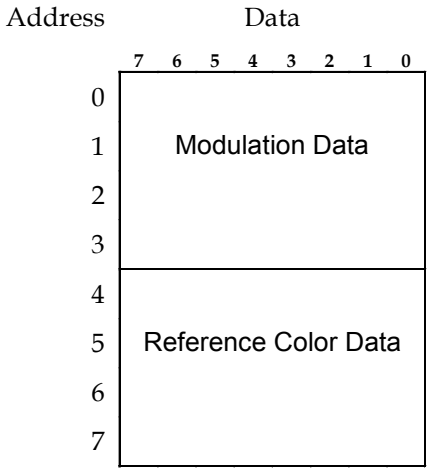
Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_BRGA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_BGR1 SCE_GXM_TEXTURE_SWIZZLE4_BRG1	• half4 • float4

PVRT2BPP

The PVRT2BPP texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_PVRT2BPP`) is a PVRT-I compressed format. For further information on this compressed texture format see the section "Appendix B: Compressed Texture Formats".

> **Note:** For further information on PVRT-I see http://web.onetel.net.uk/~simonnihal/assorted3d/fenney03texcomp.pdf

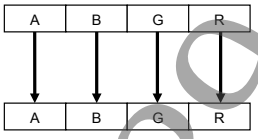
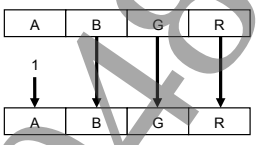
The PVRT2BPP format requires 8-byte alignment. Each 8x4 pixel block occupies 8 bytes in memory and has the following representation in memory:



For further information on the PVRT2BPP format see PVRT2BPP compressed texture format.

The following swizzle modes are supported.

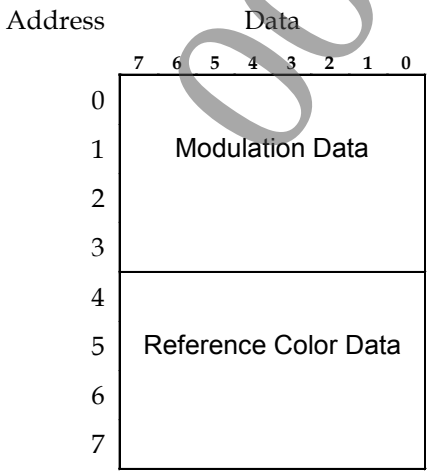
Table 51 PVRT2BPP Memory Swizzle Modes

Swizzle Mode	ABGR Representation	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR		• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
SCE_GXM_TEXTURE_SWIZZLE4_1BGR		

PVRT4BPP

The PVRT4BPP texture format (SCE_GXM_TEXTURE_BASE_FORMAT_PVRT4BPP) is a PVRT-I compressed format. For further information on this compressed texture format see the section "Appendix B: Compressed Texture Formats".

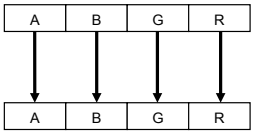
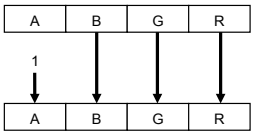
The PVRT4BPP format requires 8-byte alignment. Each 4x4 pixel block occupies 8 bytes in memory and has the following representation in memory:



For further information on the PVRT4BPP format see PVRT4BPP compressed texture format.

The following swizzle modes are supported.

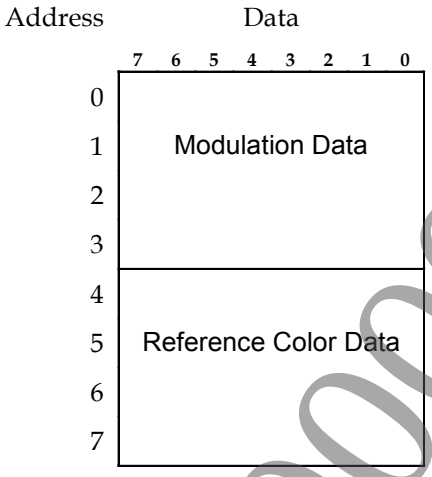
Table 52 PVRT4BPP Memory Swizzle Modes

Swizzle Mode	ABGR Representation	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR		• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
SCE_GXM_TEXTURE_SWIZZLE4_1BGR		

PVRTII2BPP

The PVRTII2BPP texture format (SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII2BPP) is a PVRT-II compressed format. For further information on this compressed texture format see the section "Appendix B: Compressed Texture Formats".

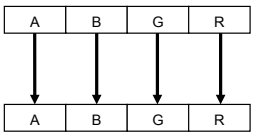
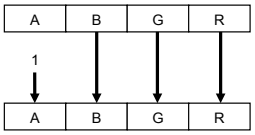
The PVRTII2BPP format requires 8-byte alignment. Each 8x4 pixel block occupies 8 bytes in memory and has the following representation in memory:



For further information on the PVRTII2BPP format see PVRTII2BPP compressed texture format.

The following swizzle modes are supported.

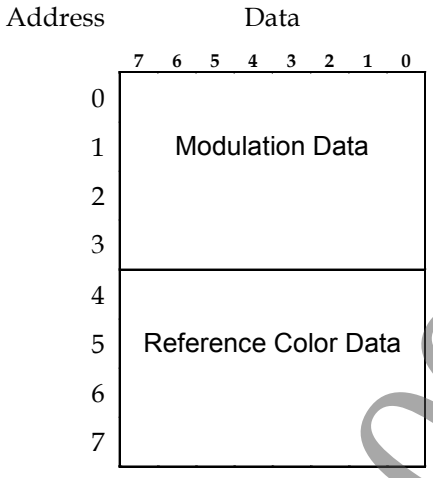
Table 53 PVRTII2BPP Memory Swizzle Modes

Swizzle Mode	ABGR Representation	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR		• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
SCE_GXM_TEXTURE_SWIZZLE4_1BGR		

PVRTII4BPP

The PVRTII4BPP texture format (SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII4BPP) is a PVRT-II compressed format. For further information on this compressed texture format see the section "Appendix B: Compressed Texture Formats".

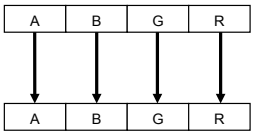
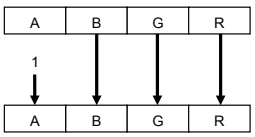
The PVRT4BPP format requires 8-byte alignment. Each 4x4 pixel block occupies 8 bytes in memory and has the following representation in memory:



For further information on the PVRTII4BPP format see PVRTII4BPP compressed texture format.

The following swizzle modes are supported.

Table 54 PVRTII4BPP Memory Swizzle Modes

Swizzle Mode	ABGR Representation	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR		• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
SCE_GXM_TEXTURE_SWIZZLE4_1BGR		

UBC1

The UBC1 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_UBC1`) is the S3 DXT1 texture compression format.

The UBC1 format requires 8-byte alignment. Each 4x4 pixel block occupies 8 bytes in memory and has the following representation in memory:

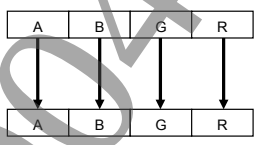
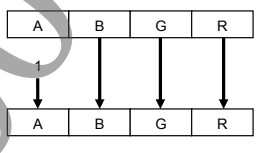
Address Data

	7	6	5	4	3	2	1	0
0	C0[7:0]							
1	C0[15:8]							
2	C1[7:0]							
3	C1[15:8]							
4	v03			v02			v01	v00
5	v13			v12			v11	v10
6	v23			v22			v21	v20
7	v33			v32			v31	v30

Where C0 and C1 are the minimum and maximum reference colors, and the block v00-v33 is the 16 2-bit color indices.

The supported swizzle modes and result formats are shown in the following table.

Table 55 UBC1 Swizzle Modes and Result Formats

Swizzle Mode	ABGR Representation	Supported Query Formats
`SCE_GXM_TEXTURE_SWIZZLE4_ABGR`		• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
`SCE_GXM_TEXTURE_SWIZZLE4_1BGR`		

UBC2

The UBC2 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_UBC2`) is the S3 DXT3 texture compression format.

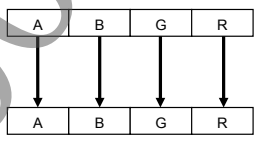
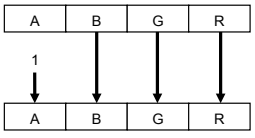
The UBC2 format requires 16-byte alignment. Each 4x4 pixel block occupies 16 bytes in memory and has the following representation in memory:

Address	Data			
	7 6 5 4	3 2 1 0		
0	a01		a00	
1	a03		a02	
2	a11		a10	
3	a13		a12	
4	a21		a20	
5	a23		a22	
6	a31		a30	
7	a33		a32	
8	C0[7:0]			
9	C0[15:8]			
10	C1[7:0]			
11	C1[15:8]			
12	v03	v02	v01	v00
13	v13	v12	v11	v10
14	v23	v22	v21	v20
15	v33	v32	v31	v30

Where the block a00a33 is the 4-bit alpha values, C0 and C1 are the minimum and maximum reference colors, and the block v00-v33 is the 16 2-bit color indices.

The supported swizzle modes and result formats are shown in the following table.

Table 56 UBC2 Swizzle Modes and Result Formats

Swizzle Mode	ABGR Representation	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR		• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
SCE_GXM_TEXTURE_SWIZZLE4_1BGR		

UBC3

The UBC3 texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_UBC3`) is the S3 DXT5 texture compression format.

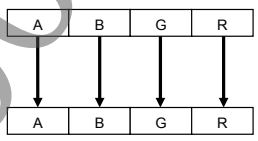
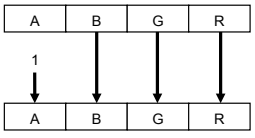
The UBC3 format requires 16-byte alignment. Each 4x4 pixel block occupies 16 bytes in memory and has the following representation in memory:

Address	Data							
	7	6	5	4	3	2	1	0
0	A0[7:0]							
1	A1[7:0]							
2	α00-α33							
3								
4								
5								
6								
7								
8	C0[7:0]							
9	C0[15:8]							
10	C1[7:0]							
11	C1[15:8]							
12	v03	v02	v01	v00				
13	v13	v12	v11	v10				
14	v23	v22	v21	v20				
15	v33	v32	v31	v30				

Where A0 and A1 are the reference alpha values, the block α00-α33 is the 16 3-bit alpha indices, C0 and C1 are the minimum and maximum reference colors, and the block v00-v33 is the 16 2-bit color indices.

The supported swizzle modes and result formats are shown in the following table.

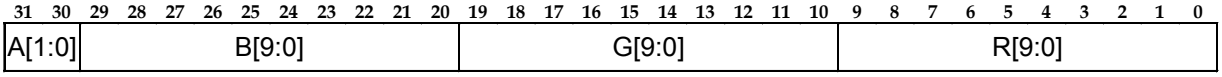
Table 57 UBC3 Swizzle Modes and Result Formats

Swizzle Mode	ABGR Representation	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR		• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
SCE_GXM_TEXTURE_SWIZZLE4_1BGR		

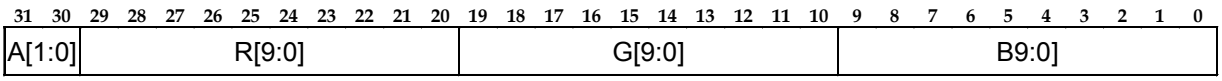
U2F10F10F10

The U2F10F10F10 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_U2F10F10F10) format requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations.

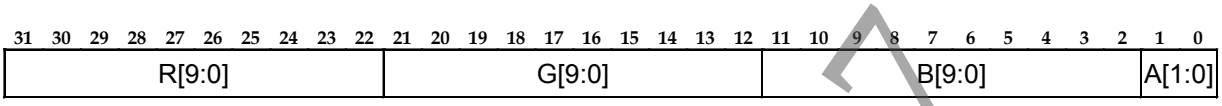
ABGR



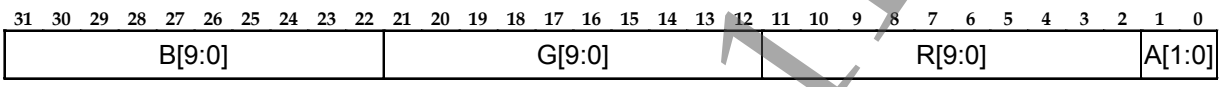
ARGB



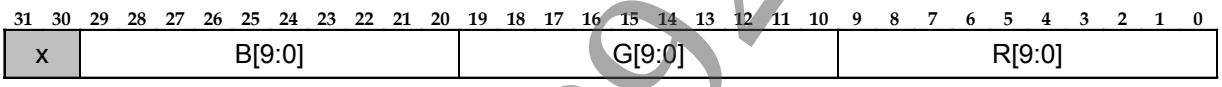
RGBA



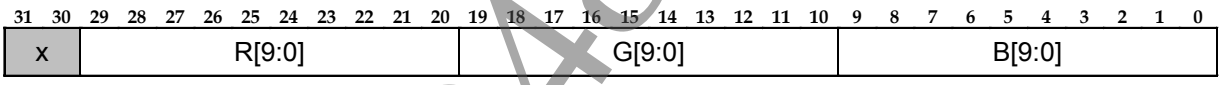
BGRA



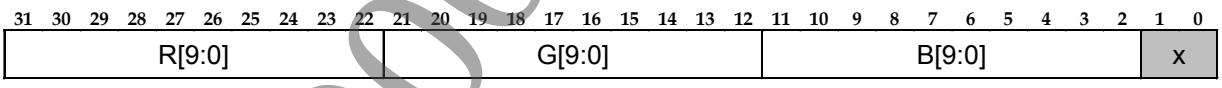
1BGR



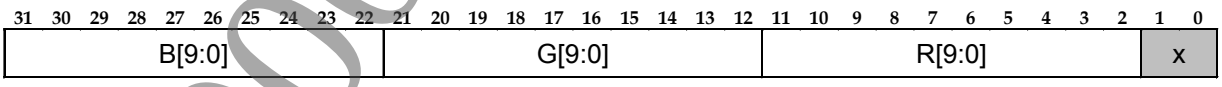
1RGB



RGB1



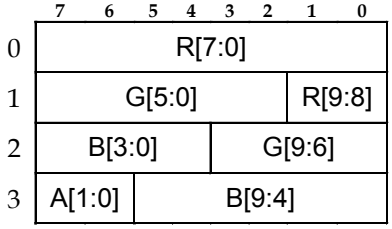
BGR1



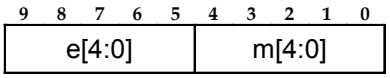
Where R is the partial-precision floating-point 10-bit red, G is the partial-precision floating-point 10-bit green, and B is the partial-precision floating-point 10-bit blue component values. A is the unsigned 2-bit alpha value. The value x is ignored, and alpha is assigned the value 1.0 in this case.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

Address Data



The 10-bit red, green and blue values are interpreted as:



Where e is the 5-bit exponent and m is the 5-bit mantissa. There is no sign bit.

The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

Table 58 U2F10F10F10 Query Formats

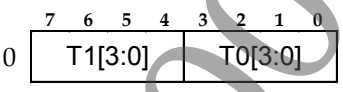
Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_BRGA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_BGR1 SCE_GXM_TEXTURE_SWIZZLE4_BRG1	• half4 • float4

P4

The P4 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_P4) requires 4-byte alignment. This is a palettized format where each texel from memory is an index into a color palette.

The texels in memory will be stored:

Address Data



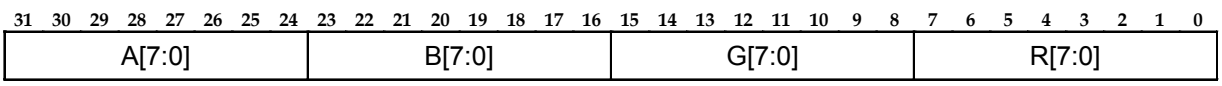
Where each texel Tn is a 4-bit index.

P4 uses a 4-bit index providing a 16-entry palette of 32-bit values.

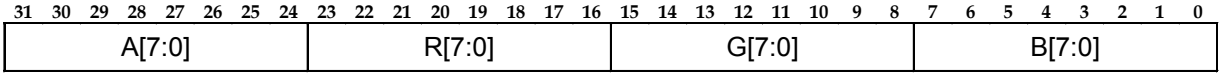
The palette should be aligned to 64 bytes.

The palette can have the following 32-bit representations.

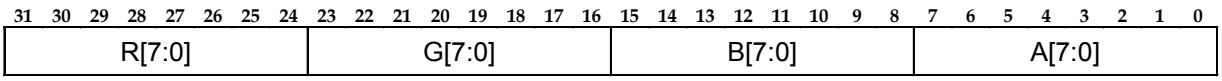
ABGR



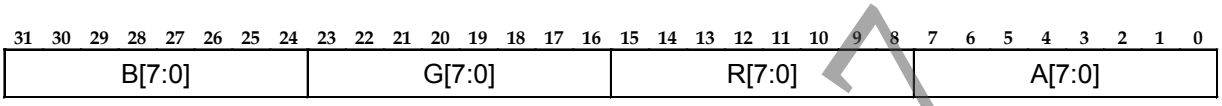
ARGB



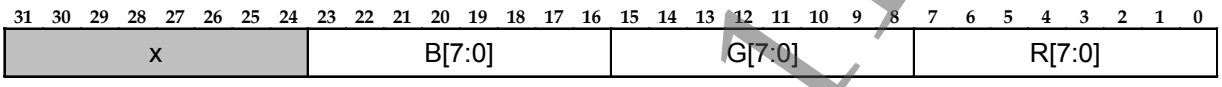
RGBA



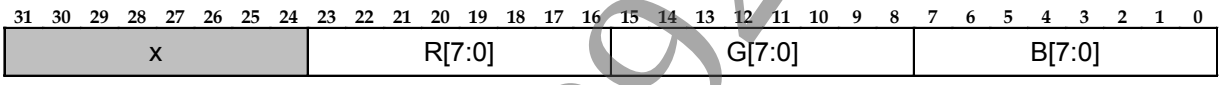
BGRA



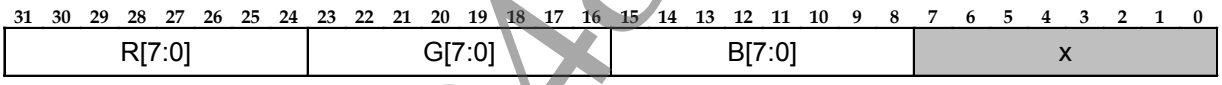
1BGR



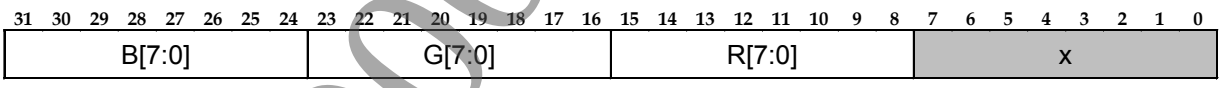
1RGB



RGB1

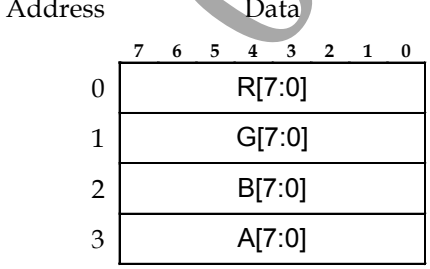


BGR1



Where A is interpreted as the unsigned 8-bit alpha, R is the unsigned 8-bit red, G is the unsigned 8-bit green, and B is the unsigned 8-bit blue component values. The value x is ignored, and alpha is assigned the value 1.0 in this case.

All of these palette representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:



The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

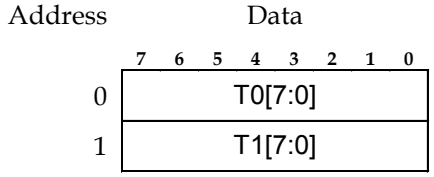
Table 59 P4 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_RGBA SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_RGB1 SCE_GXM_TEXTURE_SWIZZLE4_BGR1	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

P8

The P8 texture format (SCE_GXM_TEXTURE_BASE_FORMAT_P8) requires 4-byte alignment. This is a palettized format where each texel from memory is an index into a color palette.

The texels in memory will be stored:



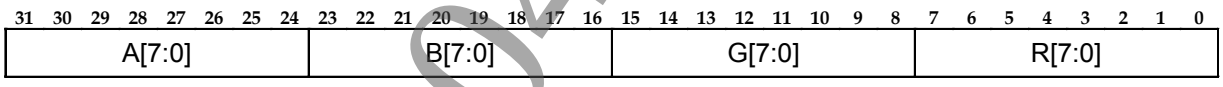
Where each texel Tn is a 8-bit index.

P8 uses a 8-bit index providing a 256-entry palette of 32-bit values.

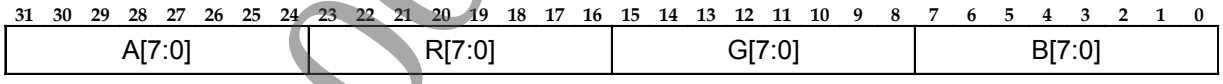
The palette should be aligned to 64 bytes.

The palette can have the following 32-bit representations.

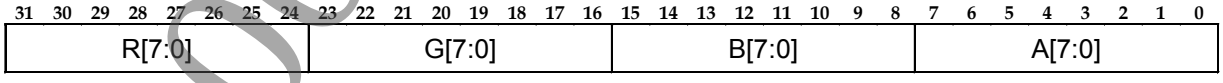
ABGR



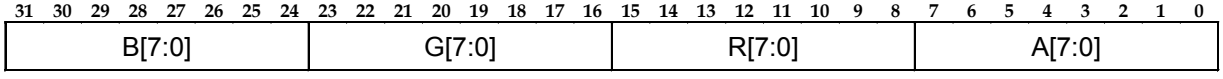
ARGB



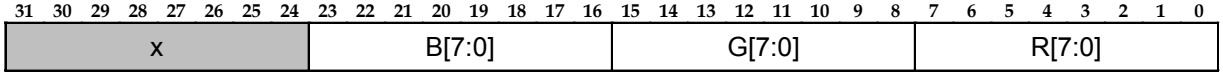
RGBA



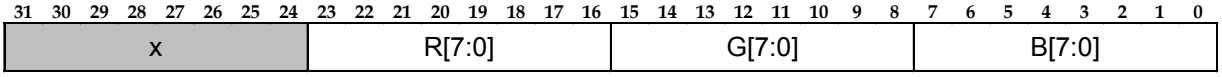
BGRA



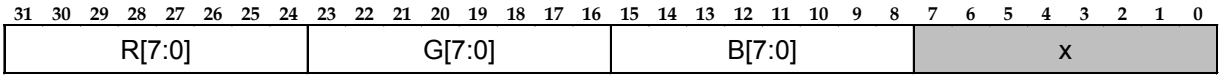
1BGR



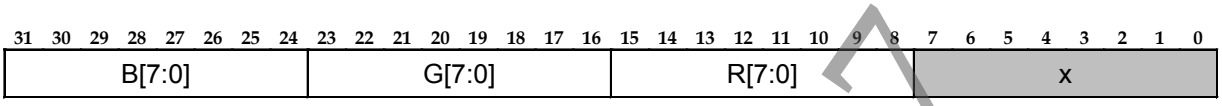
1RGB



RGB1



BGR1



Where A is interpreted as the unsigned 8-bit alpha, R is the unsigned 8-bit red, G is the unsigned 8-bit green, and B is the unsigned 8-bit blue component values. The value x is ignored.

All of these palette representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

The swizzle modes as shown in Table 41 are supported.

The supported query formats are shown in the following table.

Table 60 P8 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE4_ABGR SCE_GXM_TEXTURE_SWIZZLE4_ARGB SCE_GXM_TEXTURE_SWIZZLE4_RGBA SCE_GXM_TEXTURE_SWIZZLE4_BGRA SCE_GXM_TEXTURE_SWIZZLE4_1BGR SCE_GXM_TEXTURE_SWIZZLE4_1RGB SCE_GXM_TEXTURE_SWIZZLE4_RGB1 SCE_GXM_TEXTURE_SWIZZLE4_BGR1	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected. • float4 - When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.

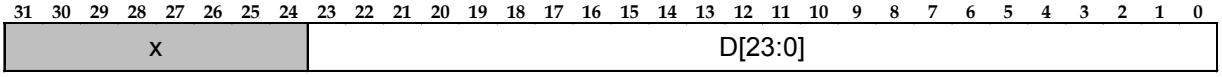
Depth and Stencil Texture Formats

The depth and stencil base texture formats only have depth and stencil values.

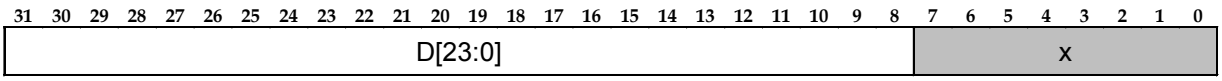
X8U24

The X8U24 depth and stencil texture format (`SCE_GXM_TEXTURE_BASE_FORMAT_X8U24`) requires 4-byte alignment. Each texel occupies 4 bytes in memory and can have the following 32-bit representations.

SD

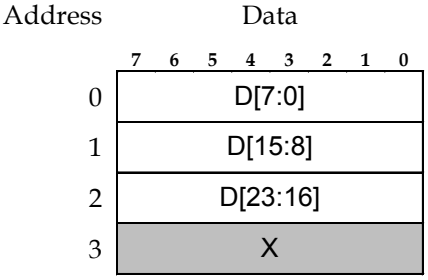


DS



Where D is the unsigned 24-bit depth value. The value x is ignored.

All of these representations are stored little-endian in memory. For example, the **SD** representation would be stored as:



The GPU only supports linear filtering of this format when used with a shadow map compare in shader code. Linear filtering of the raw texture data is not supported.

The supported swizzle modes and query formats are shown in the following table. When the D value is read as an unsigned integer, the data is returned in the low 24 bits with zeros in the upper 8 bits.

Table 61 X8U24 Swizzle Modes and Query Formats

Swizzle Mode	Supported Query Formats
`SCE_GXM_TEXTURE_SWIZZLE2_SD` `SCE_GXM_TEXTURE_SWIZZLE2_DS`	• unsigned int • float

YUV Texture Formats

YUV textures are converted into RGB during the texture load. The GPU supports two active color space conversion matrices, labeled CSC0 and CSC1, which are selected as part of the swizzle.

The default profiles for CSC0 and CSC1 are as follows:

- CSC0 defaults to `SCE_GXM_YUV_PROFILE_BT601_STANDARD: ITU-R BT.601` specification (usually applied to Standard Definition broadcasting)
- CSC1 defaults to `SCE_GXM_YUV_PROFILE_BT709_STANDARD: ITU-R BT.709` specification (associated with High Definition broadcasting)

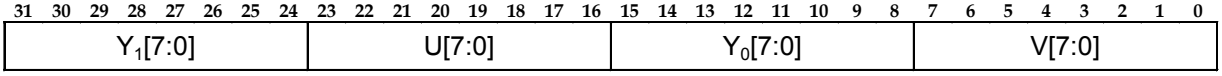
Interleaved YUV textures must be of `SCE_GXM_TEXTURE_LINEAR` or `SCE_GXM_TEXTURE_LINEAR_STRIDED` memory layout. Planar YUV textures must be of `SCE_GXM_TEXTURE_LINEAR` memory layout. YUV textures can only be used with the fragment pipeline (that is, cannot be used as vertex textures).

YUV422

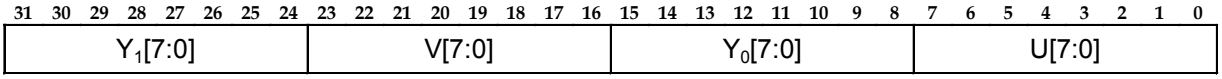
The YUV422 format (`SCE_GXM_TEXTURE_BASE_FORMAT_YUV422`) requires 4-byte alignment. This is an interleaved format. The UV channels are horizontally downsampled and stored interleaved with full resolution Y data.

Each 4-byte value encodes 2 sequential pixels within a scanline with values Y_0UV and Y_1UV.

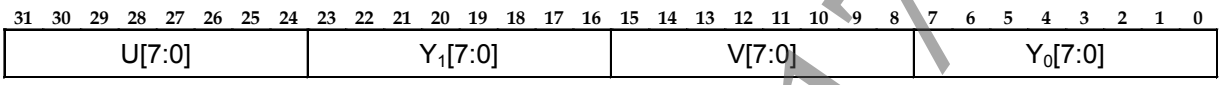
YUYV_CSC0, YUYV_CSC1



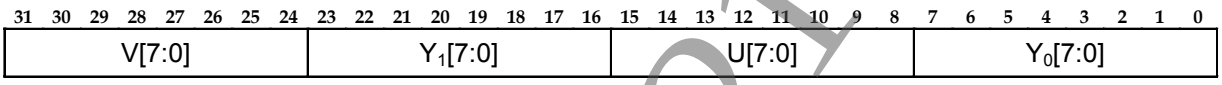
YVYU_CSC0, YVYU_CSC1



UYVY_CSC0, UYVY_CSC1

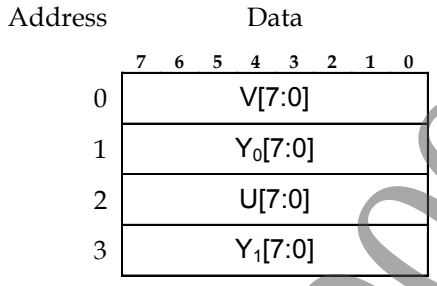


VYUY_CSC0, VYUY_CSC1



Where Y_0 is the left-hand pixel luminance, Y_1 is the right-hand pixel luminance and U and V are the shared chromaticity, all 8-bit unsigned integers.

