libgxm Overview

000004892117

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

© 2015 Sony Computer Entertainment Inc.
All Rights Reserved.
SCE Confidential

Table of Contents

000004892117

Document serial number: 000004892117

©SCEI

000004892117

Document serial number: 000004892117

1 Overview

About This Document

This document contains information about the libgxm graphics library API. It also describes how the library calls interact with the underlying GPU. The GPU in PlayStation®Vita is a multi-core derivative of the SGX architecture.

Reference Materials

Please refer to the following materials for a detailed reference:

- *GPU User's Guide* – provides detailed descriptions of GPU data formats for primitives, textures, and surfaces.
- *libgxm Reference* – provides detailed reference descriptions of each of the libgxm API functions and data structures.
- *Shader Compiler User's Guide* – contains information about the Compiler for Cg 2.2 and the PlayStation®Vita profiles.

Typographic Conventions

The following typographic conventions are used throughout this manual:

Convention	Meaning
`courier`	Indicates data types, structure/function names, literal programming code or text that a user must type.
italics	Indicates names of arguments and structure members (in structure/function definitions only). Also indicates emphasis or a variable condition. Except when hyperlinked, book references are in italics. When a term is first defined, it is often in italics.
bold	Used to indicate a note of caution. For example, "Note: You must type the book title in the Title text box of the Summary tab."
blue	Indicates a hyperlink (color printers or online only).

Document serial number: 000004892117

000004892117

2 Using the libgxm Library

Library Characteristics

libgxm is the low-level graphics library for PlayStation®Vita, providing the following three areas of functionality:

- Rendering API.
- Shader API.
- Shader Patcher API.

Rendering

The rendering API layer deals with efficiently creating hierarchies of GPU data structures for rendering geometry. It provides a rendering *context* with persistent state, from which *scenes* of draw calls are submitted to the GPU firmware layer for rendering.

Once initialized, the rendering API layer does not perform any allocations, allowing for full control of memory by the caller. In addition, sub-hierarchies of GPU data structures can be precomputed in memory for more efficient CPU usage (at the cost of larger memory footprint).

Shaders

The shader API layer provides functions to interrogate the output of the PlayStation®Vita shader compiler for the parameters exposed by the shader. Each parameter can in turn be interrogated for how it maps to a hardware resource or memory location.

Shader Patcher

The shader patcher API layer provides a more traditional API for manipulating shaders. It patches the output of the shader compiler with parameters usually expected to be provided at runtime, such as blend modes and vertex formats.

Terminology

This document uses the following terminology to describe libgxm and GPU hardware functionality:

- **Uniforms:** Shader constants declared in the shader source code, set by the API user.
- **Scene:** A group of draw calls to a particular render target. A scene generally maps to exactly one job being submitted to the GPU firmware layer.
- **Tile:** A 32 x 32 block of samples. This is 32 x 32 pixels when not using multi sample anti-aliasing (MSAA), 16 x 32 pixels when using 2x MSAA or 16 x 16 pixels when using 4x MSAA. During fragment processing, each tile of a scene is processed separately. During the processing of a tile, the colors and depth/stencil values are stored on-chip.
- **Macro Tile:** A group of tiles arranged in a higher-level grid pattern over the render target. Primitives that span multiple tiles are not stored for each tile they intersect, but for each macro tile they intersect.
- **Primitive Block:** Serialized triangle data that encodes the output of the vertex program. The primitive blocks are stored once for each macro tile, and shared between all tiles in the macro-tile.
- **Spans:** A group of pixels in a tile that are visible after depth and stencil testing and hidden surface removal. All pixels in a particular span are from the same triangle.

Debug Version of the Library

A *debug version* of libgxm is provided for use during development. See the *Development Kit Neighborhood Settings Guide* for information on how to enable this setting.