All of these representations are stored little-endian in memory. For example, the **YUYV_CSC0** representation would be stored as:



The supported swizzles and query formats are shown in the following table.

Table 62 YUV422 Query Formats

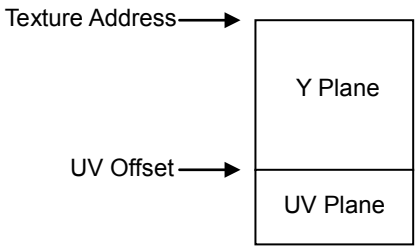
Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE_YUYV_CSC0 SCE_GXM_TEXTURE_SWIZZLE_YVYU_CSC0 SCE_GXM_TEXTURE_SWIZZLE_UYVY_CSC0 SCE_GXM_TEXTURE_SWIZZLE_VYUY_CSC0 SCE_GXM_TEXTURE_SWIZZLE_YUYV_CSC1 SCE_GXM_TEXTURE_SWIZZLE_YVYU_CSC1 SCE_GXM_TEXTURE_SWIZZLE_UYVY_CSC1 SCE_GXM_TEXTURE_SWIZZLE_VYUY_CSC1	• unsigned char4 • half4 • float4

YUV420P2

The YUV420P2 format (SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P2) requires 4-byte alignment. This is a planar format with a separate luminance (Y) plane and chromaticity (UV) plane.

The chromaticity plane is horizontally and vertically downsampled, with each chromaticity value shared by a 2x2 block of luminance values.

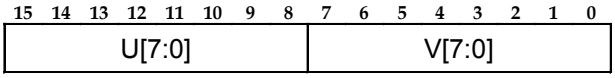
The UV plane always follows the Y plane in memory.



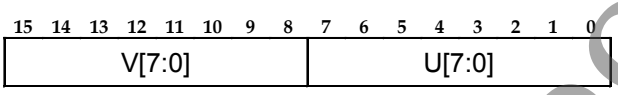
Within the Y plane, each Y value is encoded as an 8-bit unsigned integer.

Within the chromaticity plane, the UV values are stored interleaved. Each texel within the chromaticity plane can have the following representations:

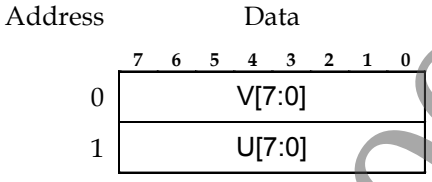
YUV_CSC0, YUV_CSC1



YVU_CSC0, YVU_CSC1



All of these representations are stored little-endian in memory. For example, the chromaticity plane in the **YUV_CSC0** representation would be stored as:



The supported swizzles and query formats are shown in the following table.

Table 63 YUV420P2 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE_YUV_CSC0 SCE_GXM_TEXTURE_SWIZZLE_YVU_CSC0 SCE_GXM_TEXTURE_SWIZZLE_YUV_CSC1 SCE_GXM_TEXTURE_SWIZZLE_YVU_CSC1	• unsigned char4 • half4 • float4

YUV420P3

The YUV420P3 format (SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P3) requires 4-byte alignment. This is a planar format with a separate luminance (Y) plane, U plane and V plane.

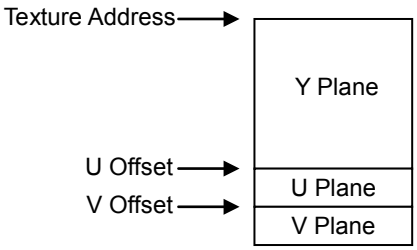
The U plane and V plane are horizontally and vertically downsampled, with each U value and V value shared by a 2x2 block of luminance values.

The U plane and V plane always follow the Y plane in memory.

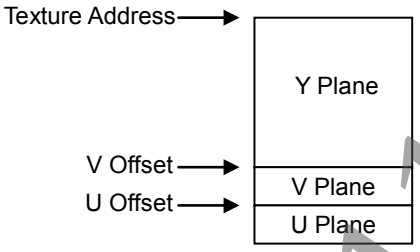
Within each plane, each pixel value is encoded as an 8-bit unsigned integer.

The swizzle defines the plane ordering in memory:

YUV_CSC0, YUV_CSC1



YVU_CSC0, YVU_CSC1



For further details on the memory layout and offsets of the YUV planes see section "YUV Linear Textures".

The supported swizzles and query formats are shown in the following table.

Table 64 YUV420P3 Query Formats

Swizzle Mode	Supported Query Formats
SCE_GXM_TEXTURE_SWIZZLE_YUV_CSC0 SCE_GXM_TEXTURE_SWIZZLE_YVU_CSC0 SCE_GXM_TEXTURE_SWIZZLE_YUV_CSC1 SCE_GXM_TEXTURE_SWIZZLE_YVU_CSC1	• unsigned char4 • half4 • float4

Texture Addressing Modes

Texture border data is optional, but affects which addressing modes can be used with the texture. There are 4 addressing modes for textures without border texels, and 4 addressing modes for textures with border texels provided.

The 4 non-border texture addressing modes may be used with textures of any type, apart from the following specific cases:

- Textures of type SCE_GXM_TEXTURE_LINEAR_STRIDED may only use the SCE_GXM_TEXTURE_ADDR_CLAMP mode.
- The SCE_GXM_TEXTURE_ADDR_MIRROR mode may only be used by textures of type SCE_GXM_TEXTURE_SWIZZLED.

The 4 addressing modes with border data may only be used with textures of type SCE_GXM_TEXTURE_SWIZZLED.

Non-Border Texture Addressing Modes

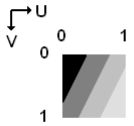
For textures that do not provide border texels, the following texture addressing modes, can be set independently for U and V:

Table 65 List of Non-Border Texture Addressing Modes

API Nomenclature	Description
SCE_GXM_TEXTURE_ADDR_REPEAT	Repeat texture infinitely
SCE_GXM_TEXTURE_ADDR_MIRROR	Mirror texture infinitely
SCE_GXM_TEXTURE_ADDR_CLAMP	Clamp to edge of texture
SCE_GXM_TEXTURE_ADDR_MIRROR_CLAMP	Mirror once then clamp to edge of texture

Original Texture

The following example texture image is used to show the different normal texture addressing modes:

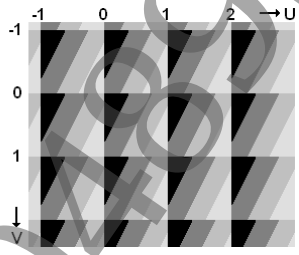


Repeat

The *repeat* addressing mode, SCE_GXM_TEXTURE_ADDR_REPEAT, is used with textures that do not provide border texels.

The texture is repeated for coordinate values outside of the range [0, 1]. Only the fractional part of the texture coordinate is used, the integer part of the texture coordinate is ignored.

The following image shows the example texture repeating in both the U and V directions:



Mirror

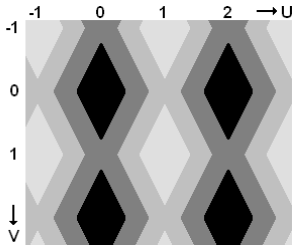
The *mirror* addressing mode, SCE_GXM_TEXTURE_ADDR_MIRROR, is used with textures that do not provide border texels.

The fractional part of the texture is used when the integer part of the texture is even.

The fractional part of the texture coordinate is mirrored using 1-frac when the integer part of the texture coordinate is odd.

This mode may only be used by textures of type SCE_GXM_TEXTURE_SWIZZLED.

The following image shows the example texture being flipped in both the U and V directions:



Clamp

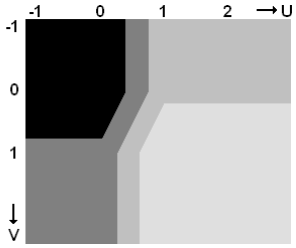
The *clamp* addressing mode, SCE_GXM_TEXTURE_ADDR_CLAMP, is used with textures that do not provide border texels.

The texture coordinate is clamped to the range [0,1]. Coordinate values greater than 1 will use 1, coordinate values less than 0 will use 0.

This mode does not filter across texture edges, so is equivalent to "clamp to edge" in other rendering APIs.

This mode is the only texture addressing mode supported by linear textures with explicit stride, `SCE_GXM_TEXTURE_LINEAR_STRIDED`.

The following image shows the example texture being clamped in both the U and V directions:



Mirror then Clamp

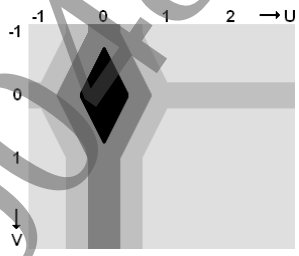
The *mirror then clamp* addressing mode, `SCE_GXM_TEXTURE_ADDR_MIRROR_CLAMP`, is used with textures that do not provide border texels.

The texture will be treated as addressing mode mirror for coordinates in the range [0, 1] and [0, -1] and then clamps the absolute value outside that range.

Coordinate values greater than 1 will use 1, coordinate values less than -1 will use 1.

This mode does not filter across texture edges, so is equivalent to "mirror once then clamp to edge" in other rendering APIs.

The following image shows the example texture being flipped then clamped in both the U and V directions:



Border Texture Addressing Modes

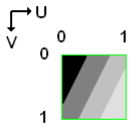
For swizzled textures that do provide border texels, the following texture addressing modes can be set independently for U and V:

Table 66 List of Border Texture Addressing Modes

API Nomenclature	Description
`SCE_GXM_TEXTURE_ADDR_REPEAT_IGNORE_BORDER`	Repeat texture infinitely, ignoring border texels
`SCE_GXM_TEXTURE_ADDR_CLAMP_IGNORE_BORDER`	Clamp to edge of texture, ignoring border texels
`SCE_GXM_TEXTURE_ADDR_CLAMP_FULL_BORDER`	Clamp to border area, showing full border texels at edges
`SCE_GXM_TEXTURE_ADDR_CLAMP_HALF_BORDER`	Clamp to edge of texture, halfway into border texels

Texture With Border

The following example texture image with a constant bright green border is used to show the different border texture addressing modes:



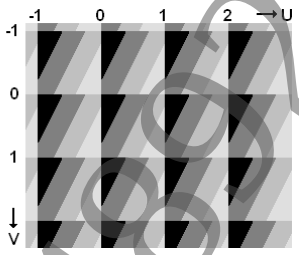
Repeat Ignore Border

The *repeat ignore border* addressing mode, `SCE_GXM_TEXTURE_ADDR_REPEAT_IGNORE_BORDER`, is used with textures that provide border texels. This addressing mode produces identical results to the *repeat* addressing mode, `SCE_GXM_TEXTURE_ADDR_REPEAT`, since the border texels are ignored.

The texture is repeated for coordinate values outside of the range [0, 1]. Only the fractional part of the texture coordinate is used, the integer part of the texture coordinate is ignored. The border texels are ignored.

This mode may only be used by textures of type `SCE_GXM_TEXTURE_SWIZZLED`.

The following image shows the example texture repeating in both the U and V directions:



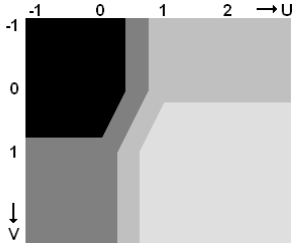
Clamp Ignore Border

The *clamp ignore border* addressing mode, `SCE_GXM_TEXTURE_ADDR_CLAMP_IGNORE_BORDER`, is used with textures that provide border texels.

The texture coordinate is clamped to the range [0,1]. Coordinate values greater than 1 will use 1, coordinate values less than 0 will use 0. The border texels are ignored, and this mode does not filter across texture edges.

This mode may only be used by textures of type `SCE_GXM_TEXTURE_SWIZZLED`.

The following image shows the example texture being clamped in both the U and V directions:



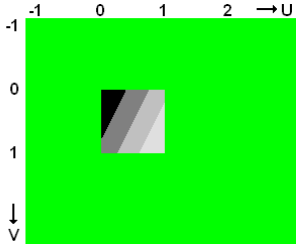
Clamp Full Border

The *clamp full border* addressing mode, `SCE_GXM_TEXTURE_ADDR_CLAMP_FULL_BORDER`, is used with textures that provide border texels.

Texture reads outside of the [0,1] range use border texels, and the texture coordinate is clamped once the coordinate is fully using border texel data.

This mode may only be used by textures of type `SCE_GXM_TEXTURE_SWIZZLED`.

The following image shows the example texture being clamped with full border in both the U and V directions:



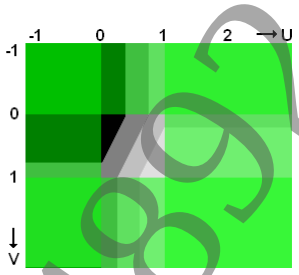
Clamp Half Border

The *clamp half border* addressing mode, `SCE_GXM_TEXTURE_ADDR_CLAMP_HALF_BORDER`, is used with textures that provide border texels.

Texture reads outside of the [0,1] range use border texels, and the texture coordinates are clamped to the [0,1] range. When using bilinear filtering, this produces a blend of 50% of the border texel data in U and V directions and 75% in the corners.

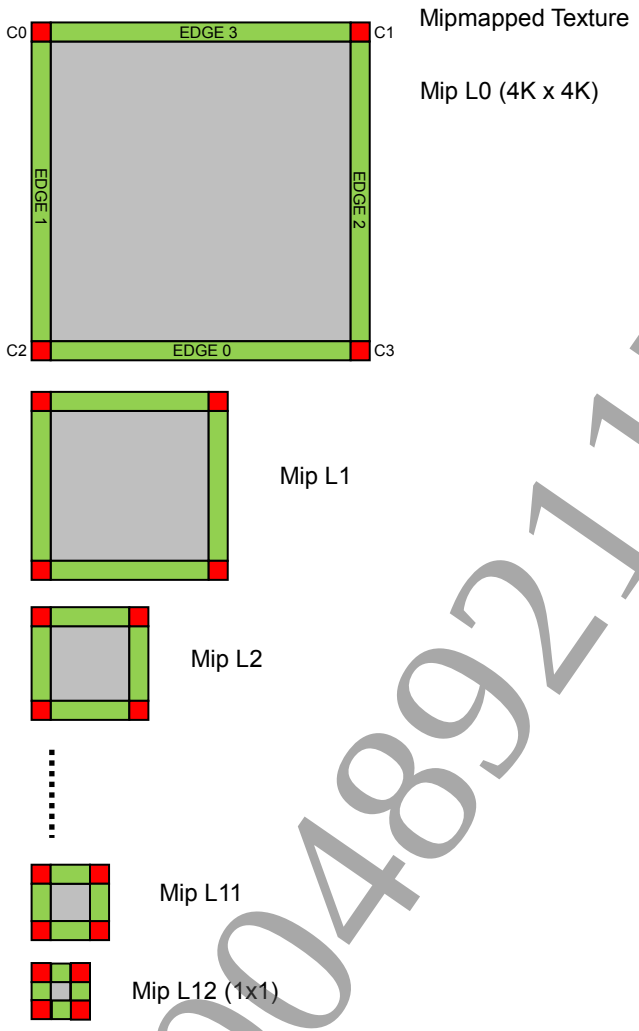
This mode may only be used by textures of type `SCE_GXM_TEXTURE_SWIZZLED`.

The following image shows the example texture being clamped with half border in both the U and V directions:



Texture Border Memory Layout

Figure 22 Border Memory Layout (4Kx4K)

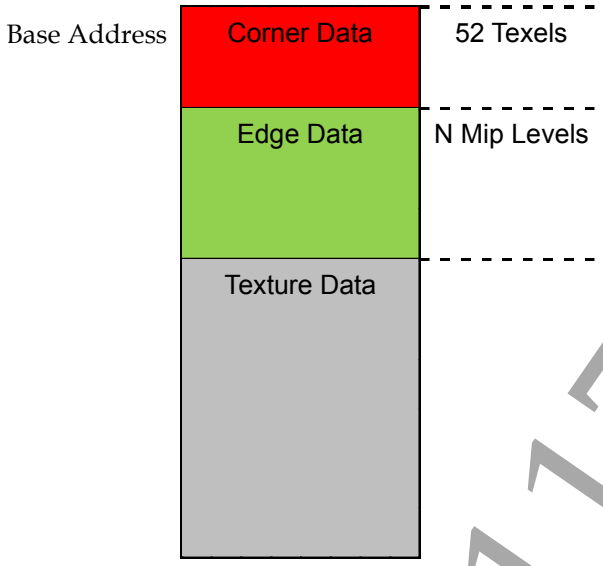


Power of 2 textures can be supplied with a one texel border that can be used with the texture map depending on the addressing modes of the texture. Each mip level has its own border consisting of 4 corners and 4 edges.

Corner data for 13 mip levels is always provided, regardless of the actual size of the texture. This therefore has a fixed footprint in memory of 52 texels.

Edge data is stored assuming that mipmaps L0 (the top mip level) to LN (the 1x1 mip level) are provided, where N depends on the size of the top mip level. Note that the full mip chain of edge texels is provided regardless of how many mip levels of texture data are stored.

Figure 23 Border Memory Addressing



Corner Data

The four corners for 13 mip levels requires 52 texels in memory.

The corner data takes up the first part of the border memory layout, starting with the corners for the 1x1 mip level up to the corners for the 4Kx4K mip level, see Figure 24. Note that this data is always stored regardless of the actual size of the texture.

The corners are stored in the following order for each mipmap:

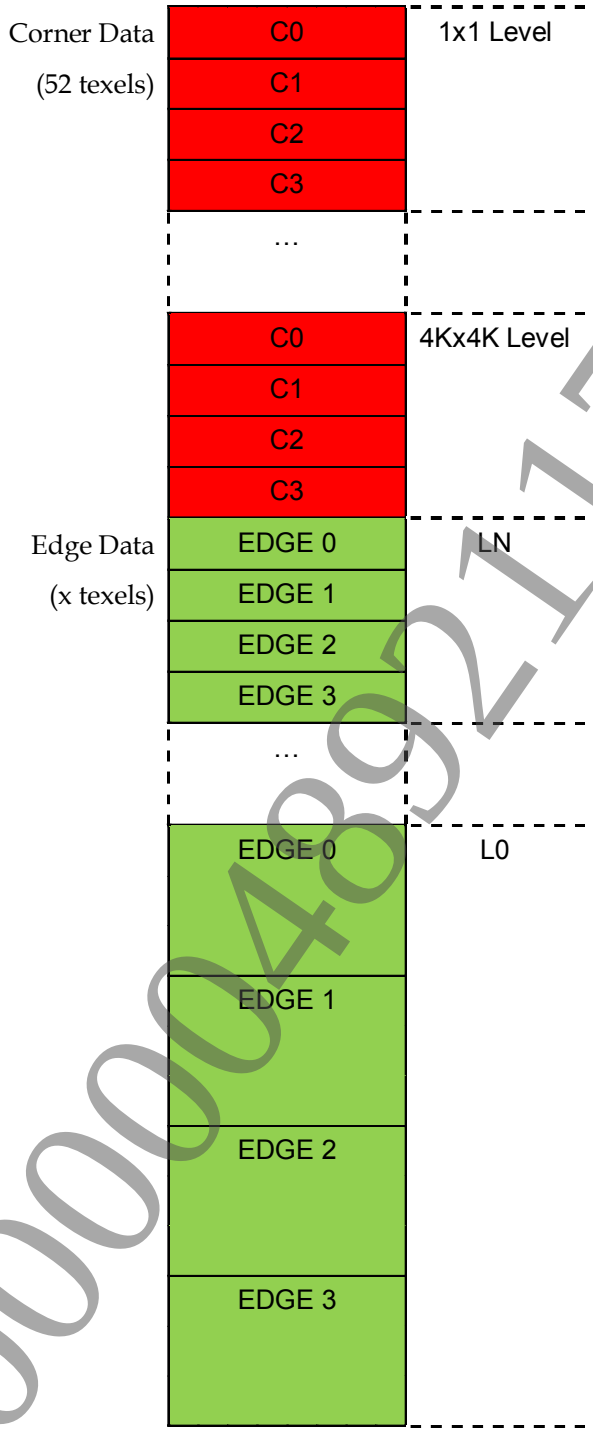
C0 (the top left corner)

C1 (the top right corner)

C2 (the bottom left corner)

C3 (the bottom right corner)

Figure 24 Corner and Edge Memory Addressing



Edge Data

The edge data follows the corner data in the border memory layout, starting with the edges for LN (the 1x1 mip level) up to the edges for L0 (the top mip level), see Figure 24.

The edges are stored in the following order for each mipmap:

EDGE 0 (the bottom edge, from left to right)

EDGE 1 (the left edge, from top to bottom)

EDGE 2 (the right edge, from top to bottom)

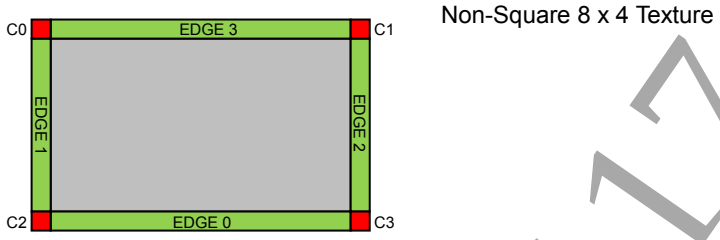
EDGE 3 (the top edge. from left to right)

Non Square Texture Edge Data

For memory usage the texture is always assumed to be square. This means that the length of all edges will be either the height or width of the texture, whichever is greater.

In the following example the texture is non square having width of 8 texels and height of 4 texels.

Figure 25 Border Memory Layout for non-Square Textures



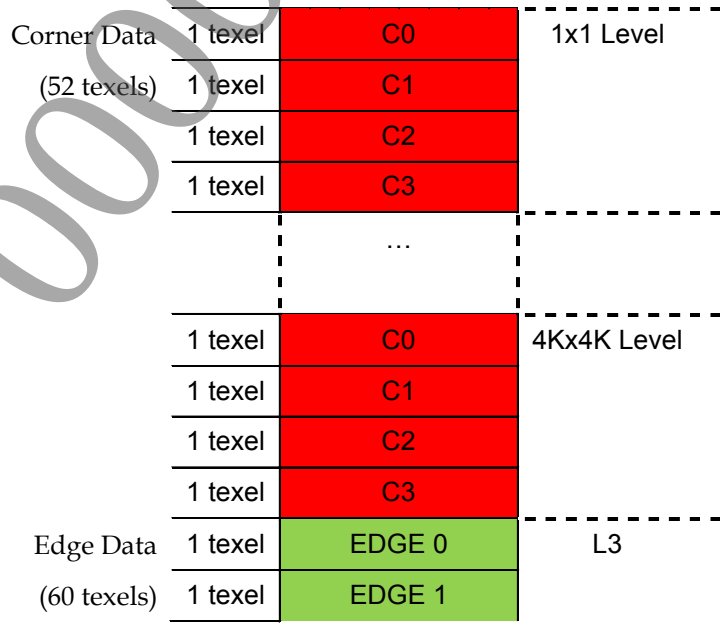
The edge texel memory usage for this texture will be calculated assuming an 8 X 8 texture map. The total number of edge texels is the sum of the length of each edge for mipmaps L3 (1 X 1) to L0 (8 X 8). In this case this will be 60 edge texels as shown in the following table.

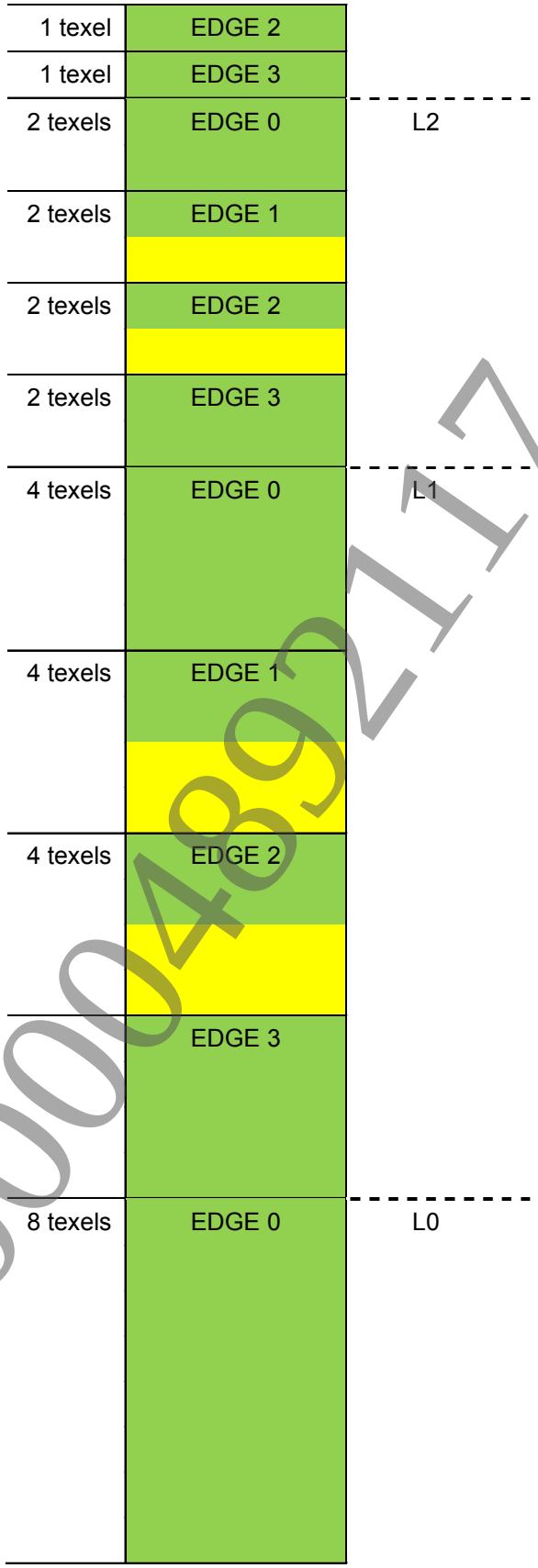
Table 67 Edge Texels Required for 8 X 8 Texture

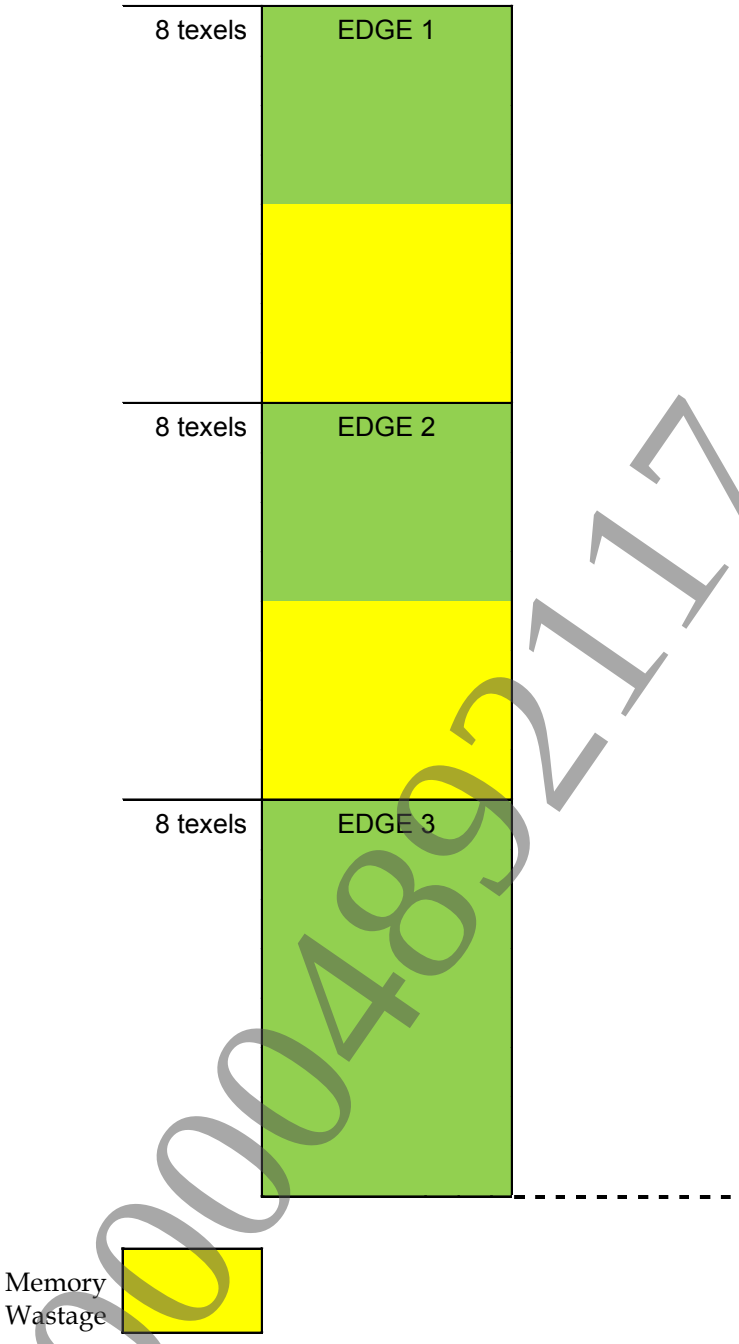
Mip Level	Mip Size	Texels
L3	1 x 1	4
L2	2 x 2	8
L1	4 x 4	16
L0	8 x 8	32
Total		60

The memory usage of this 8 x 4 texture, which only requires 46 edge texels, will therefore have some wastage as shown in the following diagram:

Figure 26 Corner and Edge Memory Addressing for non-Square Textures







Texture Data

The texture data is located in memory directly following the corner and edge data. The combined offset of corner and edge data (in texels) depends on the size of the texture according to the following table:

Table 68 Normal Texture Offset When Corner and Edge Data is Supplied

Texture Size	Texel Offset from Texture Base
1 X 1	56
2 X 2	64
4 X 4	80
8 X 8	112
16 X 16	176

Texture Size	Texel Offset from Texture Base
32 X 32	304
64 X 64	560
128 X 128	1072
256 X 256	2096
512 X 512	4144
1024 X 1024	8240
2048 X 2048	16432
4096 X 4096	32816

Single Border Color

The GPU does not directly support single border color for all borders like the OpenGL `GL_TEXTURE_BORDER_COLOR`. However this can be emulated by setting all edges and corners with the same color.

Unsupported Texture Formats

The following base texture formats and any derived from them are not supported with border memory:

- YUV Texture Formats
  `SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P2`
  `SCE_GXM_TEXTURE_BASE_FORMAT_YUV420P3`
  `SCE_GXM_TEXTURE_BASE_FORMAT_YUV422`
- Block Compressed Texture Formats
  `SCE_GXM_TEXTURE_BASE_FORMAT_PVRT2BPP`
  `SCE_GXM_TEXTURE_BASE_FORMAT_PVRT4BPP`
  `SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII2BPP`
  `SCE_GXM_TEXTURE_BASE_FORMAT_PVRTII4BPP`
  `SCE_GXM_TEXTURE_BASE_FORMAT_UBC1`
  `SCE_GXM_TEXTURE_BASE_FORMAT_UBC2`
  `SCE_GXM_TEXTURE_BASE_FORMAT_UBC3`
  `SCE_GXM_TEXTURE_BASE_FORMAT_UBC4`
  `SCE_GXM_TEXTURE_BASE_FORMAT_SBC4`
  `SCE_GXM_TEXTURE_BASE_FORMAT_UBC5`
  `SCE_GXM_TEXTURE_BASE_FORMAT_SBC5`
- Palettized Texture Formats
  `SCE_GXM_TEXTURE_BASE_FORMAT_P4`
  `SCE_GXM_TEXTURE_BASE_FORMAT_P8`

Texture Control Word Encoding

Textures are fully described by the four 32-bit words of data associated with a PDS `DOUTT` instruction.

These are documented in this section using the prefix `PDS_DOUTT#` and they match control words stored in the `SceGxmTexture` structure:

```
typedef struct SceGxmTexture {
    uint32_t controlWords[SCE_GXM_NUM_TEXTURE_CONTROL_WORDS];
} SceGxmTexture;
```

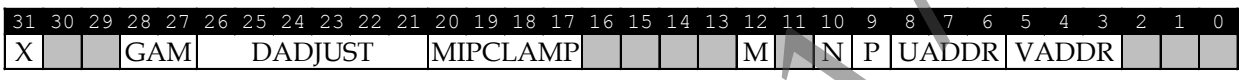
PDS_DOUTT#	SceGxmTexture
PDS_DOUTT0	controlWords[0]
PDS_DOUTT1	controlWords[1]
PDS_DOUTT2	controlWords[2]
PDS_DOUTT3	controlWords[3]

Note: The values of the four 32-bit words must be created only by using the predefined macro constants of field values and shift amounts. To ensure the consistency of application behavior in the future, it is strictly required that your application:

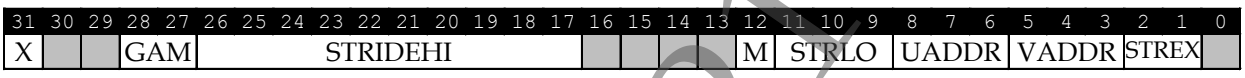
- Uses zero where a bit is not used by an encoding (shown gray in the diagrams).
- Does not use reserved values of fields.

PDS_DOUTT0

If T1_TEXTYPE is not LINEAR_STRIDED, the following encoding is used:



If T1_TEXTYPE is LINEAR_STRIDED, the following encoding is used:



Each field is documented as described in Table 69:

Table 69 PDS_DOUTT0 Texture Encoding

Bits	Name	Description
31	T0_TEXFEXT	When set, indicates that T1_TEXFORMAT refers to the extended list of formats.
28:27	T0_GAMMA	Gamma correction on texture read: 0 NONE 1 BGR_OR_R 2 Reserved 3 GR The BGR_OR_R setting converts the R, G and B channels if the texture formats has 3 or 4 components. For 1 and 2 component texture formats only the R channel is converted. The RG setting is only supported for 2 component texture formats, converting both the R and G channels. Texture formats that do not support gamma conversion should specify NONE as the gamma correction setting.
26:21	T0_DADJUST	This field may only be set if T1_TEXTYPE is not LINEAR_STRIDED. Adjustment of the level of detail by (value - 31)/8. All values are valid, but some specific values are shown below: 0 MIN -3.875 31 ZERO 0.0 63 MAX +4.0 For zero adjustment of the level of detail, a value of 31 should be set.

Bits	Name	Description
20:17	T0_MIPMAPCLAMP	This field may only be set if T1_TEXTYPE is not LINEAR_STRIDED. The level of detail to which the mip level is clamped. A value of 0 means that only the top level is supplied. A value of 12 means all 13 levels of a 4096x4096 are supplied (down to 1x1). A value of 15 indicates that there is no mip chain in the memory layout. This reduces padding requirements for YUV and cube map textures. 0 MIN Top mip present only 12 MAX Full 13 levels present 13 Reserved 14 Reserved 15 NOTMIPMAP No mips in memory layout Usually the mip count is converted into this mip clamp value (T0_MIPMAPCLAMP) using integer arithmetic modulo 16: `T0_MIPMAPCLAMP = (mip_count - 1) & 15;` In most cases, a mip clamp value of 0 has the same effect as a mip clamp value of 15 (corresponding to a mip count of 1 or 0). However for planar YUV textures or cube maps, these mip clamp values affect the memory layout between planes or cube faces. See "Memory Layouts" for more details. If T1_TEXTYPE is TILED, the smallest mip level must be at least 32 pixels in width and height.

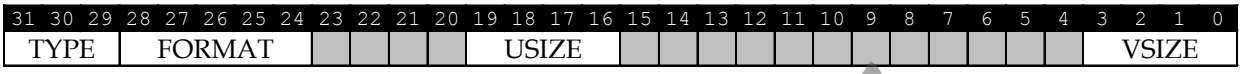
26:17	T0_STRIDEHI	This field may only be set if T1_TEXTYPE is LINEAR_STRIDED.
		Bits [12:3] of the internal stride. The internal stride is counted in 32-bit words, starting at 1 word. The stride in bytes (which must be a multiple of 4 bytes) may be converted to the internal stride using the following formula:
		`internal_stride = (byte_stride >> 2) - 1;`
		`T0_STRIDEHI = (internal_stride >> 3) & 0x3ff;`

12	T0_MAGFILTER	Linear filtering flag for magnification filter. If T1_TEXTYPE is LINEAR_STRIDED, this setting is also used for the minification filter.
10	T0_MINFILTER	This field may only be set if T1_TEXTYPE is not LINEAR_STRIDED. Linear filtering flag for minification filter.
9	T0_MIPFILTER	This field may only be set if T1_TEXTYPE is not LINEAR_STRIDED. Filtering between mip levels.
11:9	T0_STRIDELO	This field may only be set if T1_TEXTYPE is LINEAR_STRIDED. Bits [2:0] of the internal stride. The internal stride is counted in the same way as T0_STRIDEHI.
8:6	T0_UADDRMODE	The texture address mode to use for the U coordinate: 0 REPEAT 1 MIRROR 2 CLAMP 3 MIRROR_CLAMP 4 REPEAT_IGNORE_BORDER 5 CLAMP_FULL_BORDER 6 CLAMP_IGNORE_BORDER 7 CLAMP_HALF_BORDER The MIRROR address mode is only supported if T1_TEXTYPE is SWIZZLED. The address mode must be CLAMP if T1_TEXTYPE is LINEAR_STRIDED, CUBE, or CUBE_ARBITRARY. Border address modes are only supported if T1_TEXTYPE is SWIZZLED or SWIZZLED_ARBITRARY. Border address modes are not supported for YUV, block compressed, or palettized formats.

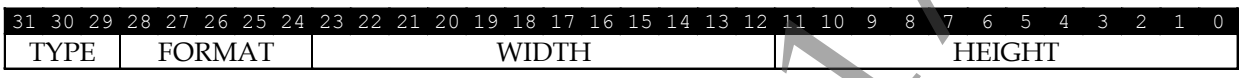
Bits	Name	Description
5:3	T0_VADDRMODE	The texture address mode to use for the V coordinate. The encoding and restrictions are the same as T0_UADDRMODE.
2:1	T0_STRIDEEX	This field may only be set if T1_TEXTYPE is LINEAR_STRIDED. Bits [14:13] of the internal stride. The internal stride is counted in the same way as T0_STRIDEHI.

PDS_DOUTT1

If T1_TEXTYPE is SWIZZLED or CUBE, the following encoding is used:



If T1_TEXTYPE is not SWIZZLED or CUBE, the following encoding is used:



Each field is documented as described in Table 70:

Table 70 PDS_DOUTT1 Texture Encoding

Bits	Name	Description
31:29	T1_TEXTYPE	The memory layout and type of the texture: 0 SWIZZLED 1 Reserved 2 CUBE 3 LINEAR 4 TILED 5 SWIZZLED_ARBITRARY 6 LINEAR_STRIDED 7 CUBE_ARBITRARY
28:24	T1_TEXFORMAT	When T0_TEXFEXT is set, the values refer to the extended set of formats. Otherwise the values refer to the standard set of formats. **Standard Format / Extended Format** 0 U8 / PVRT2BPP 1 S8 / PVRT4PPP 2 U4U4U4U4 / PVRTII2BPP 3 U8U3U3U2 / PVRTII4BPP 4 U1U5U5U5 / Reserved 5 U5U6U5 / UBC1 6 S5S5U6 / UBC2 7 U8U8 / UBC3 8 S8S8 / UBC4 9 U16 / SBC4 10 S16 / UBC5 11 F16 / SBC5 12 U8U8U8U8 / Reserved 13 S8S8S8S8 / Reserved 14 U2U10U10U10 / Reserved 15 U16U16 / Reserved 16 S16S16 / YUV420P2 17 F16F16 / YUV420P3 18 F32 / YUV422

Bits	Name	Description
		Standard Format / **Extended Format** 19 F32M / Reserved 20 X8S8S8U8 / P4 21 X8U24 / P8 22 Reserved / Reserved 23 U32 / Reserved 24 S32 / U8U8U8 25 SE5M9M9M9 / S8S8S8 26 F11F11F10 / U2F10F10F10 27 F16F16F16F16 / Reserved 28 U16U16U16U16 / Reserved 29 S16S16S16S16 / Reserved 30 F32F32 / Reserved 31 U32U32 / Reserved YUV420P2 and YUV420P3 are only supported if T1_TEXTYPE is LINEAR. YUV422 is only supported if T1_TEXTYPE is LINEAR or LINEAR_STRIDED. Block compressed formats are only supported if T1_TEXTYPE is SWIZZLED, SWIZZLED_ARBITRARY, CUBE, or CUBE_ARBITRARY. U8U8U8 and S8S8S8 are not supported if T1_TEXTYPE is LINEAR_STRIDED.
19:16	T1_USIZE	This field may only be set if T1_TEXTYPE is SWIZZLED or CUBE. The base-2 logarithm of the width of a power of 2 texture. Some specific values are shown below: 0 MIN 1 pixel 12 MAX 4096 pixels 13 Reserved 14 Reserved 15 Reserved
3:0	T1_VSIZE	This field may only be set if T1_TEXTYPE is SWIZZLED or CUBE. The base-2 logarithm of the height of a power of 2 texture. Some specific values are shown below: 0 MIN 1 pixel 12 MAX 4096 pixels 13 Reserved 14 Reserved 15 Reserved
23:12	WIDTH	This field may only be set if T1_TEXTYPE is not SWIZZLED or CUBE. The width of the texture minus 1. The minimum value of 0 indicates 1 pixel width, and the maximum value of 4095 indicates 4096 pixel width. If T1_TEXTYPE is TILED, the minimum width is 32 pixels.
11:0	HEIGHT	This field may only be set if T1_TEXTYPE is not SWIZZLED or CUBE. The height of the texture minus 1. The minimum value of 0 indicates 1 pixel height, and the maximum value of 4095 indicates 4096 pixel height. If T1_TEXTYPE is TILED, the minimum height is 32 pixels.

PDS_DOUTT2

The following encoding is used:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2	1 0
TEXADDR	LHI

Each field is documented as described in Table 71:

Table 71 PDS_DOUTT2 Texture Encoding

Bits	Name	Description
31:2	TEXADDR	Bits [31:2] of the texture address. Some texture formats require further alignment beyond the 4-byte alignment of this encoding.
1:0	LODMINHI	Bits [3:2] of the minimum level of detail value.

PDS_DOUTT3

The following encoding is used:

31	30 29 28	27 26	25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
N	SWIZ	LLO	PALETTEADDR

Each field is documented as Table 72:

Table 72 PDS_DOUTT3 Texture Encoding

Bits	Name	Description
31	NORM	Flag indicating that integer-to-float conversion inside the texture unit should be normalized to produce results in the range [0.0, 1.0] for unsigned and [-1.0, 1.0] for signed.
30:28	SWIZ	The component ordering in memory for 3 and 4 component texture formats (and depth/stencil and YUV formats), and the component ordering in the ABGR result for 1 and 2 component texture formats (and block compressed formats). For 4 or 3 component texture formats: **4 components / 3 components** 0 ABGR / BGR 1 ARGB / RGB 2 RGBA / Reserved 3 BGRA / Reserved 4 1BGR / Reserved 5 1RGB / Reserved 6 RGB1 / Reserved 7 BGR1 / Reserved For 2 component texture formats: **2 components / 1 component** 0 GR / R Packed result 1 00GR / 000R 2 GRRR / 111R 3 RGGG / RRRR 4 GRGR / 0RRR 5 00RG / 1RRR 6 Reserved / R000 7 Reserved / R111 For depth/stencil texture formats: 0 SD 1 DS For YUV formats:

Bits	Name	Description
		YUV422 / **YUV420** 0 YUYV_CSC0 / YUV_CSC0 1 YVYU_CSC0 / YVU_CSC0 2 UYVY_CSC0 / YUV_CSC1 3 VYUY_CSC0 / YVU_CSC1 4 YUYV_CSC1 / Reserved 5 YVYU_CSC1 / Reserved 6 UYVY_CSC1 / Reserved 7 VYUY_CSC1 / Reserved For UBC1, UBC2, UBC3, and PVRT formats, the component ordering must be ABGR or 1BGR. For format U8U3U3U2, the component ordering must be ARGB.
27:26	LODMINLO	Bits [1:0] of the minimum level of detail value.
25:0	PALETTEADDR	Bits [31:6] of the 64-byte aligned palette address.

10 Color Surface

Types

The color surface can be stored in different memory layouts. The color surface memory layouts have the same layout as texture data.

The following color surface memory layout types are supported:

Table 73 List of Supported Color Surface Types

Type	Notes
SCE_GXM_COLOR_SURFACE_LINEAR	Uses linear memory layout with pixels stored in scanline order.
SCE_GXM_COLOR_SURFACE_SWIZZLED	Uses swizzled memory layout with pixels stored in Morton order.
SCE_GXM_COLOR_SURFACE_TILED	Uses tiled memory layout.

Color Surface Size

The maximum color surface size is 4096 pixels in width and height.

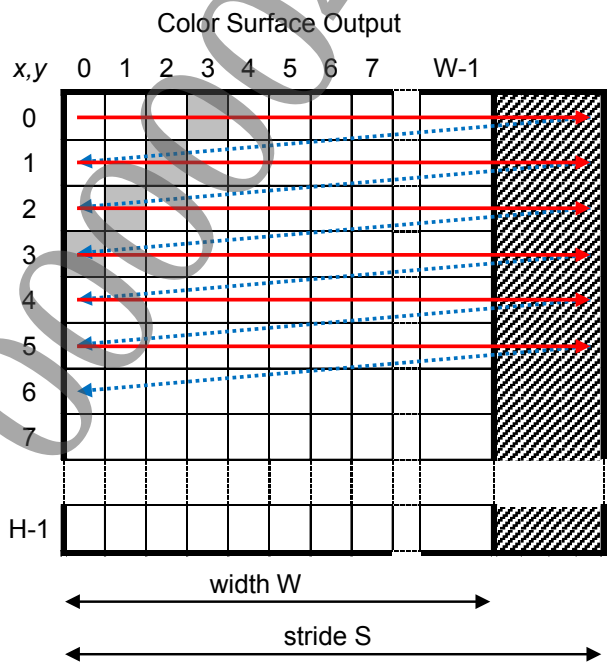
For swizzled surfaces, the minimum size is 16 pixels in width and height.

Memory Layouts

Linear Color Surface

Data is output linearly from left to right in scan line format.

Figure 27 Linear Color Surface Memory Layout



Swizzled Color Surface

Swizzled color surfaces are stored in Morton order: as a 2x2 blocks of pixels, and 2x2 blocks of blocks and so on, as shown in the following diagram.

Figure 28 Swizzled Color Surface Memory Layout

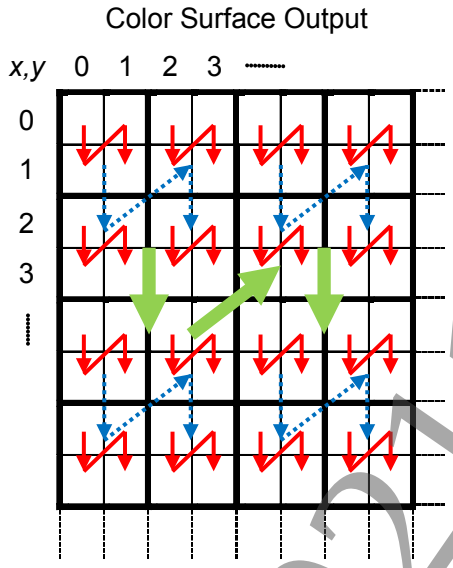


Figure 29 Swizzled Color Surface Memory Map

Memory Offset	x,y
0	0,0
1	0,1
2	1,0
3	1,1
4	0,2
5	0,3
6	1,2
7	1,3
8	2,0
9	2,1
10	3,0
	...

The address for any given pixel is calculated by interleaving the horizontal and vertical pixel coordinates as shown below:

Figure 30 Swizzled Color Surface Address Calculation

…	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
and so on	x6	y6	x5	y5	x4	y4	x3	y3	x2	y2	x1	y1	x0	y0	

When the color surface dimension is not equal on both the vertical and horizontal, the least significant bits of the address are interleaved first.

Any remaining bits are appended to the top of the address.

Swizzled surfaces produce data that is texture cache friendly for reading in later rendering passes.

Tiled Color Surface

Tiled color surfaces are stored as 32x32 tiles in memory, in scan line sequential order. The pixels are also stored in scan line sequential order within each tile.

The width and height of the color surfaces must be at least 32 pixels but need not be an exact multiple of 32. However, the memory footprint is always a whole number of tiles. So, for example, it is valid for to have a 48x40 tiled color surface, but this will have memory footprint of a 64x64 tiled color surface since this is the closest multiple of 32 in width and height.

Figure 31 Tiled Color Surface Memory Layout

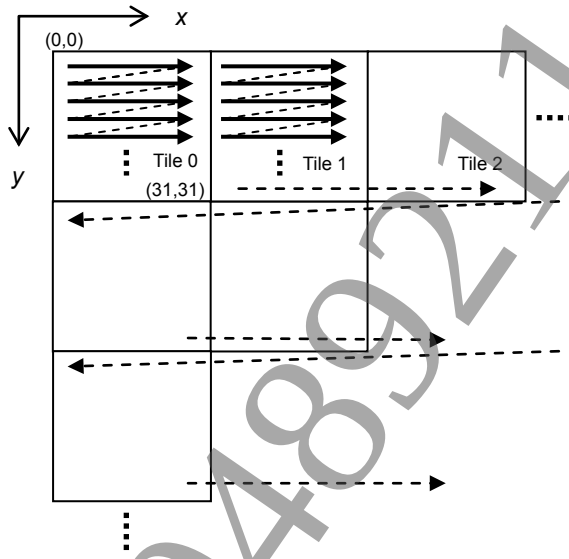
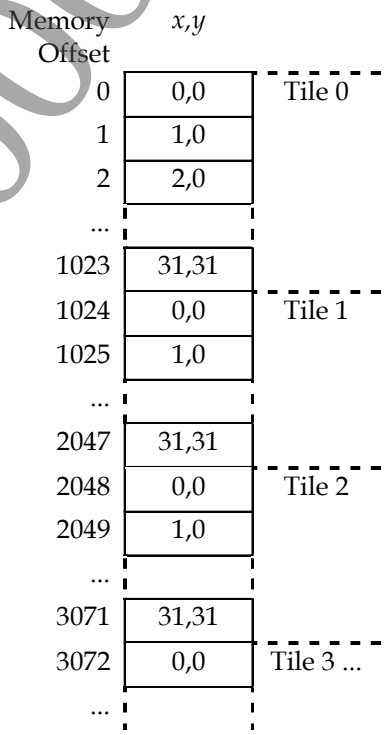
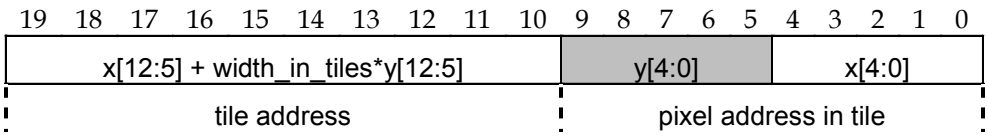


Figure 32 Tiled Color Surface Memory Map



The address for any given tile and pixel is calculated thus:

Figure 33 Tiled Color Surface Address Calculation



Where:

width_in_tiles = scan line width in tiles

x = horizontal coordinate

y = vertical coordinate

Pixel Formats

Base Color Surface Formats

The following base color surface formats are supported on the GPU:

Table 74 List of Supported Color Surface Formats

Format	Description	Matching Texture Format
SCE_GXM_COLOR_BASE_FORMAT_U8	Unsigned 8-bit.	U8
SCE_GXM_COLOR_BASE_FORMAT_S8	Signed 8-bit.	S8
SCE_GXM_COLOR_BASE_FORMAT_U16	Unsigned 16-bit.	U16
SCE_GXM_COLOR_BASE_FORMAT_S16	Signed 16-bit.	S16
SCE_GXM_COLOR_BASE_FORMAT_F16	16-bit half-precision floating-point.	F16
SCE_GXM_COLOR_BASE_FORMAT_F32	32-bit floating-point.	F32
SCE_GXM_COLOR_BASE_FORMAT_U8U8	Two unsigned 8-bit.	U8U8
SCE_GXM_COLOR_BASE_FORMAT_S8S8	Two signed 8-bit.	S8S8
SCE_GXM_COLOR_BASE_FORMAT_U16U16	Two unsigned 16-bit.	U16U16
SCE_GXM_COLOR_BASE_FORMAT_S16S16	Two signed 16-bit.	S16S16
SCE_GXM_COLOR_BASE_FORMAT_F16F16	Two 16-bit half precision floating-point.	F16F16
SCE_GXM_COLOR_BASE_FORMAT_F32F32	Two 32-bit floating-point.	F32F32
SCE_GXM_COLOR_BASE_FORMAT_U5U6U5	Unsigned 5-bit, unsigned 6-bit, unsigned 5-bit.	U5U6U5
SCE_GXM_COLOR_BASE_FORMAT_S5S5U6	Signed 5-bit, signed 5-bit, unsigned 6-bit.	S5S5U6
SCE_GXM_COLOR_BASE_FORMAT_U8U8U8	Three unsigned 8-bit.	U8U8U8
SCE_GXM_COLOR_BASE_FORMAT_SE5M9M9M9	Shared 5 bit exponent, three partial-precision floating-point mantissa.	SE5M9M9M9
SCE_GXM_COLOR_BASE_FORMAT_F11F11F10	Three partial-precision floating-point.	F11F11F10
SCE_GXM_COLOR_BASE_FORMAT_U4U4U4U4	Four unsigned 4-bit.	U4U4U4U4
SCE_GXM_COLOR_BASE_FORMAT_U8U8U8U8	Four unsigned 8-bit.	U8U8U8U8
SCE_GXM_COLOR_BASE_FORMAT_S8S8S8S8	Four signed 8-bit.	S8S8S8S8
SCE_GXM_COLOR_BASE_FORMAT_U8U3U3U2	Unsigned 8-bit, unsigned 3-bit, unsigned 3-bit, unsigned 2-bit.	U8U3U3U2

Format	Description	Matching Texture Format
SCE_GXM_COLOR_BASE_FORMAT_U1U5U5U5	Unsigned 1-bit, unsigned 5-bit, unsigned 5-bit, unsigned 5-bit.	U1U5U5U5
SCE_GXM_COLOR_BASE_FORMAT_U2U10U10U10	Unsigned 2-bit, unsigned 10-bit, unsigned 10-bit, unsigned 10-bit.	U2U10U10U10
SCE_GXM_COLOR_BASE_FORMAT_F16F16F16F16	Four 16-bit half precision floating-point.	F16F16F16F16
SCE_GXM_COLOR_BASE_FORMAT_U2F10F10F10	Unsigned 2-bit and three partial-precision floating-point.	U2F10F10F10

Derived Color Surface Formats

A color format is defined by combining a base color format and compatible swizzle mode. The libgxm library provides an enumeration of all color formats supported by the GPU. For more details see the `SceGxmColorFormat` in the *libgxm Reference*.

Output Register Format

The output register format defines the format in which fragments (that is, pixels or samples) are held in the GPU before they are stored to memory. While a tile is being rendered, the fragment program reads (during blending) and writes in this output register format (independent of the color surface format). Once all rendering within the tile is complete, all pixels within the tile are converted into the color surface format and stored in memory.

The Cg types that can be used as output register formats are:

- float2
- half4 (the default), half2
- unsigned short2
- signed short2
- unsigned char4
- signed char4

Each color surface format supports one or two output register formats.

For a single table that shows the output register formats supported by each color surface format, see the summary table in Appendix A.

Dithered Color Surfaces

The lower precision color formats support dithering when a tile is stored to memory.

Gamma Correction

When gamma correction is enabled, the shader converts values from linear color space to sRGB color space. The component values that are gamma corrected will depend on the color surface format.

Gamma correction can only be used for linear and tiled color surface memory layouts. Gamma correction cannot be used for swizzled color surface memory layouts.

Note that when both gamma correction and downscaling are enabled, the downscaling operation is performed first.

Downscale

When downscaling is enabled, a tile is downscaled using a box filter before storing to memory. Color surface downscaling can only be used with supported formats and when multi-sample antialiasing (MSAA) is enabled.

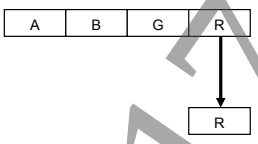
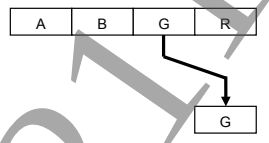
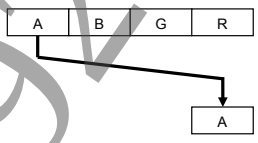
Formats that support downscaling are indicated in the Downscaling column in the output format table for each color surface format definition.

Single Component Color Surface Formats

Single Component Swizzle Modes

Unless otherwise documented in the color format description, the values from the shader in ABGR component ordering are stored to memory according to the swizzle mode shown in Table 75.

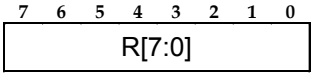
Table 75 Single Component Color Surface Swizzle Modes

Mode	ABGR Representation
SCE_GXM_COLOR_SWIZZLE1_R	
SCE_GXM_COLOR_SWIZZLE1_G	
SCE_GXM_COLOR_SWIZZLE1_A	

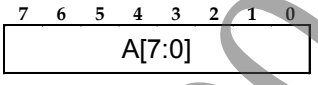
U8

The U8 color surface format (`SCE_GXM_COLOR_BASE_FORMAT_U8`) occupies 1 byte in memory and can have the following 8-bit representations:

R

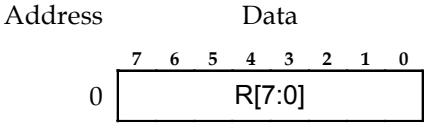


A



Where R is interpreted as the unsigned 8-bit red component value, and A is interpreted as the unsigned 8-bit alpha.

These representations are stored little-endian in memory. For example, the **R** representation would be stored as:



The swizzle modes are shown in Table 75.

The modes of operation and supported swizzle modes for this format are shown in Table 76:

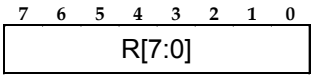
Table 76 U8 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE1_R SCE_GXM_COLOR_SWIZZLE1_A	• unsigned char4 • half2 - This output format is only available for use when gamma correction is enabled and the R representation is used. The R color component is gamma correct before being written to memory.	Not supported	Supported

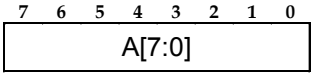
S8

The S8 color surface format (SCE_GXM_COLOR_BASE_FORMAT_S8) occupies 1 byte in memory and can have the following 8-bit representations:

R

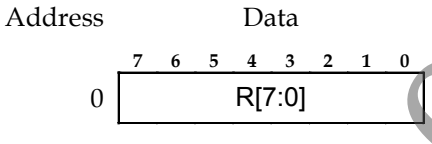


A



Where R is interpreted as the signed 8-bit red component value, and A is interpreted as the signed 8-bit alpha.