Document serial number: 000004892117

000004892117

Differences from the Release Version of libgxm

The debug version of libgxm provides the following features to aid in title development

- Additional GPU state is validated on calls to `sceGxmDraw()`, `sceGxmDrawInstanced()`, `sceGxmDrawPrecomputed()`, `sceGxmPrecomputedFragmentStateSetTexture()`, `sceGxmPrecomputedFragmentStateSetAllTextures()`, `sceGxmPrecomputedVertexStateSetTexture()`, and `sceGxmPrecomputedVertexStateSetAllTextures()`.
- Errors reported by libgxm functions are output to TTY.
- Internal asserts are enabled.

Note that CPU and memory load is increased when using the debug version of libgxm. It is advised that performance analysis of a title is performed while using the *release version* of libgxm.

Tools Version of the Library

To allow tools code to inspect the output of the shader compiler, a tools version of the library is provided in both `sdk/host_tools/lib` and `sdk/host_tools/lib.x64`. This is provided as a 32-bit or 64-bit `gxm_tool.dll`, which implements only the shader API of libgxm (i.e. all functions prefixed with `sceGxmProgram`).

Files

In target code, all libgxm functionality can be included by including the following file:

```
#include <gxm.h>
```

In tools code, the shader API of libgxm may be included by including the following file:

```
#include <gxm/program.h>
```

Sample Programs

Graphics sample programs can be found in the `sample_code/graphics` folder of the PlayStation®Vita SDK.

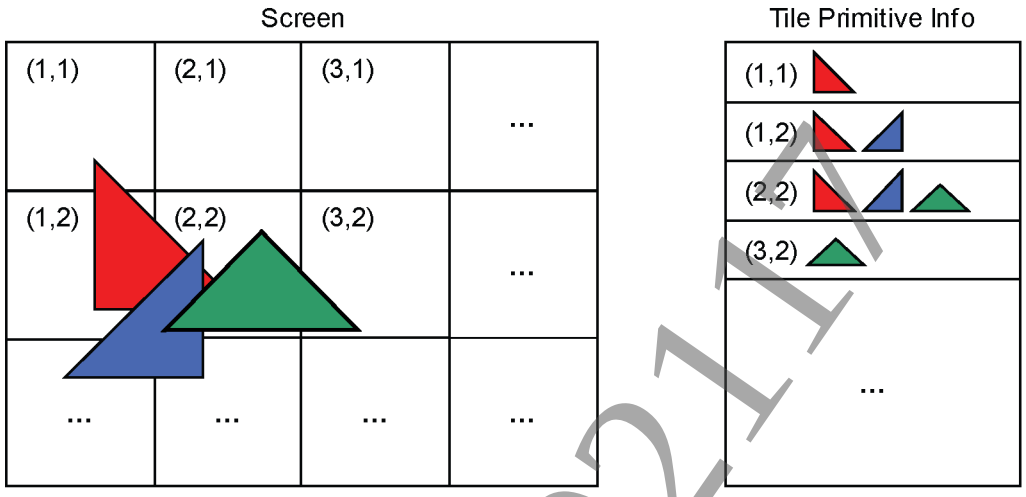
000004892117

Document serial number: 000004892117

3 Tile-Based Deferred Rendering

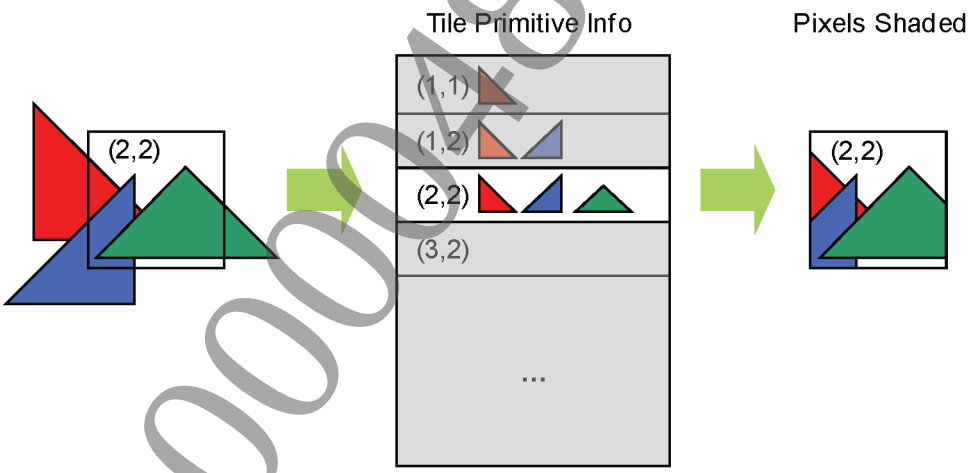
In tile-based deferred rendering, the screen is split into multiple tiles and primitives are binned according to which tiles they intersect. Figure 1 shows the binning of primitives by tile. This binning process occurs after vertex processing and before any pixel is shaded.

Figure 1 Primitives Overlaying Tiles



After the tiles have been constructed, each tile is rendered independently, as shown in Figure 2.

Figure 2 Each Tile Rendered Independently



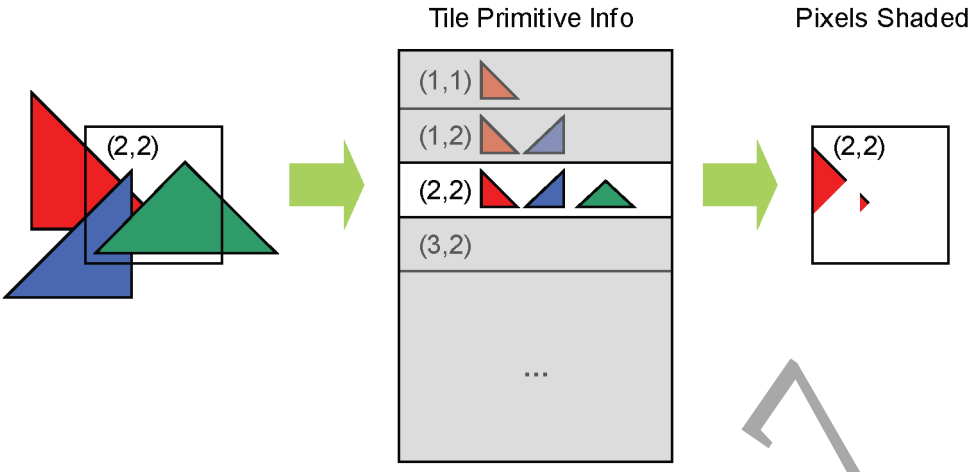
The GPU contains hidden-surface-removal hardware, so where tiles contain multiple overlapping opaque primitives, only the visible pixels are shaded. This ensures that the shader unit does not waste cycles shading pixels that are ultimately not visible. This optimization is only present for opaque primitives. Translucent primitives that use blending or color masking do not benefit from hidden surface removal, because all translucent layers must be shaded.

Primitives are still shaded in order, however, so in the example in Figure 2, after only the red triangle has been processed, the (incomplete) tile would appear as shown in Figure 3.

Document serial number: 000004892117

000004892117

Figure 3 First Primitive Shaded



As shown in the next section, the SGX architecture has further efficiencies from employing a tiled rendering algorithm, since much of the data used when shading each tile can be kept on-chip for the duration of the tile.