These representations are stored little-endian in memory. For example, the **R** representation would be stored as:



The swizzle modes are shown in Table 75.

The modes of operation and supported swizzle modes for this format are shown in the following table:

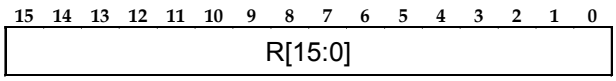
Table 77 S8 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE1_R SCE_GXM_COLOR_SWIZZLE1_A	• signed char4	Not supported	Supported

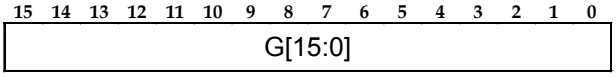
U16

The U16 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U16) occupies 2 bytes in memory and can have the following 16-bit representations:

R

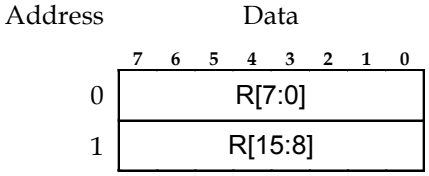


G



Where R is interpreted as the unsigned 16-bit red component value, and G is interpreted as the unsigned 16-bit green.

These representations are stored little-endian in memory. For example, the **R** representation would be stored as:



The swizzle modes are shown in Table 75.

The modes of operation and supported swizzle modes for this format are shown in the following table:

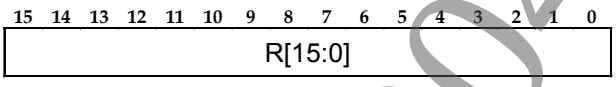
Table 78 U16 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE1_R SCE_GXM_COLOR_SWIZZLE1_G	• unsigned short2	Not supported	Supported

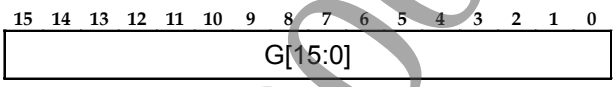
S16

The S16 color surface format (SCE_GXM_COLOR_BASE_FORMAT_S16) occupies 2 bytes in memory and can have the following 16-bit representations:

R

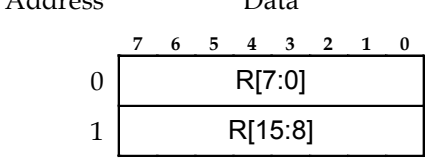


G



Where R is interpreted as the signed 16-bit red component value, and G is interpreted as the signed 16-bit green.

These representations are stored little-endian in memory. For example, the **R** representation would be stored as:



The swizzle modes are shown in Table 75.

The modes of operation and supported swizzle modes for this format are shown in the following table:

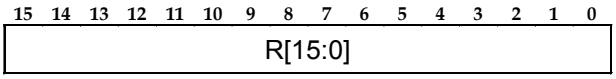
Table 79 S16 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE1_R SCE_GXM_COLOR_SWIZZLE1_G	• signed short2	Not supported	Supported

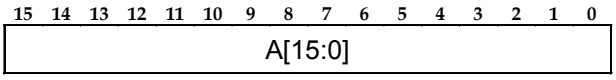
F16

The F16 color surface format (SCE_GXM_COLOR_BASE_FORMAT_F16) occupies 2 bytes in memory and can have the following 16-bit representations:

R

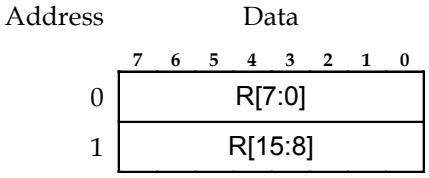


G



Where R is interpreted as the 16-bit half precision floating-point red component value, and G is interpreted as the 16-bit half precision floating-point green.

These representations are stored little-endian in memory. For example, the **R** representation would be stored as:



The swizzle modes are shown in Table 75.

The modes of operation and supported swizzle modes for this format are shown in the following table:

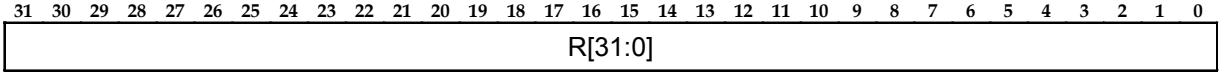
Table 80 F16 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE1_R SCE_GXM_COLOR_SWIZZLE1_G	• half2	Not supported	Supported

F32

The F32 color surface format (SCE_GXM_COLOR_BASE_FORMAT_F32) occupies 4 bytes in memory has the following 32-bit representations:

R



Where R is the 32-bit floating-point red component value.

This representation is stored little-endian in memory:

Address Data

	7	6	5	4	3	2	1	0
0	R[7:0]							
1	R[15:8]							
2	R[23:16]							
3	R[31:24]							

The swizzle modes are shown in Table 75.

The modes of operation and supported swizzle modes for this format are shown in the following table:

Table 81 F32 Output Formats

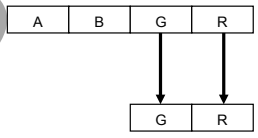
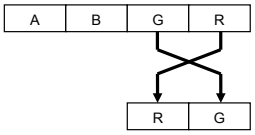
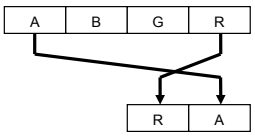
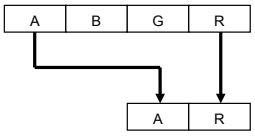
Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE1_R	• float	Not supported	Not supported

Two Component Color Surface Formats

Two Component Swizzle Modes

Unless otherwise documented in the color format description, the values from the shader in ABGR component ordering are stored to memory according to the swizzle mode shown in Table 82.

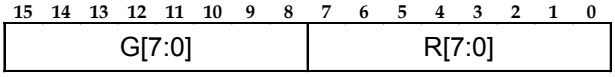
Table 82 Two Component Color Surface Swizzle Modes

Mode	ABGR Representation
SCE_GXM_COLOR_SWIZZLE2_GR	
SCE_GXM_COLOR_SWIZZLE2_RG	
SCE_GXM_COLOR_SWIZZLE2_RA	
SCE_GXM_COLOR_SWIZZLE2_AR	

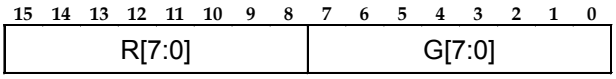
U8U8

The U8U8 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U8U8) occupies 2 bytes in memory can have the following 16-bit representations:

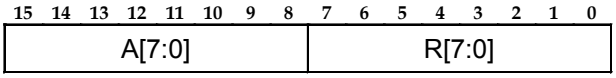
GR



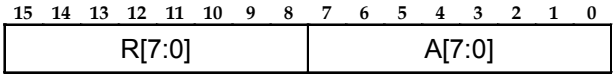
RG



AR

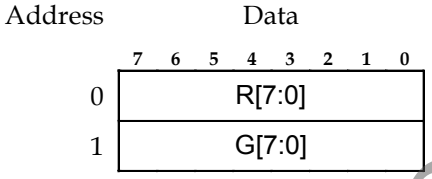


RA



Where R is the unsigned 8-bit red, G is the unsigned 8-bit green and A is the unsigned alpha component values.

All of these representations are stored little-endian in memory. For example, the **GR** representation would be stored as:



The swizzle modes as shown in Table 82 are supported.

The modes of operation for this format are shown in the following table:

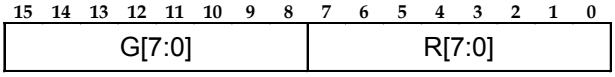
Table 83 U8U8 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE2_GR SCE_GXM_COLOR_SWIZZLE2_RG SCE_GXM_COLOR_SWIZZLE2_RA SCE_GXM_COLOR_SWIZZLE2_AR	• unsigned char4 • half2 - This output format is only available for use when gamma correction is enabled and either the GR or RG representation is used. Either just the R color component or both the R and G color components can be selected to be gamma correct before being written to memory.	Not supported	Supported

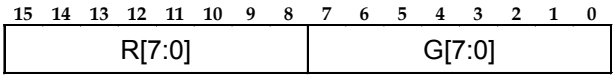
S8S8

The S8S8 color surface format (SCE_GXM_COLOR_BASE_FORMAT_S8S8) occupies 2 bytes in memory can have the following 16-bit representations:

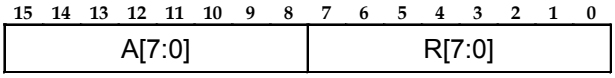
GR



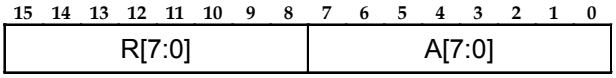
RG



AR

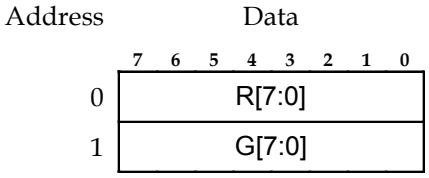


RA



Where R is the signed 8-bit red, G is the signed 8-bit green and A is the signed alpha component values.

All of these representations are stored little-endian in memory. For example, the **GR** representation would be stored as:



The swizzle modes as shown in Table 82 are supported.

The modes of operation for this format are shown in the following table:

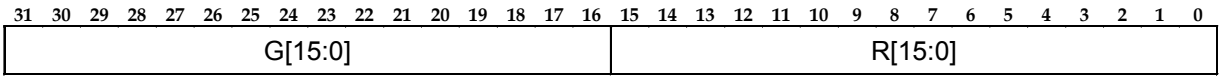
Table 84 S8S8 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE2_GR SCE_GXM_COLOR_SWIZZLE2_RG SCE_GXM_COLOR_SWIZZLE2_RA SCE_GXM_COLOR_SWIZZLE2_AR	• signed char4	Not supported	Supported

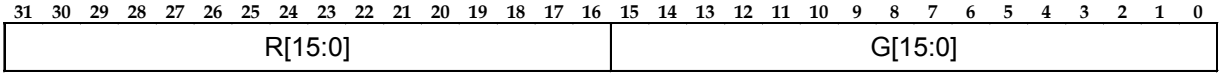
U16U16

The U16U16 color surface format (`SCE_GXM_COLOR_BASE_FORMAT_U16U16`) occupies 4 bytes in memory can have the following 32-bit representations:

GR



RG



Where R is the unsigned 16-bit red, and G is the unsigned 16-bit green component values.

These representations are stored little-endian in memory. For example, the **GR** representation would be stored as:

The swizzle modes are shown in Table 82.

The modes of operation and supported swizzle modes for this format are shown in the following table:

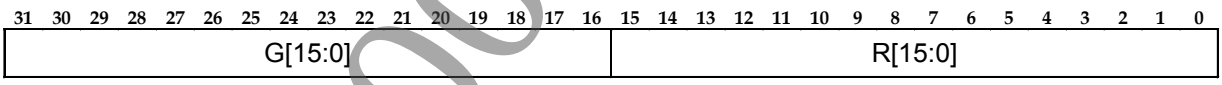
Table 85 U16U16 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE2_GR SCE_GXM_COLOR_SWIZZLE2_RG	• unsigned short2	Not supported	Supported

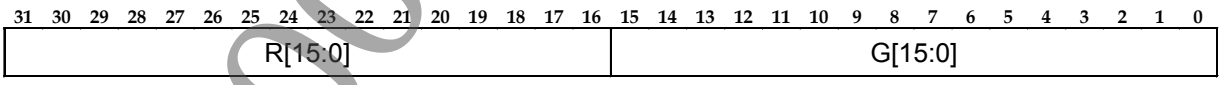
S16S16

The S16S16 color surface format (SCE_GXM_COLOR_BASE_FORMAT_S16S16) occupies 4 bytes in memory can have the following 32-bit representations:

GR

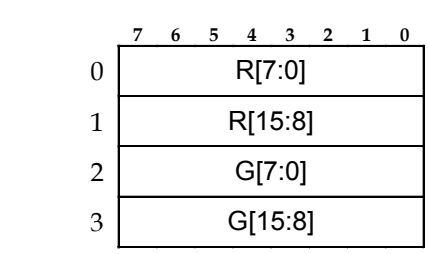


RG



Where R is the signed 16-bit red, and G is the signed 16-bit green component values.

These representations are stored little-endian in memory. For example, the **GR** representation would be stored as:



The swizzle modes are shown in Table 82.

The modes of operation and supported swizzle modes for this format are shown in the following table:

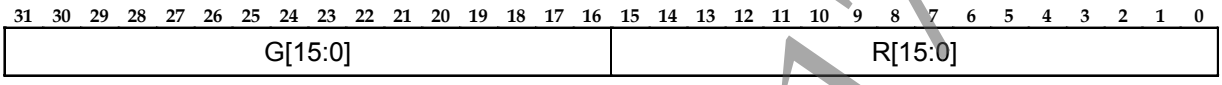
Table 86 S16S16 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE2_GR SCE_GXM_COLOR_SWIZZLE2_RG	• signed short2	Not supported	Supported

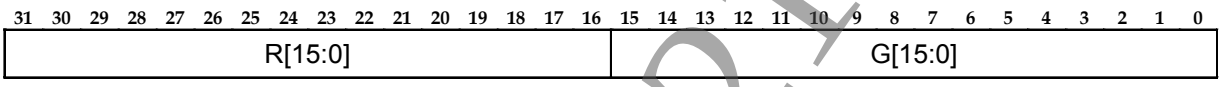
F16F16

The F16F16 color surface format (SCE_GXM_COLOR_BASE_FORMAT_F16F16) occupies 4 bytes in memory can have the following 32-bit representations:

GR



RG



Where R is the 16-bit half precision floating-point red, and G is the 16-bit half precision floating-point green component values.

These representations are stored little-endian in memory. For example, the **GR** representation would be stored as:

The swizzle modes are shown in Table 82.

The modes of operation and supported swizzle modes for this format are shown in the following table:

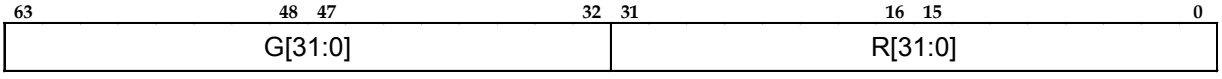
Table 87 F16F16 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE2_GR SCE_GXM_COLOR_SWIZZLE2_RG	• half2	Not supported	Supported

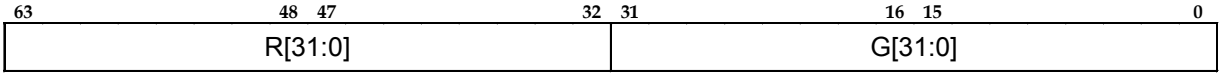
F32F32

The F32F32 color surface format (SCE_GXM_COLOR_BASE_FORMAT_F32F32) occupies 8 bytes in memory and can have the following 64-bit representations:

GR

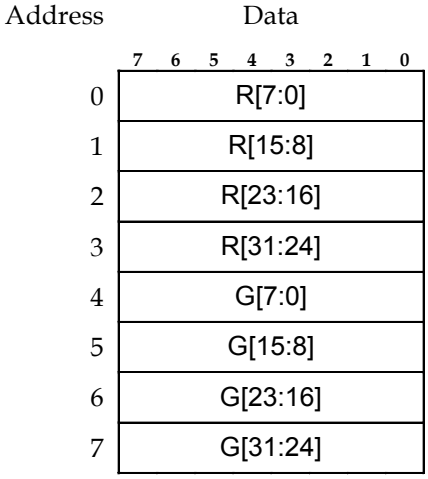


RG



Where R is the 32-bit floating-point red, and G is the 32-bit floating-point green component values.

These representations are stored little-endian in memory. For example, the **GR** representation would be stored as:



The swizzle modes are shown in Table 82.

The modes of operation for this format are shown in the following table:

Table 88 F32F32 Output Formats

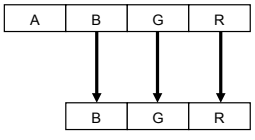
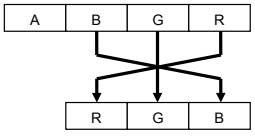
Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE2_GR SCE_GXM_COLOR_SWIZZLE2_RG	• float2	Not supported	Not supported

Three Component Color Surface Formats

Three Component Swizzle Modes

Unless otherwise documented in the color format description, the values from the shader in ABGR component ordering are stored to memory according to the swizzle mode shown in Table 89.

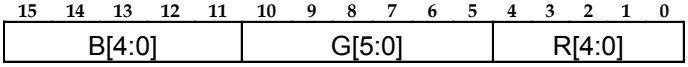
Table 89 Three Component Color Surface Swizzle Modes

Mode	ABGR Representation
SCE_GXM_COLOR_SWIZZLE3_BGR	
SCE_GXM_COLOR_SWIZZLE3_RGB	

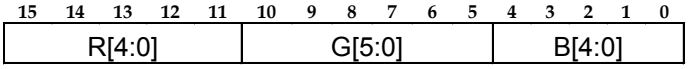
U5U6U5

The U5U6U5 color surface format (`SCE_GXM_COLOR_BASE_FORMAT_U5U6U5`) occupies 2 bytes in memory and can have the following 16-bit representations:

BGR

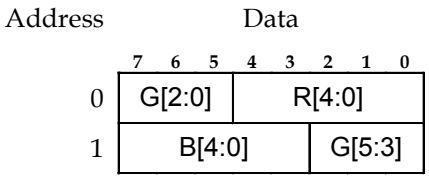


RGB



Where R is the unsigned 5-bit red, G is the unsigned 6-bit green, and B is the unsigned 5-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 89 are supported.

The modes of operation for this format are shown in the following table:

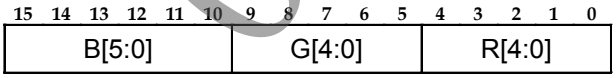
Table 90 U5U6U5 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
`SCE_GXM_COLOR_SWIZZLE3_BGR` `SCE_GXM_COLOR_SWIZZLE3_RGB`	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components are gamma correct before being written to memory.	Supported	Supported

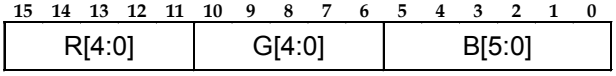
S5S5U6

The S5S5U6 color surface format (`SCE_GXM_COLOR_BASE_FORMAT_S5S5U6`) occupies 2 bytes in memory and can have the following 16-bit representations:

BGR

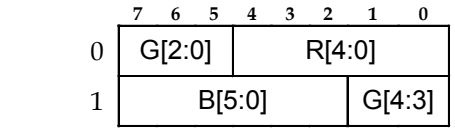


RGB



Where R is the signed 5-bit red, G is the signed 5-bit green, and B is the unsigned 6-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 89 are supported.

The supported query formats are shown in the following table.

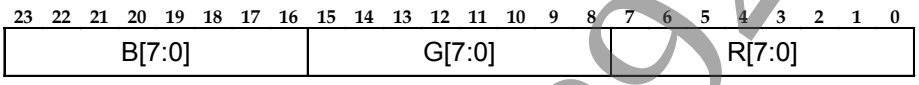
Table 91 S5S5U6 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE3_BGR SCE_GXM_COLOR_SWIZZLE3_RGB	• half4	Not supported	Supported

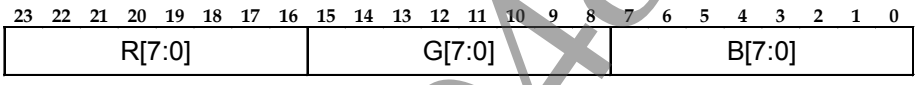
U8U8U8

The U8U8U8 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U8U8U8) is a packed texture format, occupies 3 bytes in memory and can have the following 24-bit representations.

BGR

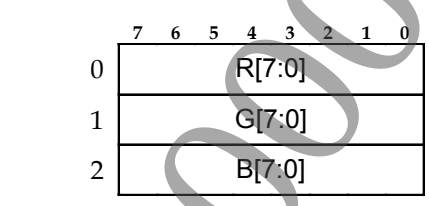


RGB



Where R is the unsigned 8-bit red, G is the unsigned 8-bit green, and B is the unsigned 8-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 89 are supported.

The modes of operation for this format are shown in the following table:

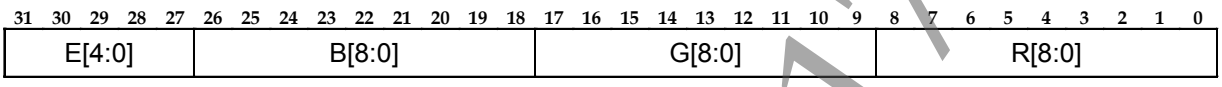
Table 92 U8U8U8 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE3_BGR SCE_GXM_COLOR_SWIZZLE3_RGB	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components are gamma correct before being written to memory.	Supported	Supported

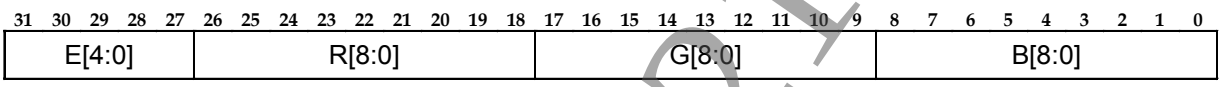
SE5M9M9M9

The SE5M9M9M9 color surface format (SCE_GXM_COLOR_BASE_FORMAT_SE5M9M9M9) occupies 4 bytes in memory and can have the following 32-bit representations:

BGR

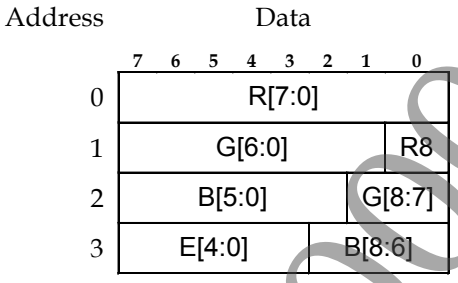


RGB



Where E is the shared 5 bit exponent, R is the partial-precision floating-point 9-bit red mantissa, G is the partial-precision floating-point 9-bit green mantissa, and B is the partial-precision floating-point 9-bit blue mantissa values. There is no sign bit.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:



The swizzle modes as shown in Table 89 are supported.

The modes of operation for this format are shown in the following table:

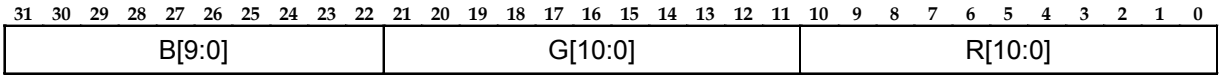
Table 93 SE5M9M9M9 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE3_BGR SCE_GXM_COLOR_SWIZZLE3_RGB	• half4	Not supported	Supported

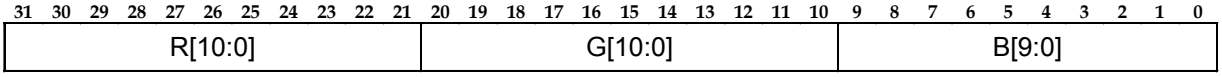
F11F11F10

The F11F11F10 color surface format (SCE_GXM_COLOR_BASE_FORMAT_F11F11F10) occupies 4 bytes in memory and can have the following 32-bit representations.

BGR

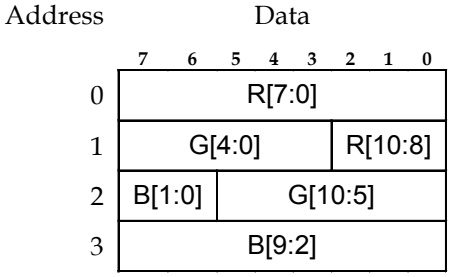


RGB

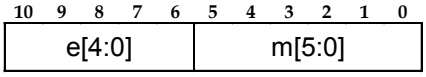


Where R is the partial-precision floating-point 11-bit red, G is the partial-precision floating-point 11-bit green, and B is the partial-precision floating-point 10-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **BGR** representation would be stored as:

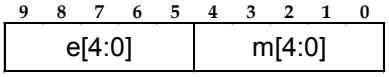


The 11-bit red and green values are interpreted as:



Where e is the 5-bit exponent and m is the 6-bit mantissa. There is no sign bit.

The 10-bit blue value is interpreted as:



Where e is the 5-bit exponent and m is the 5-bit mantissa. There is no sign bit.

The swizzle modes as shown in Table 89 are supported.

The modes of operation for this format are shown in the following table:

Table 94 F11F11F10 Output Formats

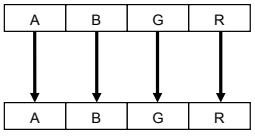
Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE3_BGR SCE_GXM_COLOR_SWIZZLE3_RGB	• half4	Not supported	Supported

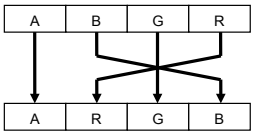
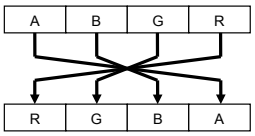
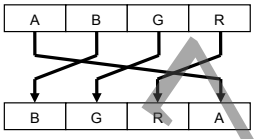
Four Component Color Surface Formats

Four Component Swizzle Modes

Unless otherwise documented in the color format description, the values from the shader in ABGR component ordering are stored to memory according to the swizzle mode shown in Table 95.

Table 95 Four Component Color Surface Swizzle Modes

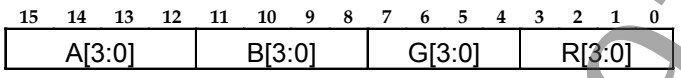
Mode	ABGR Representation
SCE_GXM_COLOR_SWIZZLE4_ABGR	

Mode	ABGR Representation
SCE_GXM_COLOR_SWIZZLE4_ARGB	
SCE_GXM_COLOR_SWIZZLE4_RGBA	
SCE_GXM_COLOR_SWIZZLE4_BGRA	

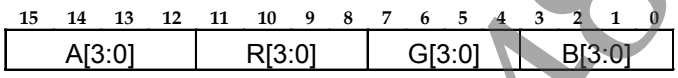
U4U4U4U4

The U4U4U4U4 color surface format (`SCE_GXM_COLOR_BASE_FORMAT_U4U4U4U4`) occupies 2 bytes in memory and can have the following 16-bit representations.

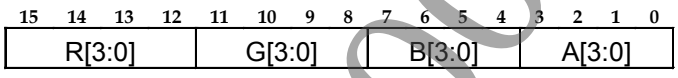
ABGR



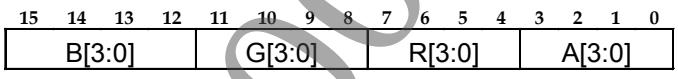
ARGB



RGBA

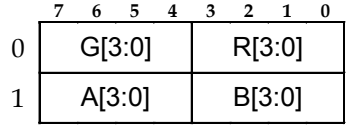


BGRA



All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

Address Data



The swizzle modes as shown in Table 95 are supported.

The modes of operation for this format are shown in the following table:

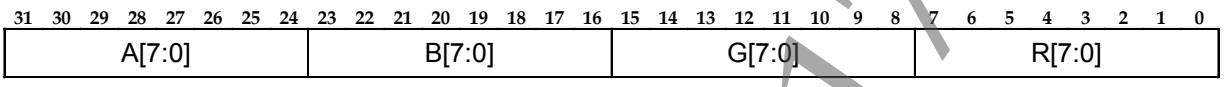
Table 96 U4U4U4U4 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE4_ABGR SCE_GXM_COLOR_SWIZZLE4_ARGB SCE_GXM_COLOR_SWIZZLE4_RGBA SCE_GXM_COLOR_SWIZZLE4_BGRA	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components are gamma correct before being written to memory.	Supported	Supported

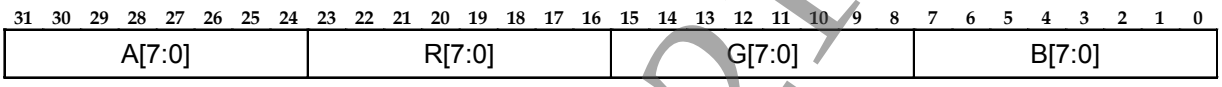
U8U8U8U8

The U8U8U8U8 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U8U8U8U8) occupies 4 bytes in memory and can have the following 32-bit representations.

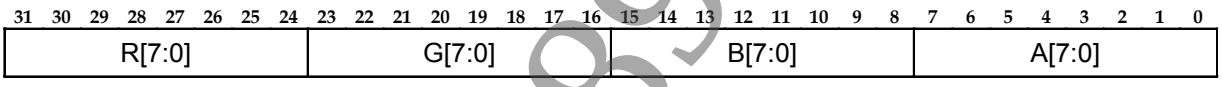
ABGR



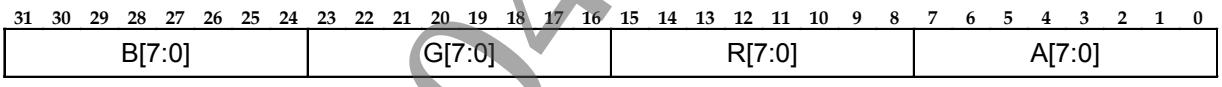
ARGB



RGBA



BGRA



Where A is interpreted as the unsigned 8-bit alpha, R is the unsigned 8-bit red, G is the unsigned 8-bit green, and B is the unsigned 8-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

The swizzle modes as shown in Table 95 are supported.

The modes of operation for this format are shown in the following table:

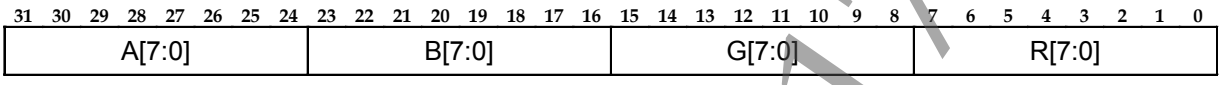
Table 97 U8U8U8U8 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE4_ABGR SCE_GXM_COLOR_SWIZZLE4_ARGB SCE_GXM_COLOR_SWIZZLE4_RGBA SCE_GXM_COLOR_SWIZZLE4_BGRA	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components are gamma correct before being written to memory.	Supported	Supported

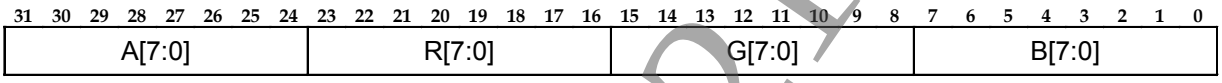
S8S8S8S8

The S8S8S8S8 color surface format (SCE_GXM_COLOR_BASE_FORMAT_S8S8S8S8) occupies 4 bytes in memory and can have the following 32-bit representations.

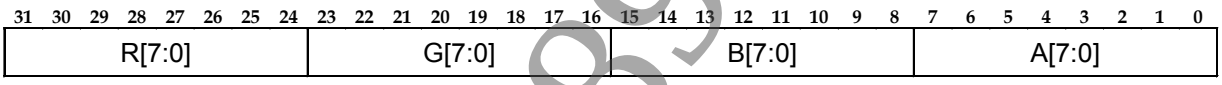
ABGR



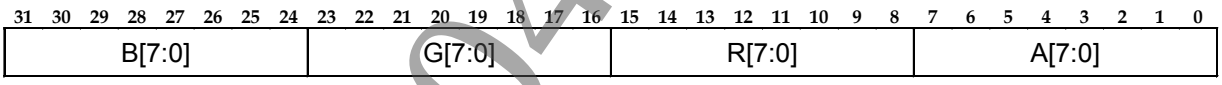
ARGB



RGBA



BGRA



Where A is interpreted as the signed 8-bit alpha, R is the signed 8-bit red, G is the signed 8-bit green, and B is the signed 8-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

The swizzle modes as shown in Table 95 are supported.

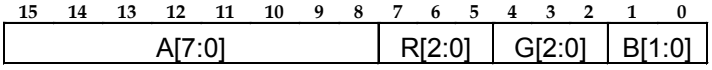
The modes of operation for this format are shown in the following table:

Table 98 S8S8S8S8 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE4_ABGR SCE_GXM_COLOR_SWIZZLE4_ARGB SCE_GXM_COLOR_SWIZZLE4_RGBA SCE_GXM_COLOR_SWIZZLE4_BGRA	• signed char4	Not supported	Supported

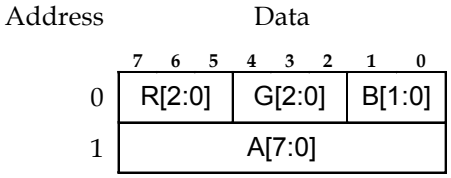
U8U3U3U2

The U8U3U3U2 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U8U3U3U2) occupies 2 bytes in memory and has the following 16-bit representation.



Where A is interpreted as the unsigned 8-bit alpha, R is the unsigned 3-bit red, G is the unsigned 3-bit green, and B is the unsigned 2-bit blue component values.

This would be stored in memory as follows:



The U8U3U3U2 format does not support swizzle mode and always uses this fixed bit layout.

The modes of operation for this format are shown in the following table:

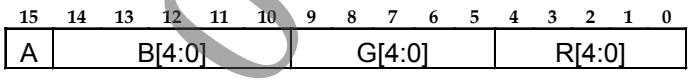
Table 99 U8U3U3U2 Output Formats

Supported Output Formats	Dither	Downscale
• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components are gamma correct before being written to memory.	Supported	Supported

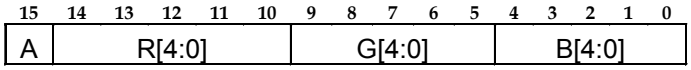
U1U5U5U5

The U1U5U5U5 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U1U5U5U5) occupies 2 bytes in memory and can have the following 16-bit representations:

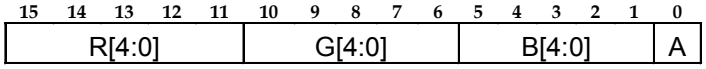
ABGR



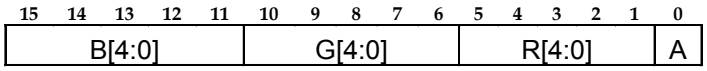
ARGB



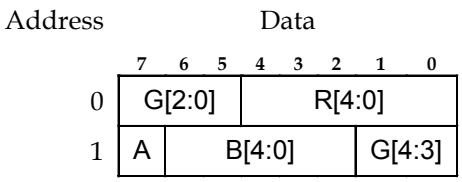
RGBA



BGRA



All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:



The swizzle modes as shown in Table 95 are supported.

The modes of operation for this format are shown in the following table:

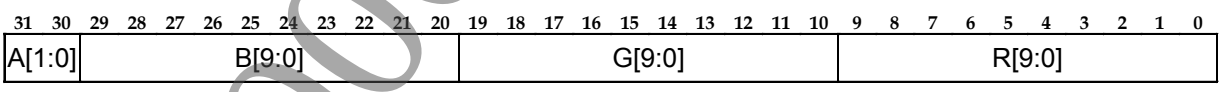
Table 100 U1U5U5U5 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE4_ABGR SCE_GXM_COLOR_SWIZZLE4_ARGB SCE_GXM_COLOR_SWIZZLE4_RGBA SCE_GXM_COLOR_SWIZZLE4_BGRA	• unsigned char4 • half4 - When gamma correction is enabled, the R, G and B color components are gamma correct before being written to memory.	Supported	Supported

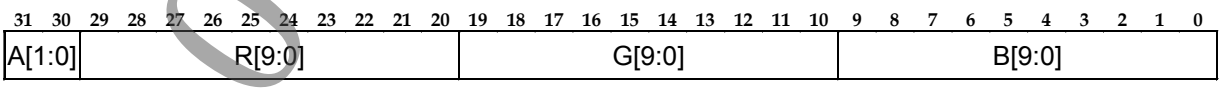
U2U10U10U10

The U2U10U10U10 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U2U10U10U10) occupies 4 bytes in memory and can have the following 32-bit representations depending on the swizzle mode:

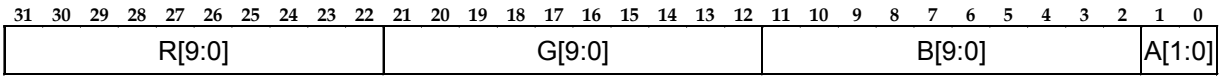
ABGR



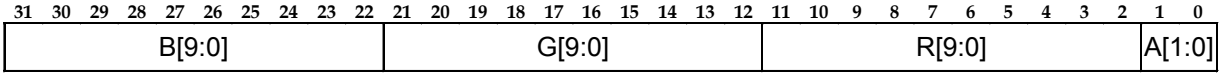
ARGB



RGBA

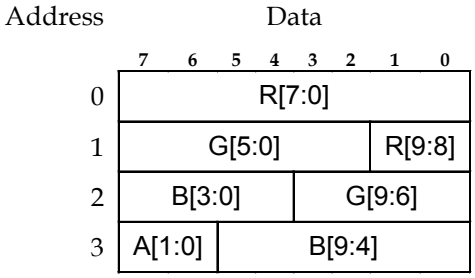


BGRA



Where A is interpreted as the unsigned 2-bit alpha, R is the unsigned 10-bit red, G is the unsigned 10-bit green, and B is the unsigned 10-bit blue component values.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:



The swizzle modes as shown in Table 95 are supported.

The modes of operation for this format are shown in the following table:

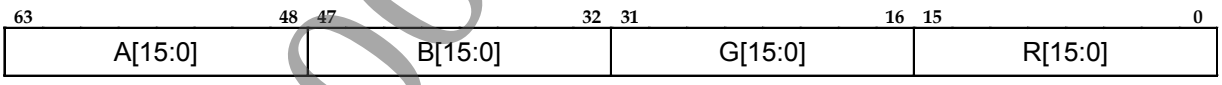
Table 101 U2U10U10U10 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE4_ABGR SCE_GXM_COLOR_SWIZZLE4_ARGB SCE_GXM_COLOR_SWIZZLE4_RGBA SCE_GXM_COLOR_SWIZZLE4_BGRA	• half4	Supported	Supported

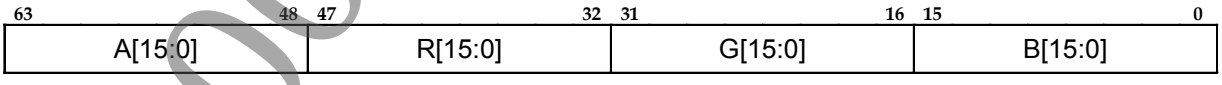
F16F16F16F16

The F16F16F16F16 color surface format (SCE_GXM_COLOR_BASE_FORMAT_F16F16F16F16) occupies 8 bytes in memory and can have the following 64-bit representations.

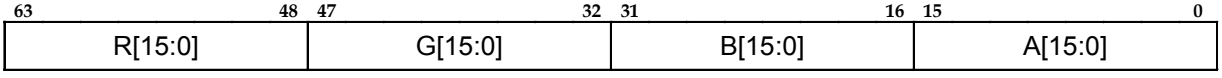
ABGR



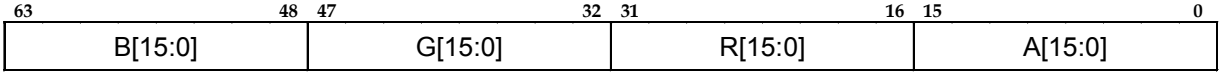
ARGB



RGBA



BGRA



Where A is interpreted as the 16-bit half precision floating-point alpha, R is the 16-bit half precision floating-point red, G is the 16-bit half precision floating-point green, and B is the 16-bit half precision floating-point blue component values.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:

Address	Data (7 6 5 4 3 2 1 0)
0	R[7:0]
1	R[15:8]
2	G[7:0]
3	G[15:8]
4	B[7:0]
5	B[15:8]
6	A[7:0]
7	A[15:8]

The swizzle modes as shown in Table 95 are supported.

The modes of operation for this format are shown in the following table:

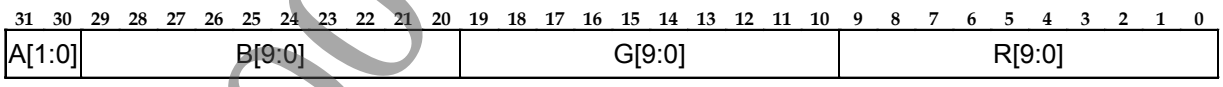
Table 102 F16F16F16F16 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE4_ABGR SCE_GXM_COLOR_SWIZZLE4_ARGB SCE_GXM_COLOR_SWIZZLE4_RGBA SCE_GXM_COLOR_SWIZZLE4_BGRA	• half4	Not supported	Supported

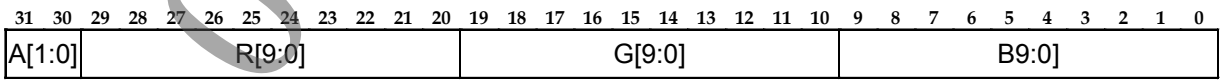
U2F10F10F10

The U2F10F10F10 color surface format (SCE_GXM_COLOR_BASE_FORMAT_U2F10F10F10) occupies 4 bytes in memory and can have the following 32-bit representations.

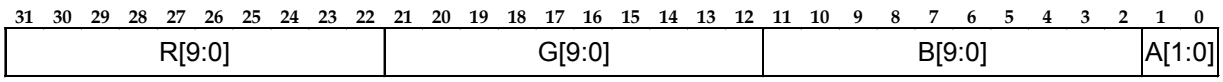
ABGR



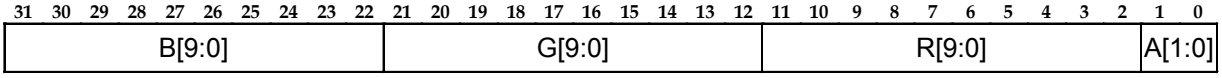
ARGB



RGBA

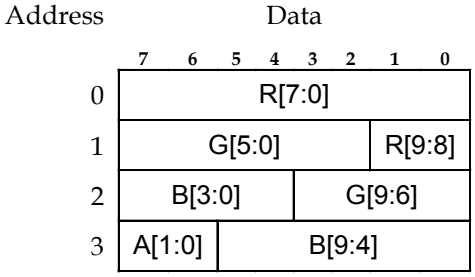


BGRA

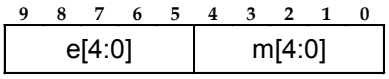


Where R is the partial-precision floating-point 10-bit red, G is the partial-precision floating-point 10-bit green, and B is the partial-precision floating-point 10-bit blue component values. A is the unsigned 2-bit alpha value.

All of these representations are stored little-endian in memory. For example, the **ABGR** representation would be stored as:



The 10-bit red, green and blue values are interpreted as:



Where e is the 5-bit exponent and m is the 5-bit mantissa. There is no sign bit.

The swizzle modes as shown in Table 95 are supported.

The modes of operation for this format are shown in the following table:

Table 103 U2F10F10F10 Output Formats

Swizzle Mode	Supported Output Formats	Dither	Downscale
SCE_GXM_COLOR_SWIZZLE4_ABGR SCE_GXM_COLOR_SWIZZLE4_ARGB SCE_GXM_COLOR_SWIZZLE4_RGBA SCE_GXM_COLOR_SWIZZLE4_BGRA	• half4	Not supported	Supported

11 Depth and Stencil Surfaces

Types

The depth and stencil surfaces can be loaded and stored in different memory layouts. The depth and stencil surface memory layouts have the same layout as pixel data.

The following depth and stencil surface memory layout types are supported:

Table 104 List of Supported Depth and Stencil Surface Types

Type	Notes
SCE_GXM_DEPTH_STENCIL_SURFACE_LINEAR	Uses linear memory layout with pixels stored in scanline order.
SCE_GXM_DEPTH_STENCIL_SURFACE_TILED	Uses tiled memory layout.

Depth and Stencil Surface Size

The maximum depth and stencil surface sizes are 4096 values in width and height.

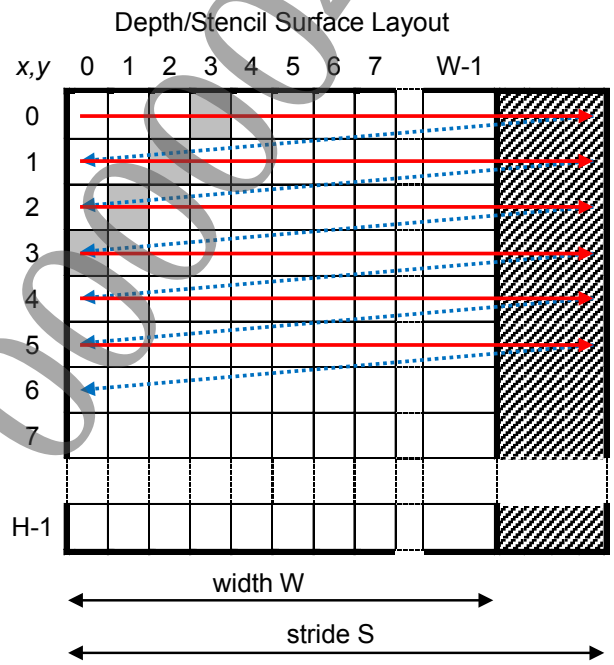
Memory Layouts

Linear Depth and Stencil Surface

Data is stored linearly from left to right in scan line format.

The width and height of the depth and stencil surfaces do not need to be an exact multiple of 32. However, the stride width must be a whole number of tiles, and the memory footprint must be a whole number of tiles.

Figure 34 Linear Depth and Stencil Surface Memory Layout

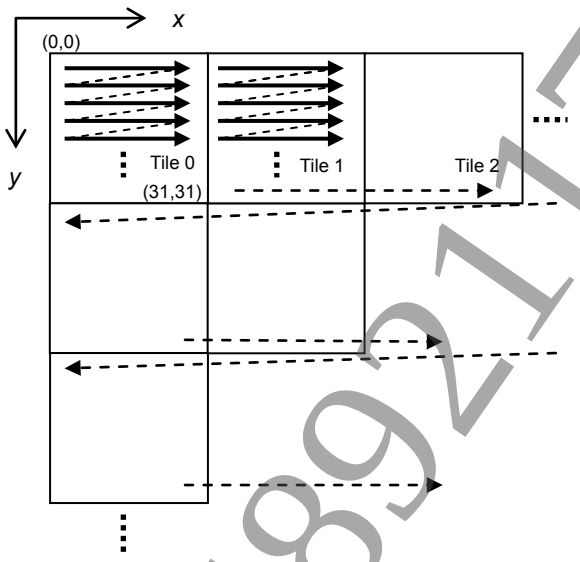


Tiled Depth and Stencil Surface

Tiled depth and stencil surfaces are stored as 32x32 tiles in memory, in scan line sequential order. The values are also stored in scan line sequential order within each tile.

The width and height of the depth and stencil surfaces must be at least 32 pixels but need not be an exact multiple of 32. However, the memory footprint is always a whole number of tiles. So, for example, it is valid for to have a 48x40 tiled depth and stencil surface, but this will have memory footprint of a 64x64 tiled depth and stencil surface since this is the closest multiple of 32 in width and height.

Figure 35 Tiled Depth and Stencil Surface Memory Layout



Depth and Stencil Formats

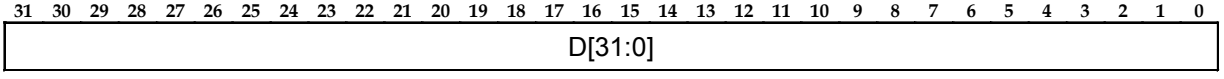
The GPU supports the following depth and stencil formats:

Table 105 List of Supported Depth and Stencil Formats

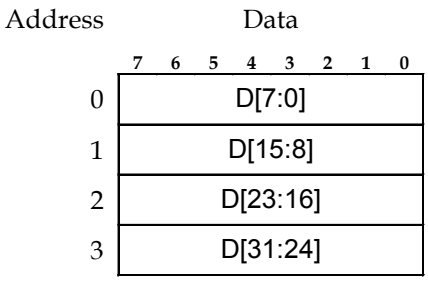
Format	Description
SCE_GXM_DEPTH_STENCIL_FORMAT_DF32	32-bit floating point depth only.
SCE_GXM_DEPTH_STENCIL_FORMAT_S8	8-bit integer stencil only.
SCE_GXM_DEPTH_STENCIL_FORMAT_DF32_S8	Separate 32-bit floating point depth and 8-bit integer stencil.
SCE_GXM_DEPTH_STENCIL_FORMAT_DF32M	32-bit floating point depth with mask in sign bit only.
SCE_GXM_DEPTH_STENCIL_FORMAT_DF32M_S8	32-bit floating point depth with mask in sign bit and 8-bit integer stencil.
SCE_GXM_DEPTH_STENCIL_FORMAT_S8D24	Packed 8-bit integer stencil and 24-bit unsigned integer depth.
SCE_GXM_DEPTH_STENCIL_FORMAT_D16	16-bit unsigned integer depth only.

DF32

The DF32 depth format (SCE_GXM_DEPTH_STENCIL_FORMAT_DF32) occupies 4 bytes in memory and has the following 32-bit representation:



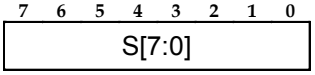
This would be stored in memory as follows:



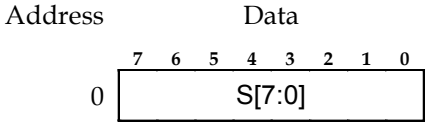
Where D is the 32-bit floating-point depth value.

S8

The S8 stencil format (`SCE_GXM_DEPTH_STENCIL_FORMAT_S8`) occupies 1 byte in memory and has the following 8 bit representation:



This would be stored in memory as follows:



Where S is interpreted as the 8-bit integer stencil value.

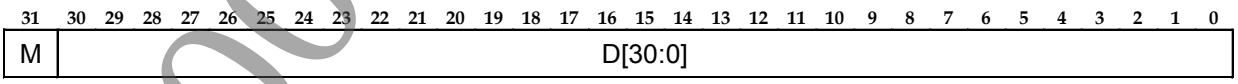
DF32_S8

The DF32_S8 depth and stencil format (`SCE_GXM_DEPTH_STENCIL_FORMAT_DF32_S8`) uses two buffers; a depth buffer using 32-bit floating-point depth values, and a separate stencil buffer using 8-bit integer stencil values.

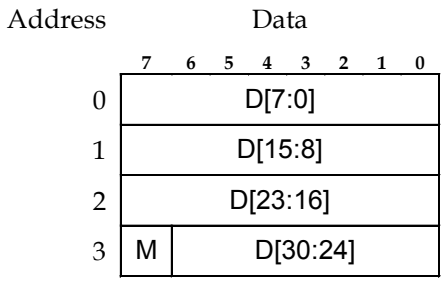
The depth values are represented in the depth buffer in the same way as the DF32 format. The stencil values are represented in the stencil buffer in the same way as the S8 format.

DF32M

The DF32M depth format (`SCE_GXM_DEPTH_STENCIL_FORMAT_DF32M`) occupies 4 bytes in memory and has the following 32-bit representation:



This would be stored in memory as follows:



Where D is the 32-bit floating-point depth value (with implicit 0 sign bit) and M is the mask bit.

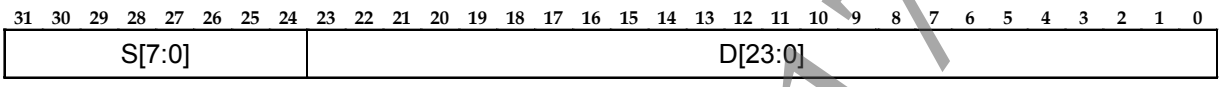
DF32M_S8

The DF32M_S8 depth and stencil format (`SCE_GXM_DEPTH_STENCIL_FORMAT_DF32M_S8`) uses two buffers; a depth buffer using 32-bit floating-point depth values with mask in the sign bit, and a separate stencil buffer using 8-bit integer stencil values.

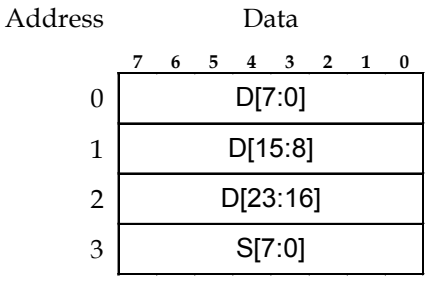
The depth values are represented in the depth buffer in the same way as the DF32M format. The stencil values are represented in the stencil buffer in the same way as the S8 format.

S8D24

The S8D24 depth and stencil format (`SCE_GXM_DEPTH_STENCIL_FORMAT_S8D24`) occupies 4 bytes in memory and has the following 32-bit representation:



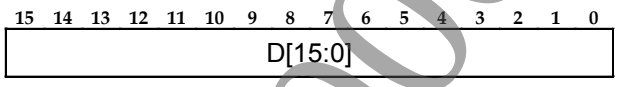
This would be stored in memory as follows:



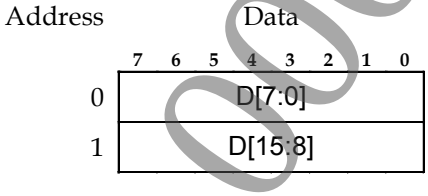
Where D is the 24-bit unsigned integer depth value, and S is the 8-bit integer stencil value.

D16

The D16 depth format (`SCE_GXM_DEPTH_STENCIL_FORMAT_D16`) occupies 2 bytes in memory and has the following 16-bit representation:



This would be stored in memory as follows:



Where D is the 16-bit unsigned integer depth value.

12 PTLA Transfers

Types

Transfer data can be stored in different memory layouts. The transfer data memory types have layouts that are compatible with those used by both texture and color surfaces.

The following transfer data memory layout types are supported:

Table 106 List of Supported Transfer Data Types

Type	Notes
SCE_GXM_TRANSFER_LINEAR	Uses linear memory layout with pixels stored in scanline order. Stride is provided explicitly, and can be negative.
SCE_GXM_TRANSFER_TILED	Uses tiled memory layout.
SCE_GXM_TRANSFER_SWIZZLED	Uses swizzled memory layout with pixels stored in Morton order.

It is possible to convert between certain memory layouts by specifying different source and destination memory types. The supported conversions are as follows:

Table 107 Supported Conversions

	Linear Destination	Tiled Destination	Swizzled Destination
Linear Source	Y	Y	Y
Tiled Source	Y	Y	N
Swizzled Source	Y	N	N

Transfer Formats

The following transfer data formats are supported on the GPU. These formats are classed as either RGB, YUV or RAW, as follows:

Table 108 List of Supported RGB Transfer Data Formats

Format	Description
SCE_GXM_TRANSFER_FORMAT_U8_R	Unsigned 8-bit.
SCE_GXM_TRANSFER_FORMAT_U4U4U4U4_ABGR	Four unsigned 4-bit..
SCE_GXM_TRANSFER_FORMAT_U1U5U5U5_ABGR	Unsigned 1-bit, unsigned 5-bit, unsigned 5-bit, unsigned 5-bit.
SCE_GXM_TRANSFER_FORMAT_U5U6U5_BGR	Unsigned 5-bit, unsigned 6-bit, unsigned 5-bit.
SCE_GXM_TRANSFER_FORMAT_U8U8_GR	Two unsigned 8-bit.
SCE_GXM_TRANSFER_FORMAT_U8U8U8_BGR	Three unsigned 8-bit.
SCE_GXM_TRANSFER_FORMAT_U8U8U8U8_ABGR	Four unsigned 8-bit.
SCE_GXM_TRANSFER_FORMAT_U2U10U10U10_ABGR	Unsigned 2-bit, unsigned 10-bit, unsigned 10-bit, unsigned 10-bit.

Table 109 List of Supported YUV Transfer Data Formats

Format	Description
SCE_GXM_TRANSFER_FORMAT_VYUY422	Interleaved YUV format (VYUY 2-pixel blocks).
SCE_GXM_TRANSFER_FORMAT_YVYU422	Interleaved YUV format (YVYU 2-pixel blocks).
SCE_GXM_TRANSFER_FORMAT_UYVY422	Interleaved YUV format (UYVY 2-pixel blocks).
SCE_GXM_TRANSFER_FORMAT_YUYV422	Interleaved YUV format (YUYV 2-pixel blocks).

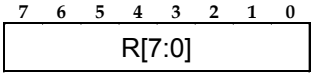
Table 110 List of Supported RAW Transfer Data Formats

Format	Description
SCE_GXM_TRANSFER_FORMAT_RAW16	Raw 16-bit.
SCE_GXM_TRANSFER_FORMAT_RAW32	Raw 32-bit
SCE_GXM_TRANSFER_FORMAT_RAW64	Raw 64-bit.
SCE_GXM_TRANSFER_FORMAT_RAW128	Raw 128-bit.

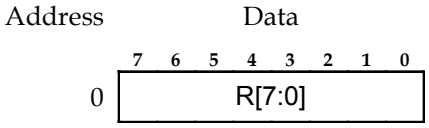
Single Component Transfer Data Formats

U8

The U8 transfer data format (SCE_GXM_TRANSFER_FORMAT_U8_R) occupies 1 byte in memory and has the following representation:

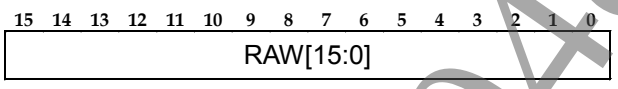


This representation is stored little-endian in memory:



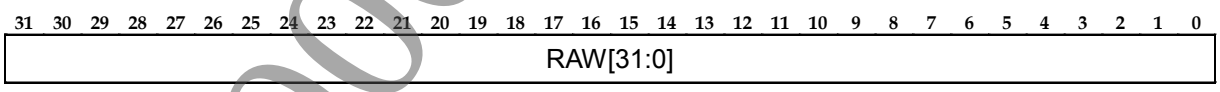
RAW16

The RAW16 transfer data format (SCE_GXM_TRANSFER_FORMAT_RAW16) occupies 2 bytes in memory, and requires 2 byte address and stride alignment. It has the following representation:



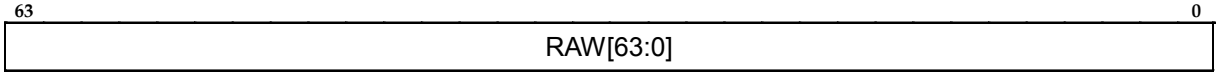
RAW32

The RAW32 transfer data format (SCE_GXM_TRANSFER_FORMAT_RAW32) occupies 4 bytes in memory, and requires 4 byte address and stride alignment. It has the following representation:



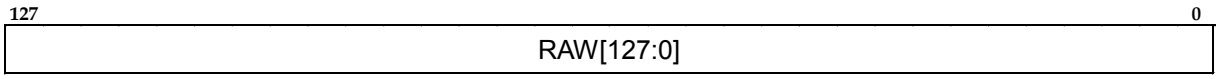
RAW64

The RAW64 transfer data format (SCE_GXM_TRANSFER_FORMAT_RAW64) occupies 8 bytes in memory, and requires 8 byte address and stride alignment. It has the following representation:



RAW128

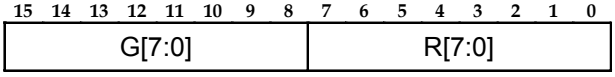
The RAW128 transfer data format (SCE_GXM_TRANSFER_FORMAT_RAW128) occupies 16 bytes in memory, and requires 16 byte address and stride alignment. It has the following representation:



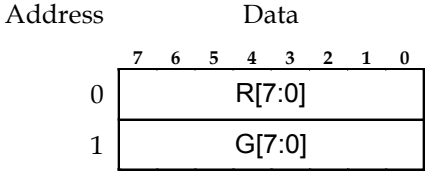
Two Component Transfer Data Formats

U8U8

The U8U8 transfer data format (`SCE_GXM_TRANSFER_FORMAT_U8U8_GR`) occupies 2 bytes in memory, and requires 2 byte address and stride alignment. It has the following representation:



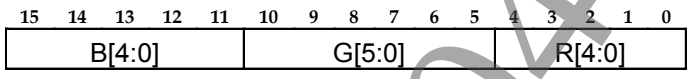
This representation is stored little-endian in memory:



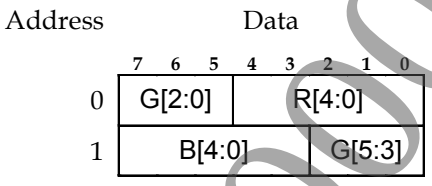
Three Component Transfer Data

U5U6U5

The U5U6U5 transfer data format (`SCE_GXM_TRANSFER_FORMAT_U5U6U5_BGR`) occupies 2 bytes in memory, and requires 2 byte address and stride alignment. It has the following representation:

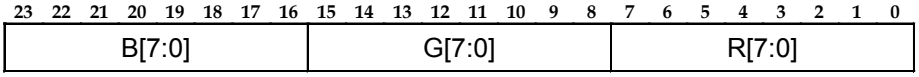


This representation is stored little-endian in memory:

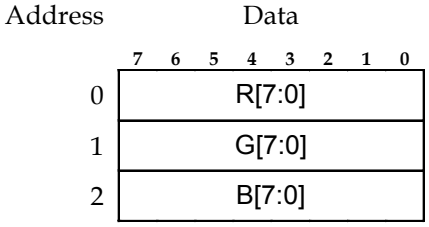


U8U8U8

The U8U8U8 transfer data format (`SCE_GXM_TRANSFER_FORMAT_U8U8U8_BGR`) occupies 3 bytes in memory, and requires 4 byte address alignment and 3 byte stride alignment. It has the following representation:



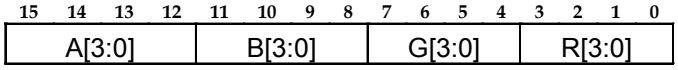
This representation is stored little-endian in memory:



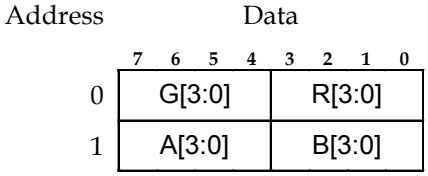
Four Component Transfer Data

U4U4U4U4

The U4U4U4U4 transfer data format (`SCE_GXM_TRANSFER_FORMAT_U4U4U4U4_ABGR`) occupies 2 bytes in memory, and requires 2 byte address and stride alignment. It has the following representation:

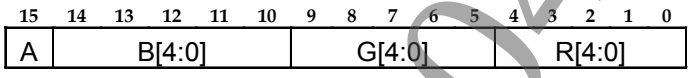


This representation is stored little-endian in memory:

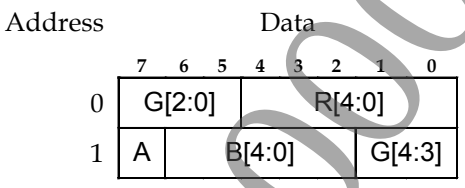


U1U5U5U5

The U1U5U5U5 transfer data format (`SCE_GXM_TRANSFER_FORMAT_U1U5U5U5_ABGR`) occupies 2 bytes in memory, and requires 2 byte address and stride alignment. It has the following representation:

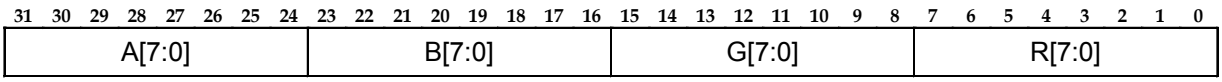


This representation is stored little-endian in memory:



U8U8U8U8

The U8U8U8U8 transfer data format (`SCE_GXM_TRANSFER_FORMAT_U8U8U8U8_ABGR`) occupies 4 bytes in memory, and requires 4 byte address and stride alignment. It has the following representation:



This representation is stored little-endian in memory:

Address	Data (7 6 5 4 3 2 1 0)
0	R[7:0]
1	G[7:0]
2	B[7:0]
3	A[7:0]

VYUY422

The VYUY422 transfer data format (`SCE_GXM_TRANSFER_FORMAT_VYUY422`) occupies 4 bytes in memory, and requires 4 byte address and stride alignment. It has the following representation:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
V[7:0]	Y_1[7:0]	U[7:0]	Y_0[7:0]

This representation is stored little-endian in memory:

Address	Data (7 6 5 4 3 2 1 0)
0	Y_0[7:0]
1	U[7:0]
2	Y_1[7:0]
3	V[7:0]

YVYU422

The YVYU422 transfer data format (`SCE_GXM_TRANSFER_FORMAT_YVYU422`) occupies 4 bytes in memory, and requires 4 byte address and stride alignment. It has the following representation:

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
Y_1[7:0]	V[7:0]	Y_0[7:0]	U[7:0]

This representation is stored little-endian in memory:

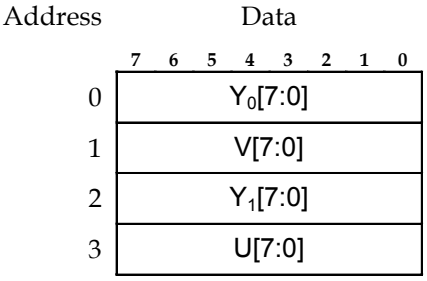
Address	Data (7 6 5 4 3 2 1 0)
0	U[7:0]
1	Y_0[7:0]
2	V[7:0]
3	Y_1[7:0]

UYVY422

The UYVY422 transfer data format (`SCE_GXM_TRANSFER_FORMAT_UYVY422`) occupies 4 bytes in memory, and requires 4 byte address and stride alignment. It has the following representation:

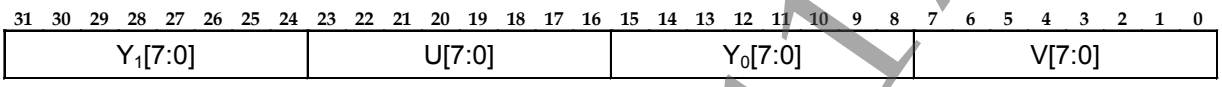
31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
U[7:0]	Y_1[7:0]	V[7:0]	Y_0[7:0]

This representation is stored little-endian in memory:

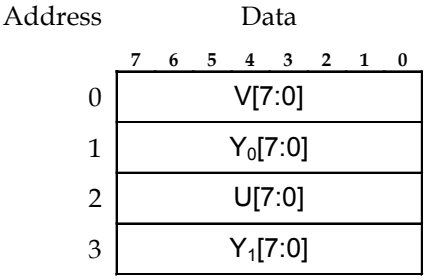


YUYV422

The YUYV422 transfer data format (`SCE_GXM_TRANSFER_FORMAT_YUYV422`) occupies 4 bytes in memory, and requires 4 byte address and stride alignment. It has the following representation:

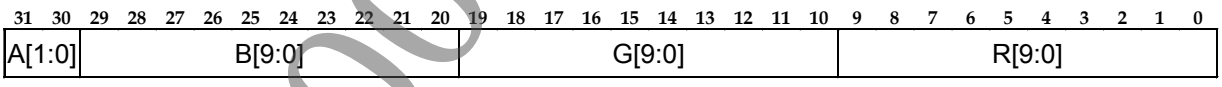


This representation is stored little-endian in memory:

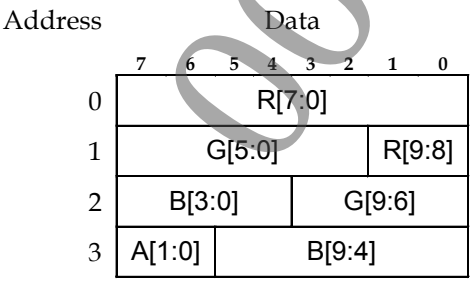


U2U10U10U10

The U2U10U10U10 transfer data format (`SCE_GXM_TRANSFER_FORMAT_U2U10U10U10_ABGR`) occupies 4 bytes in memory, and requires 4 byte address and stride alignment. It has the following representation:



This representation is stored little-endian in memory:



Format Conversion

The transfer hardware supports format conversion of data during copy and downscale, when both source and destination formats are of the same fundamental type (that is, both are RGB, or both are YUV).

Conversion between RAW types is not supported. For YUV formats the conversion simply reorders the Y, U, and V components without any additional modification.

For RGB format conversion the source data is converted to 8-bit internally by shifting such that it occupies the most significant bits of an 8-bit value. The least significant bits are then formed by replicating the most significant bits of the source data. This 8-bit internal data is then converted to the destination format by shifting and truncating the 8-bit data as appropriate.

Color Keying

It is possible to apply a color keying operation during copy of RGB and YUV data to pass or reject source data. The source data is ANDed with the color key mask, and then compared with the color key value. If this comparison is true, then the source data will either be passed or rejected depending on the color key mode. The format of the color key value and mask is as follows:

Table 111 Color Key Mask Operations

Source/Destination Format	Color Key Mask/Value Layout
RGB formats • SCE_GXM_TRANSFER_FORMAT_U8_R • SCE_GXM_TRANSFER_FORMAT_U8U8_GR • SCE_GXM_TRANSFER_FORMAT_U5U6U5_BGR • SCE_GXM_TRANSFER_FORMAT_U8U8U8_BGR • SCE_GXM_TRANSFER_FORMAT_U4U4U4U4_ABGR • SCE_GXM_TRANSFER_FORMAT_U1U5U5U5_ABGR • SCE_GXM_TRANSFER_FORMAT_U8U8U8U8_ABGR If only one of source or destination format is SCE_GXM_TRANSFER_FORMAT_U2U10U10U10_ABGR, then color keying is not supported.	SCE_GXM_TRANSFER_FORMAT_U8U8U8U8_ABGR, with source data expanded as described for format conversion.
Both source and destination formats are SCE_GXM_TRANSFER_U2U10U10U10_ABGR.	SCE_GXM_TRANSFER_FORMAT_U2U10U10U10_ABGR
YUV formats • SCE_GXM_TRANSFER_FORMAT_VYUY422 • SCE_GXM_TRANSFER_FORMAT_YVYU422 • SCE_GXM_TRANSFER_FORMAT_UYVY422 • SCE_GXM_TRANSFER_FORMAT_YUYV422	Layout of destination format.
RAW formats • SCE_GXM_TRANSFER_FORMAT_RAW16 • SCE_GXM_TRANSFER_FORMAT_RAW32 • SCE_GXM_TRANSFER_FORMAT_RAW64 • SCE_GXM_TRANSFER_FORMAT_RAW128	Not supported

Negative Strides

When performing transfer operations on data with a type of SCE_GXM_TRANSFER_LINEAR, it is possible to provide a stride that is negative. Using this, it is possible to copy or downscale data while inverting in the Y axis. When using negative strides, care must be taken to ensure that the address is pointing to the start of the first row that will be processed, and that the Y offset is considered as being in the direction of the stride. The hardware itself computes the starting address of a row from the transfer address supplied, as follows:

```
row_start_address = transfer_address + (y * stride)
```

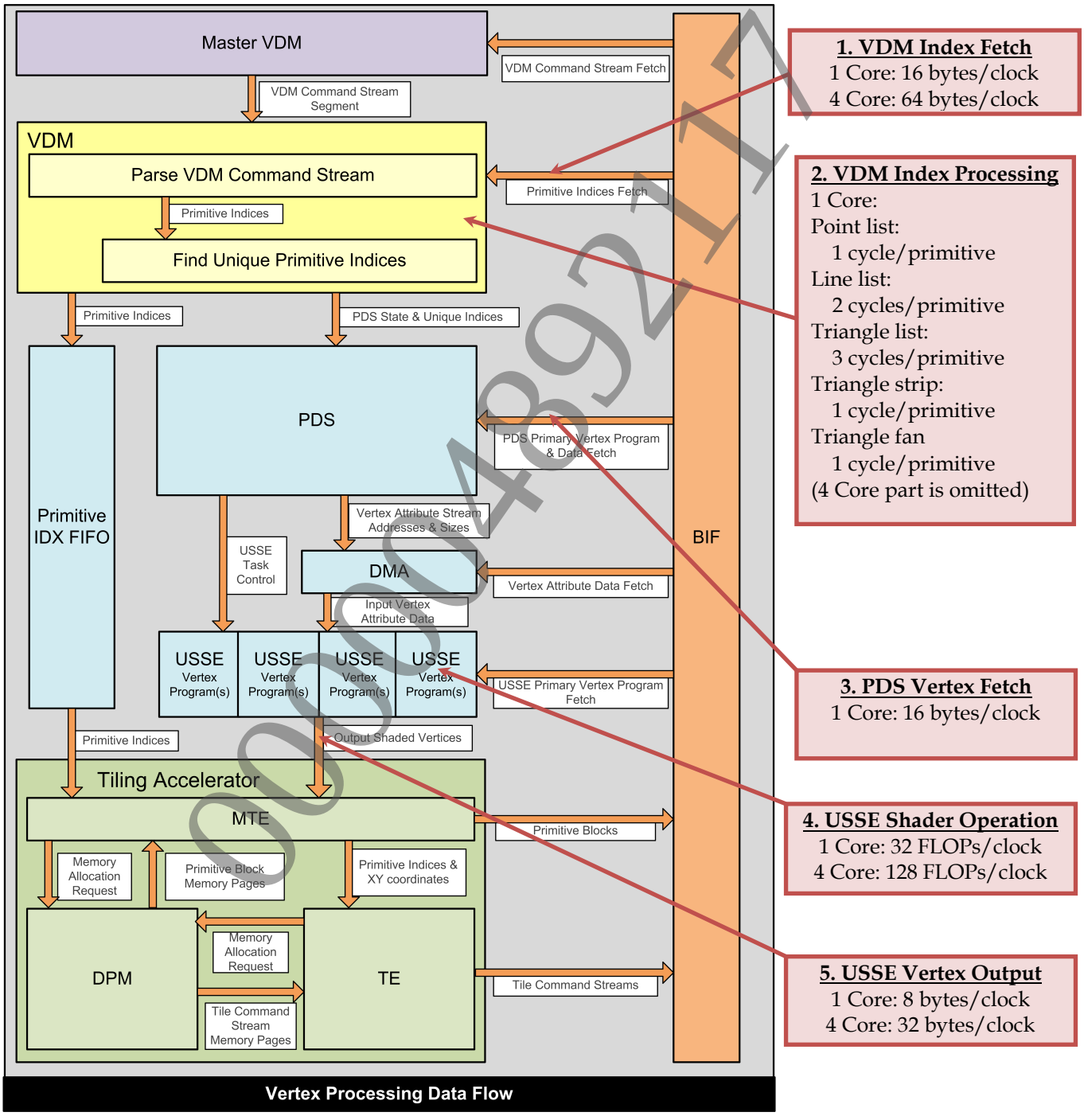
13 Theoretical Maximum Performance

This chapter provides theoretical maximum performance of the SGX543MP4+ on a per-block basis. Please refer to Chapter 14 for basic checkpoints for efficient GPU programming.

Note that the clock speed of PlayStation®Vita GPU is currently not exposed.

Vertex Pipeline

Figure 36 Theoretical Maximum Performance in Vertex Pipeline



1. VDM Index Fetch

Performance

1 Core	16 bytes/clock
4 Core	64 bytes/clock

Description

This is the performance that the VDM fetches index data from memory. The PlayStation®Vita GPU can process 16-bit and 32-bit indices. However, because 32-bit indices take longer to process within the PDS, you might not be able to get this peak performance.

2. VDM Index Processing

Performance

1 Core	Point list	1 cycle/primitive
	Line list	2 cycles/primitive
	Triangle list	3 cycles/primitive
	Triangle strip	1 cycle/primitive
	Triangle fan	1 cycle/primitive

Description

This is the performance required by the VDM to assemble primitives from fetched indices and generate Vertex Shader instances of vertices corresponding to those indices. The table above lists the number of cycles required to assemble each type of primitive (however, 3 cycles are required to assemble the initial triangle in triangle strips and triangle fans).

The VDM contains associative memory (CAM) where processed indices can be stored in advance. When an index of a primitive to be assembled is already stored in CAM, no Vertex Shader instance is generated for the vertex corresponding to that index, and no shading is performed. When all indices forming a primitive are hit in CAM, the index processing cost will be the number of cycles shown above, since the primitive only needs to be assembled. When at least one index was not hit in CAM, further cycles for generating Vertex Shader instances for the other vertices and control words are required in addition to the cycles shown above. In the case when *all* indices required to assemble primitives would not be hit in CAM, a further 3 cycles is required in addition to the cycles shown above.

Also, see "Triangle Strip (TRIANGLE_STRIP) versus Triangle List (TRIANGLES)" for a tip regarding the PlayStation®Vita GPU performance differences between these types of primitives.

3. PDS Vertex Fetch

Performance

1 Core	16 bytes/clock
4 Core	64 bytes/clock

Description

This is the performance that the PDS fetches vertex data from memory. Bus width between DCU L2 and BIF is 128-bit. When data size of a vertex is large and cache efficiency is low, this processing could be a bottleneck. Vertex data could be fetched more efficiently without limitation of this bus width when vertex data hit on DCU.

4. USSE Shader Operation

Performance

1 Core	32 FLOPs/clock
4 Core	128 FLOPs/clock

Description

This is the performance that the USSE processes shader operations. The PlayStation®Vita GPU contains four USSE pipes, and each pipe can execute a SIMD instruction per clock for 4-components operands. Considering that ADD and MUL instruction is 1 FLOP (floating operation) and MAD (multiply add operation) instruction is 2 FLOPs, this theoretical performance is worked out as below.

4 (USSE pipes) × 4 (components operand) × 2 (FLOPs) = 32 FLOPs/clock

Details of USSE shader instructions will be disclosed in the future.

5. USSE Vertex Output

Performance

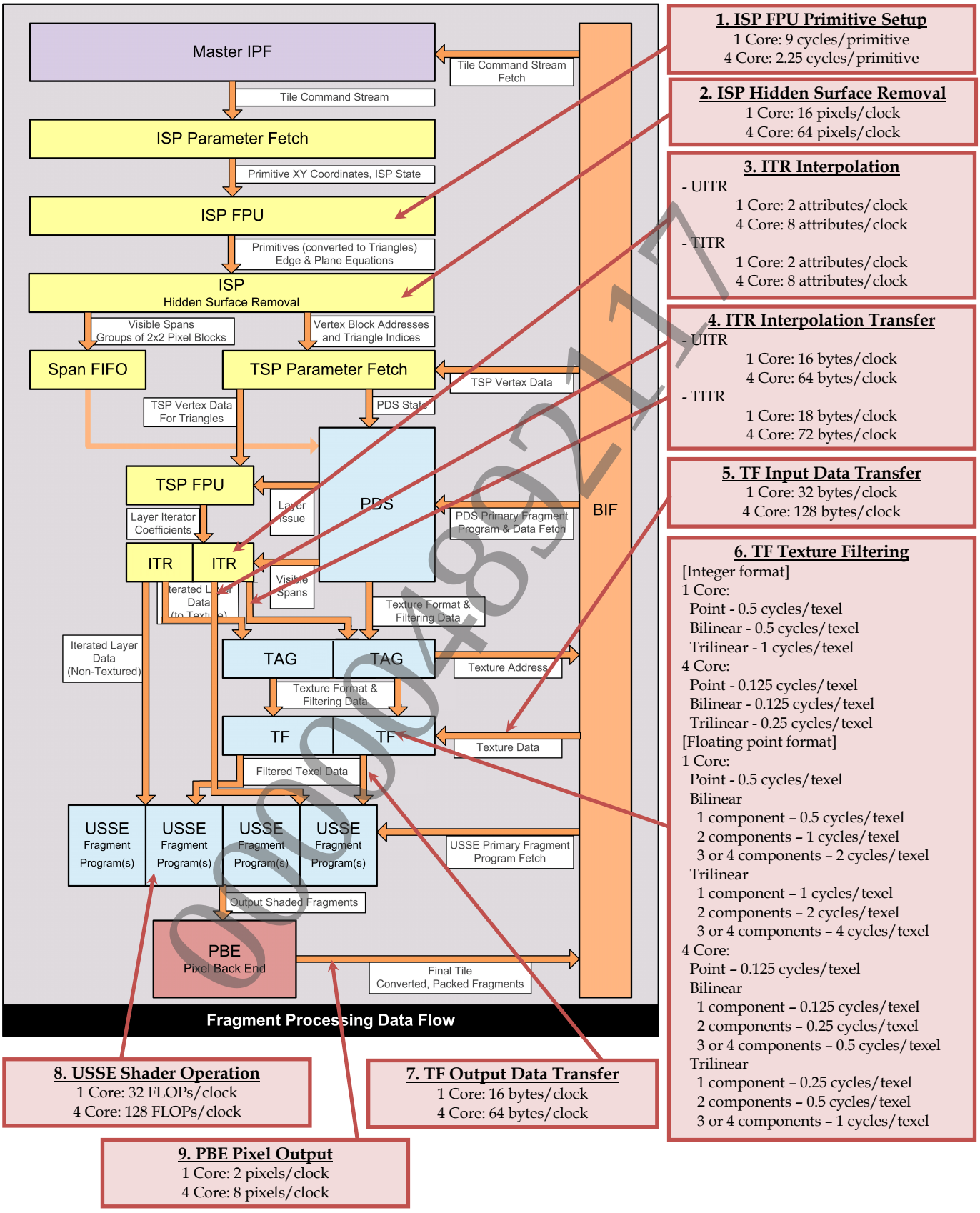
1 Core	8 bytes/clock
4 Core	32 bytes/clock

Description

This is the performance of bandwidth which the USSE pipes transport shaded vertex attribute data to the MTE. The bus width between the USSE and the MTE is 64-bit. The MTE reads vertex data which correspond to the indices queued in VDM FIFO. At this time, MTE cannot access to multiple USSE pipes in parallel.

Fragment Pipeline

Figure 37 Theoretical Maximum Performance in Fragment Pipeline



1. ISP FPU Primitive Setup

Performance

1 Core	9 cycles/primitive
4 Core	2.25 cycles/primitive

Description

This is the performance that the ISP FPU performs primitive setup. If small primitives are rendered with light-weight shader, this processing could be a bottleneck. The performance is the same for all primitive types.

2. ISP Hidden Surface Removal

Performance

1 Core	16 pixels/clock
4 Core	64 pixels/clock

Description

This is the performance at which the ISP performs depth/stencil/mask test for hidden surface removal. Note that when alpha test (DISCARD) or depth replacement is performed, it is necessary to execute some fragment shader operations before hidden surface removal is performed. The use of blending and color write masking also reduce the effectiveness of hidden surface removal.

3. ITR Interpolation

Performance

1 Core	UITR	2 attributes/clock
	TITR	2 attributes/clock
4 Core	UITR	8 attributes/clock
	TITR	8 attributes/clock

Description

This is the performance that the ITR calculates colors and texture coordinates of pixels based on coefficients that are transported from the TSP FPU. ITR contains 2 sub units: TITR and UITR. The TITR is responsible for the interpolation of texture coordinates forwarded to TAG; that is, for independent texture reads. The UITR is responsible for the interpolation of all attributes directly forwarded to USSE, such as, colors and texture coordinates for dependent texture reads. There are 2 execution pipes within the TITR or the UITR. Note that 1 attribute indicates up to 4-components vector and this performance does not depend on component counts and data type of an attribute. Therefore, when this processing causes a bottleneck, packing multiple attributes in one vector might be able to resolve the issue.

4. ITR Interpolation Transfer

Performance

1 Core	UITR	16 bytes/clock
	TITR	18 bytes/clock
4 Core	UITR	64 bytes/clock
	TITR	72 bytes/clock

Description

This is the performance that the ITR transports interpolated colors or texture coordinates to the USSE or the TAG. Note that the bus width between the TITR and the TAG is different from the bus width between the UITR and the USSE.

- UITR – USSE: The bus width is 64-bit. Each component of an attribute is processed as 24-bit float format (s1e8m15: 1-bit sign, 8-bit exponent, 15-bit mantissa), and forwarded as 32-bit float format (s1e8m23: 1-bit sign, 8-bit exponent, 23-bit mantissa). That is, the lower 8 bits of the 32-bit UITR output are always zero.
- TITR – TAG: There are three 24-bit buses. Each component of a texture coordinate is processed and forward as 24-bit float format.

When data size of attributes forwarded to the USSE is larger than the bus width, this processing could be a bottleneck. When this limitation of the bus width causes a bottleneck, the following methods might resolve the issue:

- Downgrading the precision of varying attributes from single precision floating-point (32-bit) to half precision floating-point (16-bit).
- Do not output the components of a varying attribute which are not used in fragment shader.

This optimization also contributes to reduction of memory usage for parameter buffer.

5. TF Input Data Transfer

Performance

1 Core	32 bytes/clock
4 Core	128 bytes/clock

Example: U8U8U8U8

1 Core	Point	0.5 cycles/texel
	Bilinear	0.5 cycles/texel
	Trilinear	1 cycles/texel
4 Core	Point	0.125 cycles/texel
	Bilinear	0.125 cycles/texel
	Trilinear	0.25 cycles/texel

Example: F16F16F16F16

1 Core	Point	1 cycles/texel
	Bilinear	1 cycles/texel
	Trilinear	2 cycles/texel
4 Core	Point	0.25 cycles/texel
	Bilinear	0.25 cycles/texel
	Trilinear	0.5 cycles/texel

Description

This is the performance of data transfer from TCU to TF. Within a single SGX Core, there are two TCU L0 blocks and TF blocks. Between a TCU L0 block and a TF block, there are four 32-bit buses, and only one 32-bit bus is used for point sampling, and all four 32-bit buses are used for bilinear or trilinear filtering.

The examples above show the cycles to transfer all texture data that are necessary to create a single filtered texel (The numbers of texture samples necessary for each of point, bilinear and trilinear filtering are 1, 4 and 8.)

Note that if you use 64-bit format such as F16F16F16F16 and point filtering, the number of cycles required for the input transfer will be greater than the number of cycles required for the filtering operation (Refer to the next section for the detailed performance of Filtering Operation.)

Also, Three-component floating point texture formats such as F11F11F10 and SE5M9M9M9 are processed in the same manner as F16F16F16F16. So, though their actual bit-depth is 32-bit, they will take the same cycles as 64-bit format.

6. TF Filtering Operation

Performance

Integer format

1 Core	Point	0.5 cycles/texel
	Bilinear	0.5 cycles/texel
	Trilinear	1 cycles/texel
4 Core	Point	0.125 cycles/texel
	Bilinear	0.125 cycles/texel
	Trilinear	0.25 cycles/texel

Floating point format

		Number of components			
		1	2	3	4
1 Core	Point	0.5 cycles/texel			
	Bilinear	0.5 cycles/texel	1 cycles/texel	2 cycles/texel	
	Trilinear	1 cycles/texel	2 cycles/texel	4 cycles/texel	
4 Core	Point	0.125 cycles/texel			
	Bilinear	0.125 cycles/texel	0.25 cycles/texel	0.5 cycles/texel	
	Trilinear	0.25 cycles/texel	0.5 cycles/texel	1 cycles/texel	

Example: U8U8U8U8

1 Core	Point	0.5 cycles/texel
	Bilinear	0.5 cycles/texel
	Trilinear	1 cycles/texel
4 Core	Point	0.125 cycles/texel
	Bilinear	0.125 cycles/texel
	Trilinear	0.25 cycles/texel

Example: F16F16F16F16

1 Core	Point	0.5 cycles/texel
	Bilinear	2 cycles/texel
	Trilinear	4 cycles/texel
4 Core	Point	0.125 cycles/texel
	Bilinear	0.5 cycles/texel
	Trilinear	1 cycles/texel

Description

This is the performance of texture filtering. Each filtering takes the number of cycles shown above to produce filtered texels. The performance can decline when there is no texture data hit in the TCU and the latency of memory accesses becomes visible.

Note that the maximum performance of floating point texture is different from that of integer texture. If you use bilinear or trilinear filtering on floating point texture, necessary cycles will change depending on the number of components in the texture format. Three-component texture formats such as F11F11F10 and SE5M9M9M9 are processed in the same manner as four-component floating point textures such as F16F16F16F16. As shown in the examples above, bilinear or trilinear filtering using F16F16F16F16 format requires four times as many cycles as compared to the U8U8U8U8 format.

If gamma correction is enabled, gamma correction operation is applied to each texture sample before texture filtering and F16 (16-bit float, s1e5m10: 1-bit sign, 5-bit exponent, 10-bit mantissa) is always used

for texture filtering for each component regardless of the texture format itself, so the performance may deteriorate.

7. TF Output Data Transfer

Performance

1 Core	16 bytes/clock
4 Core	64 bytes/clock

Description

This is the performance of the transfer of filtered texel data from TF to USSE.

There are two TFs in one core. Each TF transfers texel data to two USSEs. The bus width between one TF and one USSE is 64-bit. One TF can only transfer data to one USSE in one cycle, so the above performance is obtained with 64-bit x 2 TFs x 4 cores.

8. USSE Shader Operation

Performance

1 Core	32 FLOPs/clock
4 Core	128 FLOPs/clock

Description

Fragment shader FLOP performance is the same as vertex shader FLOP performance. Please refer to the vertex pipeline section for further details.

9. PBE Pixel Output

Performance

1 Core	2 pixels/clock
4 Core	8 pixels/clock

Description

This is the performance that the PBE writes pixel data to memory. The PBE reads shaded pixel data from USSE pipes and writes to memory after converting the format specified by the user. Note that this performance does not depend on the format of the output register or color buffer. However, it may not be possible to obtain this theoretical performance for the following reasons:

(1) MSAA

When MSAA is enabled, sample data for generating the final pixel can no longer be read from the unified store in one block except for a combination of 32-bit output register format and 2x mode, therefore throughput will decrease.

(2) Gamma correction

Because performance can vary when gamma correction is performed, it should be disabled unless it is needed. Details will be disclosed in the future.

(3) Unified store bank conflict

When the 64-bit output register format is used and MSAA is enabled, the amount of data read by PBE will increase, so bank conflicts will occur more frequently when accessing the unified store. This unified store bank conflict can be observed by Razor.

(4) Color buffer location and memory layout

Throughput may decrease due to the combination of the color buffer location and memory layout. See Chapter 14 for recommended settings.

PTLA

This section provides theoretical maximum performance of the PTLA unit. The maximum transfer speed (pixel/clock) changes depending on the type of transfer operation and the bpp (bits per pixel) of transferred data. The following sections show the performance details.

For additional information on the PTLA unit, refer to "Transfers using PTLA".

Copy Operation with No Format Conversion

Performance
10.66 pixel/clock (8bpp)
6.40 pixel/clock (16bpp)
4.57 pixel/clock (24bpp)
3.55 pixel/clock (32bpp)
1.78 pixel/clock (64bpp)
0.89 pixel/clock (128bpp)

Copy Operation With Format Conversion

Performance
4.00 pixel/clock (destination = 24bpp or 16bpp or 8bpp)
3.55 pixel/clock (destination = 32bpp)

Note that transfer performance is not affected by source bpp.

Downscale Operation

Performance
2.10 pixel/clock (any format)

Note that pixel count shows the source buffer performance.

Fill Operation

Performance
4.00 pixel/clock (any format)

14 Basic Checkpoints for GPU Programming

This chapter provides basic checkpoints for better GPU performance, basically from the viewpoint of programming. Some sections include explanation about related Razor performance metrics.

Avoid Partial Rendering

If partial rendering is triggered, the resultant rendering time will increase dramatically. Please allocate enough memory as parameter buffer to avoid partial rendering whenever possible.

Parameter buffer size can be specified as one of parameters for `sceGxmInitialize()` function.

Whether or not partial rendering has been triggered can be checked by Razor Live Metrics. The amount of parameter buffer being used in run-time can be observed by Razor Live Metrics or by GPU Metrics.

Keep the Number of Scenes to a Minimum

The PlayStation®Vita GPU tile-based deferred rendering architecture performs hidden surface removal within each scene so that only visible pixels are fragment shaded. Since hidden surface removal is performed only with each scene - and not across scene boundaries - it is advised that multiple draws to a single surface (including clear polygons) should always be within a single scene whenever possible.

For example, performing a single clear within a scene and then performing draws to the same surface in a later scene is very inefficient. This is because the clear fragment shader will be executed for all pixels (which would not be necessary if the draws were grouped in the same scene, due to hidden surface removal). Additionally, the entire surface will be unnecessarily flushed to memory (on clear scene end) and loaded from memory (on draw scene begin), thus further reducing performance due to unnecessary memory access.

(1) Bad example pseudo code:

```
beginScene(); //clear
drawClearPolygon();
endScene();
beginScene(); //draw
drawGeometry ();
endScene();
```

(2) Good example pseudo code:

```
beginScene(); //clear and
draw
drawClearPolygon();
drawGeometry ();
endScene();
```

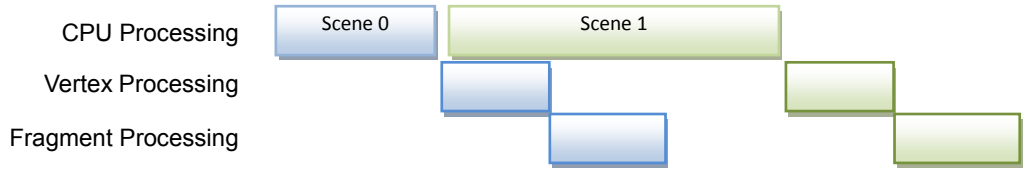
In addition to reduced performance, using more scenes than is absolutely necessary can also reduce the quality of the final rendered image. For example, due to intermediate PBE pixel format conversion to lower precision surface formats. Additionally, the quality of MSAA edge anti-aliasing may be reduced, since fragment samples are basically downsampled to pixels each time the tiles of a scene are stored to memory.

Build Scenes Efficiently in the CPU to Avoid the GPU Going into Standby

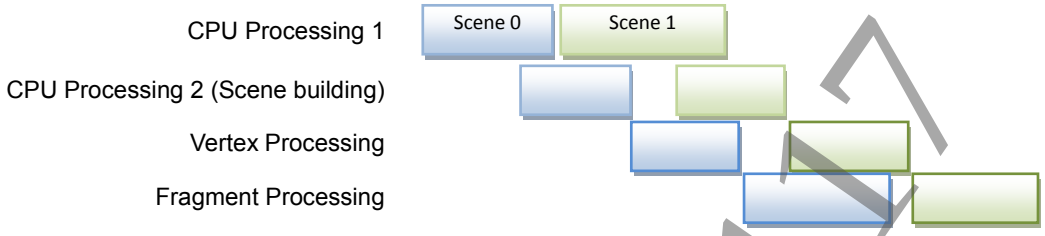
Scenes are basically transferred to the GPU when `EndScene` is called. If the CPU time between one `EndScene` and the next `EndScene` is too long, the GPU may go into standby after finishing its processing during that time, which will reduce the overall efficiency ((1) in the diagram below). In this case, to prevent the GPU going into standby during a scene, the desirable countermeasure for both the CPU and the GPU is to minimize the time between `EndScene` calls by optimizing the scene-building processing in the CPU (using parallelization, precomputation, and deferred context).

Figure 38 GPU Standby and Alternatives

(1) Example of waiting time occurring in the GPU



(2) Example of efficient pipeline processing



In example (1) in Figure 38, it is possible to reduce the overall processing time by intentionally splitting Scene 1 halfway through, but this method is not recommended because of the inefficiency generated by the increase in scene changes in the GPU.

Vertex Indices and Vertex Input Attributes

Vertex input attributes should be interleaved in memory whenever possible, as it will improve memory access efficiency, potentially resulting in better performance.

Sometimes only position attributes are required, for example, in the cases like shadow mapping. In such cases, it is recommended to store position attributes in a single array and other attributes interleaved.

Also, within each SGX core, the VDM retains last 12 index values during Draw Index List processing. If an incoming index is identical to one of the 12 previous indices, the SGX core will skip vertex shading required for that index. By sorting the order of indices based on this behavior, redundant vertex shading can be avoided. Vertex Cache Optimizer library is available for optimizing vertex data for optimum performance.

Vertex input attributes are cached on the DCU.

Vertex Index Formats: 16-bit or 32-bit

32-bit indices require more PDS processing than 16-bit indices. Usually this PDS processing doesn't become a bottleneck, however, in some cases where vertices are supposed to be processed at high throughput, this PDS processing may become a bottleneck and result in USSE throughput slowdown. It is recommended that 16-bit indices are used in most cases.

In cases where 32-bit indices are required, please note that only the low 24-bits are used.

Triangle Strip (TRIANGLE_STRIP) versus Triangle List (TRIANGLES)

Which is better in terms of PlayStation®Vita GPU performance, Triangle strip (TRIANGLE_STRIP) or Triangle list (TRIANGLES)?
There are two factors that mainly decide the PlayStation®Vita GPU processing efficiency of TRIANGLE_STRIP and TRIANGLES: the cycles required for primitive assembly and the count of vertices that are shaded.

(1) Cycles Required for Primitive Assembly

As described in Chapter 12, for Primitive assembly more cycles are required for TRIANGLES than TRIANGLE_STRIP. However, in case of TRIANGLE_STRIP, degenerated triangles must be inserted in order to connect as a single strip, which can remove much of the advantage of the higher assembly throughput of TRIANGLE_STRIP. In general, in terms of total cycles required for Primitive assembly, there is not much difference between TRIANGLE_STRIP and TRIANGLES.

(2) Count of Vertices that are Shaded (reuse ratio of vertex index CAM)

TRIANGLE_STRIP can be easily converted into TRIANGLES, just by recomposing its indices. During the conversion, the count of vertices that are shaded (which equals the count of vertices that did not hit the CAM) does not change. So the cost for vertex shading does not change in this case. However, it is not always easy to convert TRIANGLES to TRIANGLE_STRIP, as there are more constraints about vertex ordering on TRIANGLE_STRIP than on TRIANGLES, (due to adjacent triangles stripping and backface directioning, for example). Eventually, more efficient sorting can be performed in TRIANGLES, so in general the vertex CAM reuse ratio of TRIANGLES is always higher than or equal to that of TRIANGLE_STRIP. For the same reason, the spatial locality of accessed vertex data on memory of TRIANGLES is always higher than or equal to that of TRIANGLE_STRIP.

Considering (1) and (2) above, in theory it depends on particular cases which of TRIANGLE_STRIP or TRIANGLES is better in performance. However, in practice, the cycles required for vertex shading is much longer (thus much dominant) than that for Primitive assembly. So, we can say that (2) affects overall GPU performance more dominantly than (1) does.

In conclusion, we can say that in most cases TRIANGLES results in better GPU performance than TRIANGLE_STRIP, if the indices and the vertices are sorted properly - for example by using Vertex Cache Optimizer.

One of the exceptions is rendering simple geometries such as planes and curved surfaces using a very short vertex shader, in which case CAM reuse ratio does not differ much between TRIANGLE_STRIP and TRIANGLES, and because of higher Primitive assembly throughput, TRIANGLE_STRIP can result in better performance than TRIANGLES.

Culled Primitives and Vertex Processing

As culling is performed within the MTE after vertex shading, reading vertex data from memory and executing vertex shading is required even for culled primitives. So, if objects that are completely out of screen or back-faced primitives are inputted successively, vertex attributes fetch or vertex shading might become a bottleneck.

In such cases, depending on the load balancing of CPU and GPU, it may be better to perform object culling on the CPU side in advance. Similarly, the use of CPU object level-of-detail is also likely to prove beneficial. Such methods also have the benefit of reducing the possibility of partial renders.

Avoiding ISP Triangle Count HW Limit Flush

There is an upper limit to the number of primitives retained by the ISP which it determined to be visible as a result of hidden surface removal. During tile processing, if the number of retained primitives exceeds this limit, the ISP will flush all of the primitives being retained at that time to the USSE and will then resume hidden surface removal of subsequent primitives. This is called an ISP Triangle Count HW (Hardware) Limit Flush. If this occurs, fragment shading will ultimately be performed for pixels that are not visible and this reduces performance. Primitives that are determined not to be visible during hidden surface removal are not retained by the ISP.

The specific upper limit depends on the frequency with which the texture state changes and whether or not discard is used. If discard is used, or if the texture state changes with each primitive, the number of primitives that can be retained will be a minimum of 63. If discard is not used, the texture state generally

does not change that frequently. Therefore, more than 63 primitives can be retained. Typically, several hundred primitives can be retained (other detailed conditions are omitted).

Whether or not an ISP Triangle Count HW Limit Flush occurs can be determined by a Razor GPU trace.

Relationship Between Small Polygons and Vertex Processing Time

When many small polygons are processed, vertex processing time is affected not only by the pure increase of the vertex shading amount, but also by the increase of tiling process.

Usually such an increase of tiling process doesn't become a dominant slowdown factor, but for example, if there are lots of small (and thin) triangles randomly located on the rendered surface, the corresponding vertex job can be bottlenecked by the tiling process. In such cases, changing to higher sample resolution or enabling MSAA can increase the processing time of the vertex job because more tiles will intersect with the same input triangles in those cases, which increases the tiling process.

Relationship Between Small Polygons and Fragment Processing Time

When many small polygons are processed, not only is vertex processing time affected, but fragment processing time is affected as well. The main reasons why small polygons affect fragment processing time are: the ISP cost of primitive set up that is performed before hidden surface removal; the ISP HW Limit Flush; the TSP cost of fetching fragment input attributes; and a reduction in USSE operating efficiency. Taking these causes into consideration, it is recommended that you avoid unnecessary small polygons and use a suitable object LOD so that USSE operating efficiency drops as little as possible (see section “Relationship of Quadrants in a Tile and USSE“).

Fragment Input Attributes and ITR Interpolation/Output Throughput

As for fragment input attributes, there are two major bottleneck reasons: the interpolation at ITR itself and the output transfer from ITR.

First, the interpolation at ITR performance depends on the number of attributes, not on the number of components. So if the interpolation at ITR becomes a bottleneck, it is beneficial to pack multiple attributes into single attribute (ex: pack two half2 attributes into one half4 attributes).

Next, the output transfer from ITR performance depends on the size of attributes. So for example, a fp32 x 4 component attribute transfer takes twice the number of cycles compared to a fp16 x 4 component attribute. If the transfer becomes a bottleneck, the performance will be probably improved by making the size of the attributes 64-bit or less.

Also note that if lots of small polygons are rendered, the cost of fetching vertex output attributes from memory (performed by TSP) cannot be ignored and can affect the efficiency of the pipeline, as described in section “Relationship Between Small Polygons and Fragment Processing Time“.

Half and Float in Cg Program

If half and float variables are mixed in a Cg program, implicit cast instructions (for variable format conversions) are inserted. In that case, consistently using only half or only float for corresponding processing can reduce the number of instructions.

psp2shaderperf can be used to check the expected number of processing cycles and the disassembly of a Cg program.

USSE Register Bank Clashes

Please note that reducing the number of instructions will not always result in better performance. One reason for this is USSE register resource conflicts known as *bank clashes*.

Where bank clashes take place within your scenes can be viewed within Razor.

Use .xy Components for Non-Dependent Texture Fetch

A texture look-up function (ex: `tex2D()`)which has components other than .xy as texture coordinate will be always regarded as dependent texture fetch. This is because texture fetch instruction requires its coordinates to be placed within the .xy components and so component twiddling to .xy needs to be executed within the fragment shader.

To avoid such unexpected dependent texture fetches, texture coordinates for non-dependent texture fetches should always use .xy components. If you intentionally use dependent texture fetch, you don't have to use .xy components.

The performance differences between non-dependent and dependent texture fetches will be described in the future.

Texture Access and the TCU

Textures should be mip-mapped for TCU (Texture Cache Unit) efficiency whenever possible.

Actual performance of texture read is affected mainly by the following factors:

- Texture coordinates computation and transfer at TITR (UITR in case of dependent texture)
- Texture data fetch from memory at TCU (Caching efficiency, Memory access latency, bandwidth)
- Texture data input transfer and filtering operation at TF
- Count and frequency of issued texture instructions

For example, if multiple texture fetches each depend on the previous texture fetch result, the processing latency of TCU and TF cannot be hidden, and USSE processing can be very inefficient.

If fragment USSE processing is bound by non-dependent texture fetches, each part of the TITR - TAG - TCU - TF flow should be checked to determine the reason for the bottleneck.

If fragment USSE processing is bound by dependent texture fetches, each part of the UITR - USSE (texture coordinates calculation) - TAG - TCU - TF flow should be checked to determine the reason for the bottleneck.

Texture Memory Layout: Linear or Tiled or Swizzled

Basically Tiled and Swizzled layout result in better memory access efficiency. However, depending on the access pattern, Linear may result in better performance.

From the viewpoint of consumed memory bandwidth, a compressed format such as UBC (DXT) or PVRT is recommended. Exactly which of Tiled/Linear/Swizzle is most efficient in performance depends on the rendering scene and is still being investigated. The results of these investigations will be described in the future.

Surface Memory Layout: Linear or Tiled or Swizzled

When a surface is located on video memory, Linear layout is faster than Tiled and Swizzled layout in terms of PBE write performance. For a surface that is not read as a texture later, Linear layout is recommended. Video can only be output from a Linear layout surface.

For a surface that is read as a texture later, Tiled and Swizzled may have better performance due to better texture cache access efficiency. This is a trade-off of writing performance and reading performance. Note that in some cases, a Linear layout results in the best performance for both writing and reading.

Exactly which of Tiled/Linear/Swizzle is most efficient in performance depends on the rendering scene and is still being investigated. The investigation result will be added in the future.

Disable the Fragment Program when it is no Longer Needed

If the color value calculation in a fragment program is no longer needed, such as when creating a shadow map, it is recommend that the application disable the fragment program using one of the following methods. If the fragment program is not disabled, the USSE will unnecessarily execute it, which may lead to a drop in performance.

(1) Set `SceGxmBlendInfo.ColorMask` to `NONE` (which will automatically disable the fragment program).

(2) Disable the program using `sceGxmSetFrontFragmentProgramEnable()` or `sceGxmSetBackFragmentProgramEnable()`.

Caution: The fragment program should not be disabled if it is performing discard or depth-replace operations. If the fragment program is disabled, the processing associated with the discard or depth-replace operations will be disabled, and the depth buffer will not be updated.

Shader Pass Type Switching

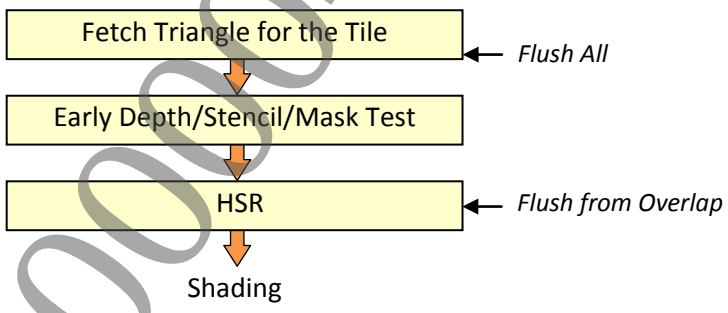
Frequently switching among Shader pass types (Opaque/Translucent/Discard/Depth-Replace) may result in inefficient processing of each tile. Because switching occurs within all tiles that include corresponding primitives, and depending on which pass types are changed, because one switch can incur a pipeline flush within ISP, such frequent switching in a single scene may cause pipeline bubbles and result in lower USSE efficiency. See "ISP Flushes Based on Shader Pass Type" for more information.

Rendering each of opaque geometries, masked geometries, and blended geometries at a time whenever possible can keep the number of switching minimum. The number of ISP pass switches and where they take place within your scenes can be viewed within Razor.

ISP Flushes Based on Shader Pass Type

Figure 39 shows the basic pipeline stages for the ISP processing of hidden surface removal.

Figure 39 ISP Hidden Surface Removal Pipeline and Flushes



ISP fetches a triangle, performs early depth/stencil/mask test per pixel, performs hidden surface removal at the HSR stage, and sends pixels to later units (PDS/USSE) for shading. The HSR stage can hold single pass-type information for each pixel in the tile to correctly perform hidden surface removal.

Under certain conditions, ISP flushes all pixels being tracked so far at HSR on to later stages, empties all pass-type information being tracked, and continues processing for the rest of the tile. There are two such ISP flushes: "Flush All" and "Flush from Overlap", as shown in Figure 39.

Flush All

At the "Fetch Triangle for the Tile" stage, when the pass type for the triangle changes from Discard or Depth-Replace to Opaque or Translucent, ISP flushes all non-empty pixels being tracked for the tile thus far and empties all existing per-pixel pass type information. To ensure that the new opaque or translucent primitive is depth/stencil/mask tested against the correct values, further processing of the fetched

triangle will stall (before "Early Depth/Stencil/Mask Test" is performed) until the Discard or Depth-Replace results of all the flushed pixels have been computed by USSE.

Note: The pass-type transition described here is per-primitive, not per-pixel.

Flush from Overlap

At the HSR stage, if an incoming pixel overlaps with an existing (non-empty) pixel, and if the pass type transition between the two overlapped pixels is not from Opaque or Translucent to Opaque, ISP flushes all non-empty pixels being tracked in the tile at HSR so far and empties all existing per-pixel pass type information. In this case, HSR can immediately continue processing the incoming pixels without waiting for the flushed pixels to be processed. For that reason, Flush from Overlap is usually lighter than Flush All.

Note: The pass-type transition described here is per-pixel, not per-primitive.

Though HSR does not stall immediately from this flush, the flush can result in USSE inefficiency due to imbalance over four USSE pipes. This can occur, for example, if flushes occur successively and if the flushed pixels cover only limited area of the tile.

Table 112 Pass Type Transition at HSR and Flush from Overlap

Existing Pixel Pass Type	Incoming Pixel Pass Type	Result
Opaque or Translucent	Opaque	An unshaded, existing pixel at HSR is overwritten by an incoming pixel without flush. (This is the hidden surface removal.)
All other cases		Flush from Overlap: flush all existing pixels for shading.

In addition, note that not only does the Early Depth/Stencil/Mask Test occur for pixels with Opaque or Translucent pass type, but also for pixels with Discard pass type. For pixels with Depth-Replace pass type, the Early Depth/Stencil/Mask Test will not occur; instead a depth/stencil/mask test is only performed after USSE has computed the final depth values.

Semi-transparent Pixel and Fully Transparent Pixel (ex: sprite with alpha)

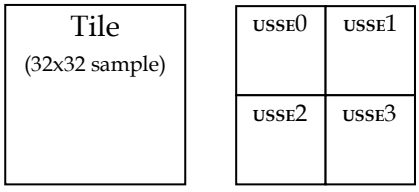
The GPU does not differentiate fully transparent pixels from semi-transparent pixels and so processes them in the same (blending) manner.

For example, if most pixels in a triangle are fully transparent (alpha=0) for sprite rendering, the pixel processing time for blending those pixels are consumed in vain. Using finer mesh that fits with the shape of semi-transparent or opaque pixels to reduce fully transparent pixels would be more efficient.

Relationship of Quadrants in a Tile and USSE

Each of four 16x16 fragment samples in a tile is called a *quadrant*. Pixels included in each tile quadrant will be separately distributed amongst each core's 4 USSE pipes as follows. For example, pixels included in top-left quadrant are always processed by USSE0:

Figure 40 4 Quadrants in a Tile and USSE



Rasterized pixels are sent to USSE and are shaded there. Different fragment programs can run in parallel. So the amount of fragment shading for each quadrant may differ greatly. When this occurs, some USSE pipes will be running and while other USSE pipes will be idle.

Rendering lots of small polygons can result in such uneven distribution of 4 USSE's due to this hardware binding and may result in USSE slowdown. Especially if blending is enabled, this tendency becomes higher as synchronization is required for overlapped pixels in a screen.

Using object LOD and avoiding too many small polygons can result in better USSE efficiency.

Resource Location: Main memory or Video memory

Simply considering that the available memory bandwidth is much wider in video memory than in main memory, it is recommended that you normally locate resources such as textures and surfaces in video memory.

However, if bandwidth of video memory becomes a bottleneck, moving some resources to main memory may result in better load balancing between main memory and video memory; and so may improve overall performance.

Avoid Cache-enabled Main Memory Access from the GPU

Access to cache-enabled main memory from the GPU is less efficient than access to video memory or cache-disabled main memory because its bandwidth is particularly small and its latency is extremely long.

For example, when each type of GPU context that is passed to `sceGxmCreateContext()` is placed in cache-enabled main memory, access from the GPU to the cache-enabled main memory will occur frequently during the scene processing, resulting in reduced efficiency. It is recommended that the context and other data used by the GPU for rendering are placed in the video memory as much as possible, and whenever the main memory is to be used, that they are placed in the cache-disabled memory.

Avoid Frequent Video Memory Access from the CPU

Video memory access from the CPU leads to processing with extremely long latency. Therefore, the efficiency of CPU processing can be improved by putting data with values that are frequently rewritten by the CPU into the main memory instead of the video memory. However, because GPU efficiency is generally improved by putting data into video memory, rather than main memory, it is recommended to consider whether to put data in main memory or in video memory based on the frequency of rewriting by the CPU and the frequency of reading by the GPU.

For example, if the coordinate values of particles or the texture structure are going to be frequently rewritten by the CPU, it is recommended placing this data in the main memory.

Observing the Load Balancing Between Four GPU Cores

The data distribution to each of the four GPU cores is basically performed in units of primitives for vertex processing, and in units of tiles for fragment processing. The distribution is performed automatically by hardware to achieve the best load balancing, and in most cases the distribution is properly performed and well-balanced.

Rarely, it is observed that only a few cores are working and the other cores are not working much around the end of a vertex job or a fragment job.

For example, if there are a few vertex shaders or fragment tiles which take much longer time than others and if only one core receives all such vertices or tiles that take longer, processing time unbalance between cores can occur because other cores finish their processing earlier.

Another example is, when fragment shaders used in all tiles a core received are much shorter (for example a clear shader) than the shaders used in the other cores, the core can finish its processing much earlier than the other cores.

By using Razor GPU Trace, you can accurately profile when and how long each of the four GPU cores has processed during a vertex job or during a fragment job, and you may be able to find specific shaders or tiles that have taken a longer or shorter time, resulting in an unbalance. However, as the distribution is controlled by hardware, the application is not able to control which vertices, or tiles, to send to which core. Depending on the causes of the unbalance, it may be possible to improve the balance by optimizing or modifying the corresponding shaders. Of course, as the distribution is decided dynamically, there is not always a concrete solution for such a balancing problem.

Transfers using PTLA

Transfer operations are performed mainly by the dedicated PTLA hardware unit inside the GPU, and are performed asynchronously to the CPU, the GPU vertex processing, and the GPU fragment processing by default. However, doing transfer processing using PTLA may affect (and be affected by) the GPU performance of existing vertex processing, or fragment processing, if they are performed (fully or partially) in parallel over time, because some GPU units and interfaces are shared across vertex processing, fragment processing and transfer processing.

If main memory is specified as both PTLA transfer source *and* destination, the PTLA transfer speed can be slower than the case for video memory, as the main memory bandwidth will be the bottleneck. If cached main memory is specified as transfer source or destination (or both), the PTLA transfer speed can be much slower.

Also, it is generally more efficient to keep the number of submitted transfer operations to a minimum, because there is a little GPU firmware overhead between two transfer operations.

Appendix A: Data Formats Summary

The following table lists the supported swizzle modes and query formats for all texture base formats.

Texture Base Format	Swizzle	float4	float2	float	half4	half2	half	unsigned int4	unsigned int2	unsigned int	signed int4	signed int2	signed int	unsigned short4	unsigned short2	unsigned short	signed short4	signed short2	signed short	unsigned char4	unsigned char2	unsigned char	signed char4	signed char2	signed char
U8	C1	•1		•1,2	•1		•1,2													•		•2			
S8	C1	•		•2	•		•2																•		•2
U16	C1	•		•2										•		•2									
S16	C1	•		•2													•		•2						
F16	C1	•		•2	•		•2																		
U32	C1							•		•2															
S32	C1										•		•2												
F32	C1	•		•2																					
F32M	C1	•		•2																					
U8U8	C2	•3	•3,4		•3	•3,4														•	•4				
S8S8	C2	•	•4		•	•4																	•	•4	
U16U16	C2	•	•4											•	•4										
S16S16	C2	•	•4														•	•4							
F16F16	C2	•	•4		•	•4																			
U32U32	C2							•	•4																
F32F32	C2	•	•4																						
U8U8U8	C3	•5			•5															•					
U5U6U5	C3	•5			•5															•					
S5S5U6	C3	•5			•5																				
S8S8S8	C3	•			•																		•		
F11F11F10	C3	•			•																				
SE5M9M9M9	C3	•			•																				
X8S8S8U8	C3	•			•																				
U4U4U4U4	C4	•5			•5															•					
U8U8U8U8	C4	•5			•5															•					
S8S8S8S8	C4	•			•																		•		
U8U3U3U2	SZ1	•5			•5															•					
U1U5U5U5	C4	•5			•5															•					
U16U16U16U16	C4	•												•											
S16S16S16S16	C4	•															•								
F16F16F16F16	C4	•			•																				
U2F10F10F10	C4	•			•																				
U2U10U10U10	C4	•			•																				
P4	C4	•5			•5															•					
P8	C4	•5			•5															•					
PVRT2BPP	SZ2	•5			•5															•					
PVRT4BPP	SZ2	•5			•5															•					
PVRTII2BPP	SZ2	•5			•5															•					
PVRTII4BPP	SZ2	•5			•5															•					
UBC1	SZ3	•5			•5															•					
UBC2	SZ3	•5			•5															•					
UBC3	SZ3	•5			•5															•					
UBC4	C1	•1		•1,2	•1		•1,2													•		•2			
SBC4	C1	•		•2	•		•2																•		•2
UBC5	C2	•3	•3,4		•3	•3,4														•	•4				
SBC5	C2	•	•4		•	•4																	•	•4	
X8U24	SD			•						•															
YUV422	Y1	•			•															•					
YUV420P2	Y2	•			•															•					
YUV420P3	Y2	•			•															•					

Notes
1. When gamma correction is enabled, the R color component from memory will be gamma corrected.
2. For R swizzle mode only
3. Gamma correction can be enabled on just the R color component from memory, or both the R and G color components from memory.
4. For GR swizzle mode only
5. When gamma correction is enabled, the R, G and B color components from memory will be gamma corrected.
C1. Supports swizzles 000R 111R RRRR 0RRR 1RRR R000 R111 R
C2. Supports swizzles 00GR GRRR RGGG GRGR 00RG GR
C3. Supports swizzles BGR RGB
C4. Supports swizzles ABGR ARGB RGBA BGRA 1BGR 1RGB RGB1 BGR1
SZ1. Supports ARGB layout only
SZ2. Supports swizzles ABGR 1BGR
SZ3. Supports swizzle ABGR only
SD. Supports swizzles SD DS
Y1. Supports swizzles YUYV_CSC0 YUYV_CSC1 YVYU_CSC0 YVYU_CSC1 UYVY_CSC0 UYVY_CSC1 VYUY_CSC0 VYUY_CSC1
Y2. Supports swizzles YUV_CSC0 YUV_CSC1 YVU_CSC0 YVU_CSC1

The following table lists the supported swizzle modes, output register formats, dither and downscale features for each color base format.

Color Surface Base Format	Swizzle	float2	float	half4	half2	unsigned short2	signed short2	unsigned char4	signed char4	Dither	Downscale
U8	SZ5				•[1]			•			•
S8	SZ2								•		•
U16	C1					•					•
S16	C1						•				•
F16	C1				•						•
F32	SZ3		•								
U8U8	C2				•[2]			•			•
S8S8	C2								•		•
U16U16	SZ4					•					•
S16S16	SZ4						•				•
F16F16	SZ4				•						•
F32F32	SZ4	•									
U5U6U5	C3			•[3]				•		•	•
S5S5U6	C3			•							•
U8U8U8	C3			•[3]				•		•	•
SE5M9M9M9	C3			•							•
F11F11F10	C3			•							•
U4U4U4U4	C4			•[3]				•		•	•
U8U8U8U8	C4			•[3]				•		•	•
S8S8S8S8	C4								•		•
U8U3U3U2	SZ1			•[3]				•		•	•
U1U5U5U5	C4			•[3]				•		•	•
U2U10U10U10	C4			•						•	•
F16F16F16F16	C4			•							•
U2F10F10F10	C4			•							•

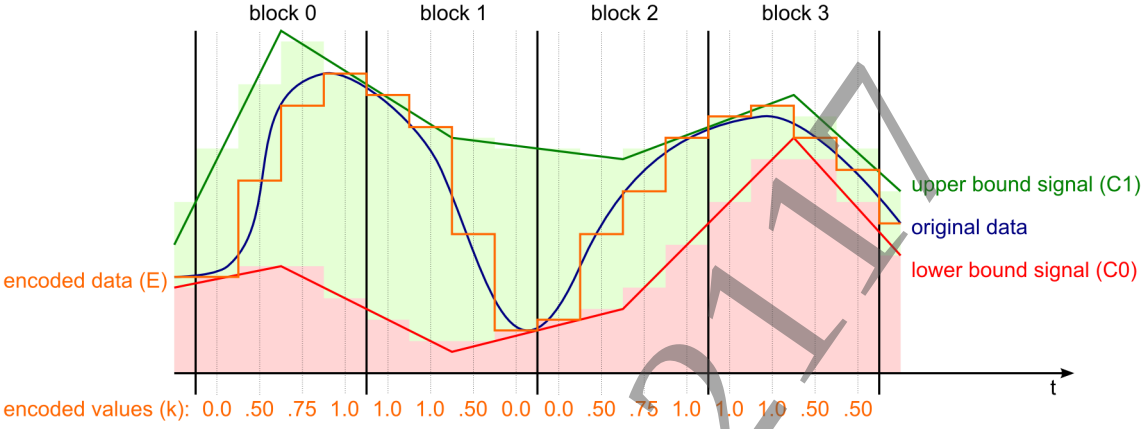
Notes
1. This output format is only available for use when gamma correction is enabled and the R representation is used. The R color component is gamma correct before being written to memory.
2. This output format is only available for use when gamma correction is enabled and either the GR or RG representation is used. Either just the R color component or both the R and G color components can be selected to be gamma correct before being written to memory.
3. When gamma correction is enabled, the R, G and B color components are gamma correct before being written to memory.
C1. Supports swizzles R G
C2. Supports swizzles GR RG RA AR
C3. Supports swizzles BGR RGB
C4. Supports swizzles ABGR ARGB RGBA BGRA
SZ1. Supports ARGB layout only.
SZ2. Supports swizzles R A
SZ3. Supports swizzle R only
SZ4. Supports swizzles GR RG

SZ5. Supports swizzles R G A
SD. Supports swizzles SD DS

Appendix B: Compressed Texture Formats

The GPU supports a series of block-based, lossy compressed texture formats, using the PVRT compression algorithm. This algorithm operates by encoding a high-frequency signal as linear combination of two lower-frequency bounding signals (*C0* and *C1*), weighted by a high-frequency, quantized modulating signal (*k*). Figure 41 illustrates how it works for a 1-Dimensional example, using a period size of *4* samples for the bounding signals and *K={0.0, .25, .50, .75, 1.0}* as the quantization set for *k*.

Figure 41 1-Dimensional example of PVRT compression



$$E(t) = (1 - k(t))C0(t) + k(t)C1(t)$$

The bounding signals are encoded with high precision, while the higher frequency modulating signal *k(t)* is encoded using the index *i(t)* of the corresponding quantized value. This balance is where the economy of space introduced by this algorithm resides.

The GPU currently supports four PVRT variants, which mainly differ on the quantization sets they use, whose size determines the compression rate they achieve. Also, some of them introduce additional encoding modes, which allow them to address specific compression artifacts.

Table 113 Compressed Texture Formats

Format name	Compression rate	Bounding signal modes
PVRT-I 4bpp	8:1	Interpolated
PVRT-I 2bpp	16:1	Interpolated
PVRT-II 4bpp	8:1	Interpolated, constant, palettized
PVRT-II 2bpp	16:1	Interpolated, constant, palettized

All PVRT variants can handle RGB and RGBA textures, but unlike other compression algorithms they encode alpha information in the same way they encode other color components, rather than including it as a separate channel.

Benefits

The benefits of texture compression are felt in the following areas:

- **Memory footprint:** the smaller size of compressed texture assets allows the use of a larger variety of textures to render a scene, or to improve visual quality by increasing their resolution.
- **Performance:** asymptotically, the memory traffic needed to render a textured asset decreases by the same ratio as the texture size. Furthermore, block-based compression formats allow for a better use of caching mechanisms to leverage the cost of memory accesses across locally-coherent texture fetches.

- **Power consumption:** since memory traffic is one of the main power drains in embedded systems, the reduced requirements of compressed textures helps increase battery life.

Drawbacks

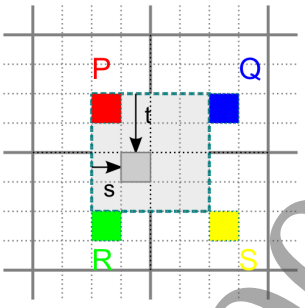
The benefits enumerated above come at the expense of trade-offs in:

- **Color resolution:** PVRT compression encodes the bounding signal values as R5G5B5, or R4G4B4A3 in the case of textures with alpha. Although interpolation is done at full precision, this can reduce color precision throughout.
- **Dimension constraints:** the dimensions of PVRT-compressed textures must be a multiple of the block size. Also they must be powers of 2 to allow for the swizzling inherent to the format.
- **Artifacts:** depending on the compression algorithms used and the nature of the input data, PVRT-compressed textures may present perceptible distortions.
- **Compression costs:** compressing texture data with minimal errors may require iterative algorithms that are quite expensive both in terms of memory and time, preventing a quick turnaround in asset iteration. Most compression tools expose options to leverage quality against performance. In any case, real-time compression is hardly possible and compressed textures must be treated as read-only data.

PVRT-I 4bpp

PVRT-I extends the encoding paradigm illustrated by Figure 41 to the 2-dimensional domain of the texture, which its 4bpp variant divides into blocks of 4x4 texels. In a similar way to Figure 41, bounding signals are constructed by bilinearly interpolating upper and lower values across adjacent blocks. Figure 42 names such blocks *P, Q, R, S* and provides the expression of the encoded data for the region comprised between their respective centers (referred to as *intrablock*).

Figure 42 PVRT-I encoding of a texture



$$C0(s,t) = \left(1-\frac{t}{4}\right)\left(\left(1-\frac{s}{4}\right)C0_P + \frac{s}{4}C0_Q\right) + \frac{t}{4}\left(\left(1-\frac{s}{4}\right)C0_S + \frac{s}{4}C0_R\right)$$

$$C1(s,t) = \left(1-\frac{t}{4}\right)\left(\left(1-\frac{s}{4}\right)C1_P + \frac{s}{4}C1_Q\right) + \frac{t}{4}\left(\left(1-\frac{s}{4}\right)C1_S + \frac{s}{4}C1_R\right)$$

$$E(s,t) = (1-k(s,t))C0(s,t) + k(s,t)C1(s,t)$$

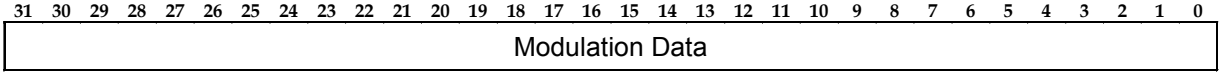
PVRT-I allows two different encoding modes, the first of them (mode 0) uses the quantization set *K={0, 3/8, 5/8, 1}* and follows the expression for *E* given in Figure 41 for each of the texture components.

The second mode (mode 1) evaluates a different expression for the color and alpha components depending on the modulation index for each texel:

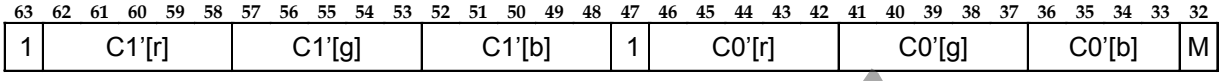
$$i(s,t) = \begin{cases} 0 \Rightarrow & E(s,t)[r,g,b] = C0(s,t)[r,g,b] & E(s,t)[a] = C0(s,t)[a] \\ 1 \Rightarrow & E(s,t)[r,g,b] = 0.5(C0(s,t)[r,g,b] + C1(s,t)[r,g,b]) & E(s,t)[a] = 0.5(C0(s,t)[a] + C1(s,t)[a]) \\ 2 \Rightarrow & E(s,t)[r,g,b] = 0.5(C0(s,t)[r,g,b] + C1(s,t)[r,g,b]) & E(s,t)[a] = 0 \\ 3 \Rightarrow & E(s,t)[r,g,b] = C1(s,t)[r,g,b] & E(s,t)[a] = C1(s,t)[a] \end{cases}$$

This choice of expressions provides higher accuracy in areas where alpha values drop sharply towards zero (e.g. decal borders).

In both modes, the values of *i(s,t)* can be stored using 2 bits, thus requiring 32bits for the modulation data of a 4x4 block. This is stored along the LSB of the block data, following row-major order:



The remaining 32 MSB contain the values of *C0* and *C1* for the block, as well as a flag indicating the encoding mode used. In the case of entirely opaque blocks, *C0* and *C1* are stored in R5G5B4 and R5G5B5 formats, respectively:

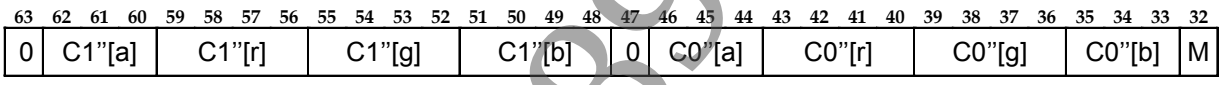


When decompressing, the bounding signal components are unpacked to 8bit using the expressions below:

$$C0[r] = (C0'[r] << 3) \mid (C0'[r] >> 2) \qquad C1[r] = (C1'[r] << 3) \mid (C1'[r] >> 2)$$
$$C0[g] = (C0'[g] << 3) \mid (C0'[g] >> 2) \qquad C1[g] = (C1'[g] << 3) \mid (C1'[g] >> 2)$$
$$C0[b] = (C0'[b] << 4) \mid C0'[b] \qquad C1[b] = (C1'[b] << 3) \mid (C1'[b] >> 2)$$
$$C0[a] = (C0'[a] << 4) \mid C0'[a] \qquad C1[a] = (C1'[a] << 4) \mid C1'[a]$$

taking 0xF as the value for the missing *C0[a]* and *C1[a]* components.

In the case of blocks with transparency information, *C0* and *C1* are stored instead in R4G4B3A3 and R4G4B4A3 formats, respectively:

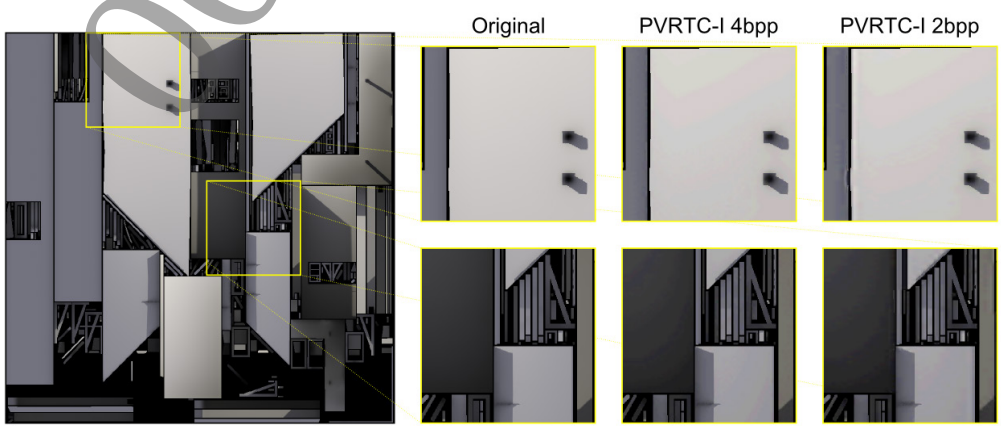


The unpacking to 8bit components, *C1''* and *C0''* are first converted to their R5G5B5A4 counterparts, and then expanded using the same expression which maps *C1'* to *C1*:

$$C0'[r] = (C0''[r] << 1) \mid (C0''[r] >> 3) \qquad C1'[r] = (C1''[r] << 1) \mid (C1''[r] >> 3)$$
$$C0'[g] = (C0''[g] << 1) \mid (C0''[g] >> 3) \qquad C1'[g] = (C1''[g] << 1) \mid (C1''[g] >> 3)$$
$$C0'[b] = (C0''[b] << 2) \mid (C0''[b] >> 1) \qquad C1'[b] = (C1''[b] << 1) \mid (C1''[b] >> 3)$$
$$C0'[a] = C0''[a] << 1 \qquad C1'[a] = C1''[a] << 1$$

It is worth noting that when using these unpacking expressions, the range of the alpha component for RGBA blocks is limited to *[0,0xEE]*, rather than *[0,0xFF]*. Thus it is not possible to have fully opaque texels in such blocks.

Figure 43 Color shifting and banding artifacts in PVRTC texture compression



Also, the asymmetry in component precision for *C0* and *C1* results introduces color shifting in the compressed texture, since the accuracy achieved for each component is different. Together with the inherent banding artifacts, this makes PVRT compression rather unsuitable for textures with large grayscale regions, such as lightmaps, as demonstrated in Figure 43.

Whereas the block size is simply 8 bytes, we require a minimum of 4 blocks for the evaluation of a given texel, thus the memory read required for a texture fetch is 32 bytes. Coherent texture fetches will reuse blocks heavily, allowing for the caching mechanisms of the GPU to leverage the cost of the initial fetch.

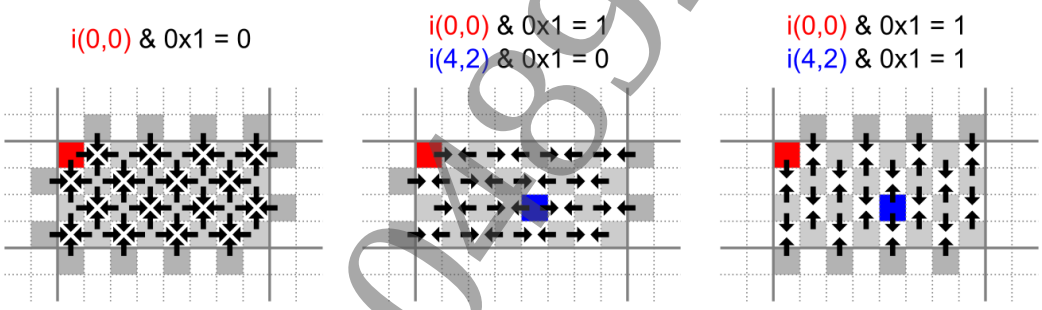
PVRT-I 2bpp

PVRT-I has a second variant that encodes 8x4 texel portions of the texture using the same 64bit block, hence the 16:1 compression rate. This is achieved through one of the two following encoding methods:

- **Mode 0:** store 1 bit per texel, selecting either *C0(s,t)* or *C1(s,t)* as the value for the encoded texture *E(s,t)*.
- **Mode 1:** store 2 bits per texel for each of the even positions in a checkerboard pattern, and compute the modulation values for their odd counterparts by interpolation. Then evaluate the encoded texture using the quantization set *K={0, 3/8, 5/8, 1}*, as in PVRT-I 4bpp mode 0.

Mode 1 considers three possible interpolation patterns: bilinear, horizontal and vertical, depending on the least significant bit of the modulation indices for some of the block's even texels. The details are given in Figure 44.

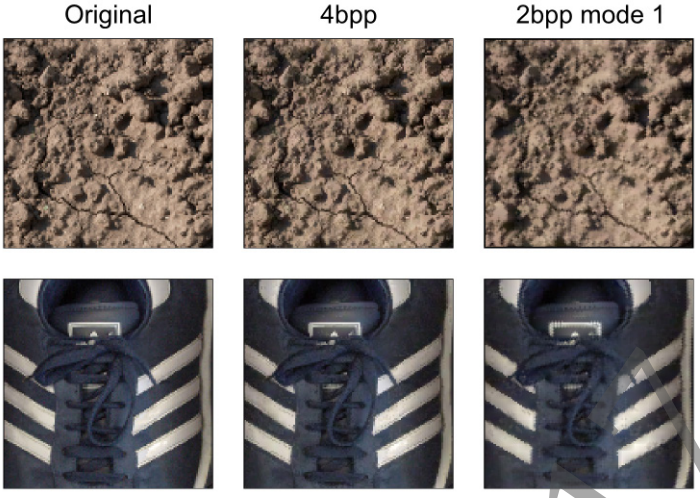
Figure 44 Blending patterns for mode 1 of PVRT-I 2bpp



On decoding, the bits used to select the blending pattern are set to the other bit of the corresponding modulation index. This guarantees we can always encode extreme values accurately.

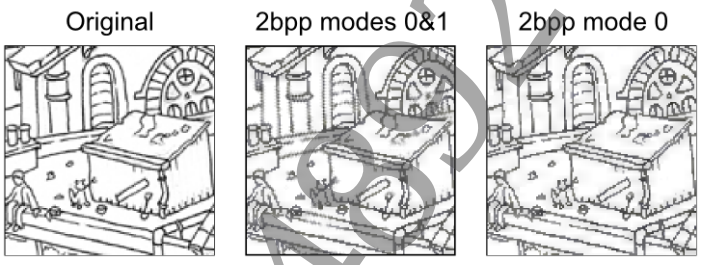
The quality tradeoffs taken by PVRT-I 2bpp are larger than its 4bpp counterpart, yet in many cases the size reduction easily justifies its use. Mode 1 introduces considerable errors on a texel per texel basis, but these errors may not be especially noticeable unless the source texture has a clearly defined structure, such as hard boundaries or outlines. Figure 45 shows a couple of examples of favorable cases.

Figure 45 Favorable cases for PVRT-I 2bpp mode 1



If the source texture presents harsh features, such as line drawing or text, mode 0 performs significantly better. One of such cases can be found in Figure 46.

Figure 46 Pathological case for PVRT-2bpp mode 1



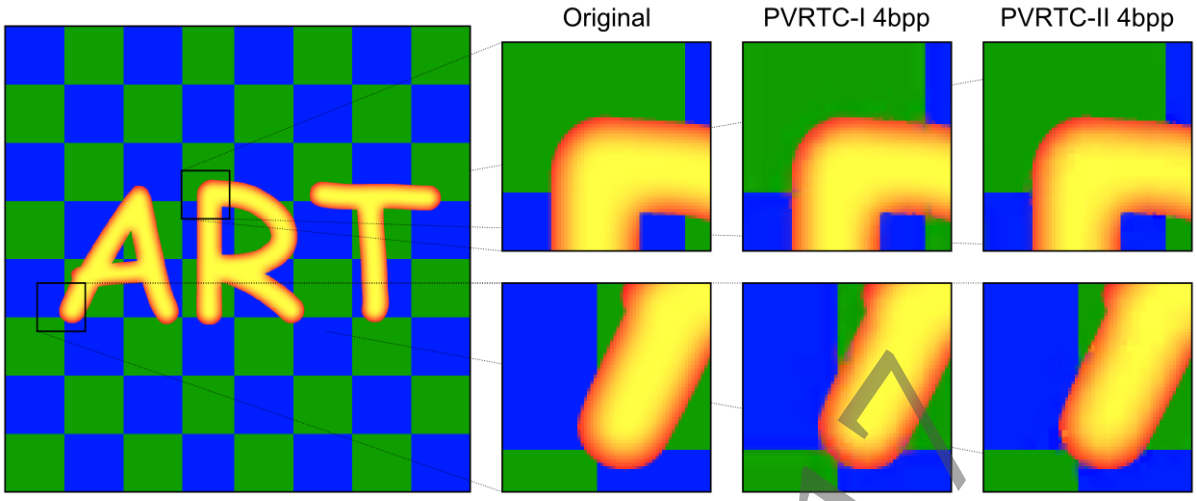
Also, the results are generally sharper than using a downsampled version of the texture compressed in 4bpp mode.

As with the PVRT-I 4bpp variant, the memory accessed for a single texture fetch is 32 bytes, yet there is even further opportunity for block reuse across coherent fetches since the block size is larger.

PVRT-II 4bpp

It seems intuitive from Figure 41 that the error introduced by PVRT compression can be related to the separation between the upper and lower bounding signals. In blocks with steep changes and high frequency features that separation will be necessarily large, and because of the way *C0* and *C1* are constructed by linear interpolation, it will carry on to adjacent blocks, thus spreading the error. This produces some characteristic *smearing artifacts* around image discontinuities. Some of these effects are illustrated in Figure 47.

Figure 47 Examples of the smearing artifacts tackled by PVRT-II



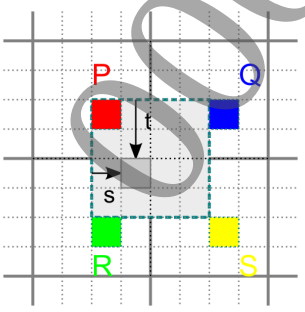
These artifacts can also be noticed across the boundaries of non-periodic textures or along the seams of texture atlases. The second revision of the PVRT compression algorithm, PVRT-II, aims to tackle them and reduce the overall error by introducing new encoding modes.

Also, PVRT-II addresses some of the problems that its predecessor showed when compressing alpha information.

PVRT-II 4bpp encoding modes

- **Mode 0:** equivalent to PVRT-I 4bpp mode 0.
- **Mode 1:** equivalent to PVRT-I 4bpp mode 1, with the exception that when $i(s,t) = 2$ the texel color is set to transparent black (pre-multiplied alpha).
- **Mode 2:** (non-interpolated intrablocks) this mode assumes that the bounding signals *C0* and *C1* remain constant across each intrablock, set to the value corresponding to its NW corner. The expression of the encoded texture takes the simple form given by Figure 48.
- **Mode 3:** (palettised intrablocks) this mode defines a local palette of 4 4x4 texels for each intrablock, using combinations of the *C0* and *C1* values in the neighboring blocks (See Figure 49). The encoded texture at a given position [s,t] in an intrablock is given by the corresponding texel in the palette entry indexed by $i(s,t)$.

Figure 48 Encoding of non interpolated intrablocks in PVRT-II 4bpp

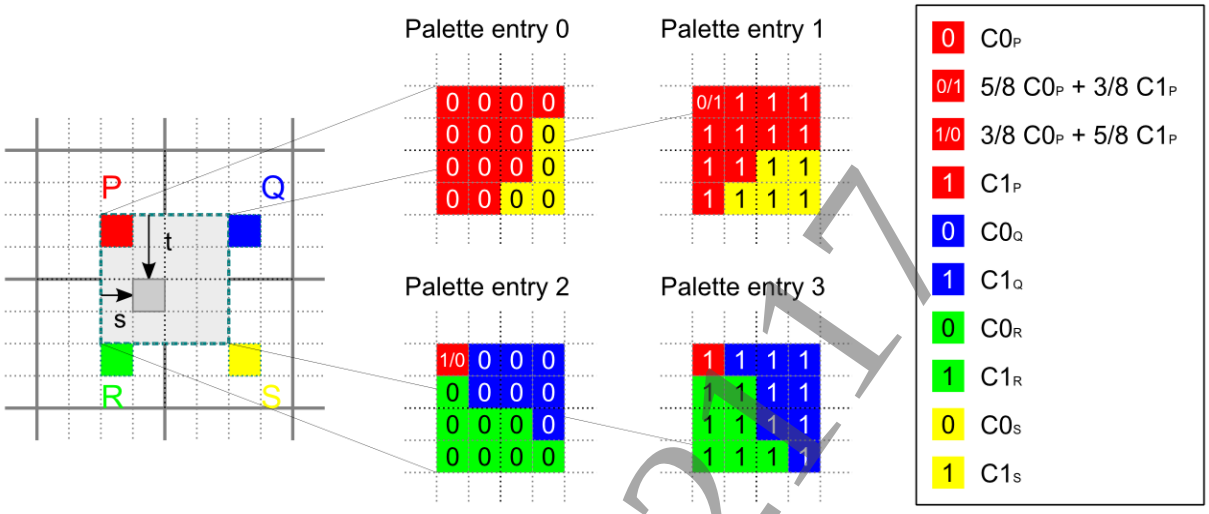


$$E(s,t) = (1 - k(s,t))C0_P + k(s,t)C1_P$$

Mode 2 mode favors the encoding of textures which aren't necessarily smooth across intrablock boundaries, as the bounding signal values can be chosen for each of them independently. This is very similar to the compression performed by other algorithms such as DXT, although the alpha channel is encoded along color information.

In fact, a DXT1-compressed RGB texture could be almost directly converted to its 4bpp PVRT-II counterpart if the later was to use non-interpolated intrablocks throughout. There would be however a loss in quality, because the encoding used for *C0* and *C1* values in 4bpp PVRT-II blocks has less precision than the R5G6B5 variant used by DXT.

Figure 49 Intrablock palette in PVRT-II 4bpp mode 3

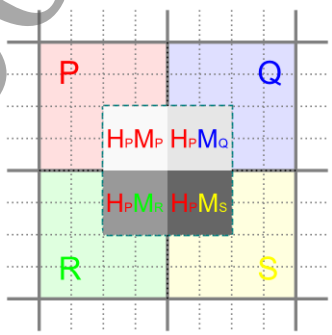


Just like mode 2, mode 3 is not interpolated, but it still relates the bounding signal values of adjacent blocks to the compression error in the intrablock in-between. This prevents the compressor from choosing such bounding signal values independently to better handle sharp changes spanning several intrablocks. Mode 2 is better suited for such scenario.

Determining the encoding mode for a given texel

In addition to *M*, PVRT-II uses a second bit in the block data (bit 47, namely *H*), to select the encoding mode. However, this is not done per block, but per intrablock quarter, following the scheme illustrated in Figure 50.

Figure 50 Encoding modes in each quarter of a PVRT-II 4bpp intrablock



Changes to alpha encoding

PVRT-II addresses the upper limit of *0xEE* for alpha values encoded by PVRT-I, affecting the compression of transparent textures with fully opaque regions. This problem originated from the expansion of *C0* and *C1* from R4G4B3A3 and R4G4B4A3 to R8G8B8A8. In the intermediate expansion to R5G5B4A4 and R5B5G5A4, PVRT-II uses the following expressions for alpha components:

$$C0'[a] = C0''[a] << 1$$
$$C1'[a] = (C1''[a] << 1) | 1$$

While this is not a symmetric solution, it still allows encoding the full range *[0,0xFF]* of alpha values, as long as *C1* contains the upper bound for the alpha channel. It is up to the compressor to guarantee so. A more flexible solution (e.g. using the same expression as for *C0'[b]*) is not possible due to hardware constraints.

PVRT-II 2bpp

The 2bpp variant of PVRT-II adds non-interpolated variants (modes 2 and 3) to the existing encoding modes in PVRT-I 2bpp (modes 0 and 1, respectively). Bits *H* and *M* are used in identical fashion to PVRT-II 4bpp in order to determine a given texel's mode. The results of using non-interpolated modes are equivalent.