000004892117

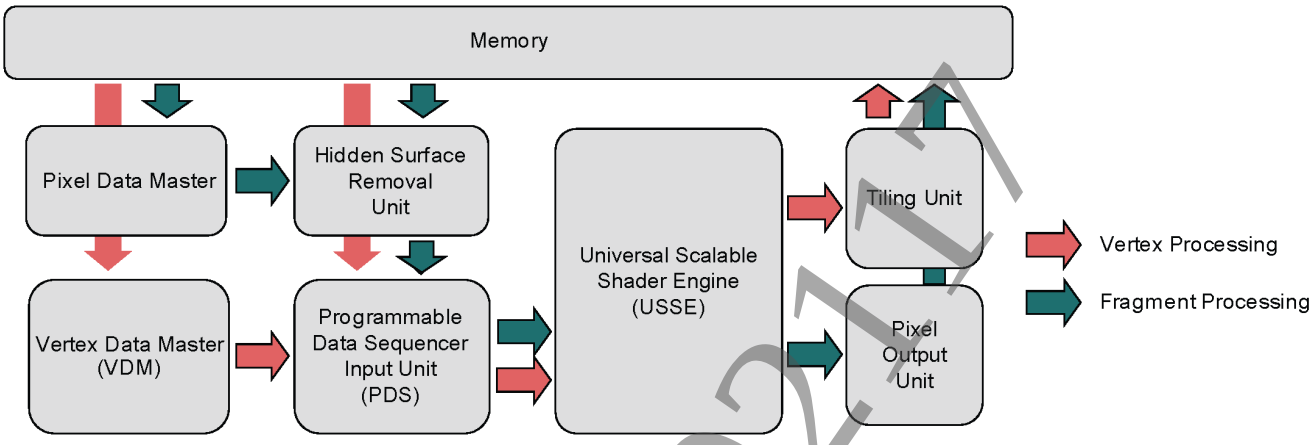
Document serial number: 000004892117

4 SGX Architecture

SGX Block Diagram

As shown in Figure 4, data flows through the GPU twice: once for vertex processing and once for fragment processing. The next sections describe these two data flows.

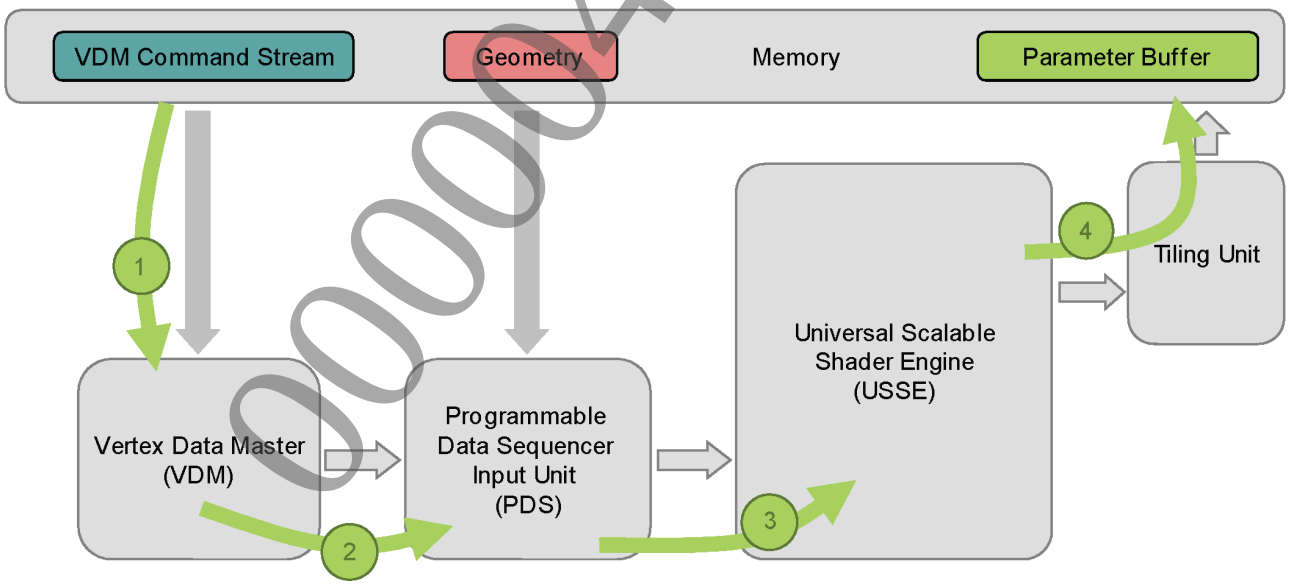
Figure 4 SGX Processing Pipelines



Vertex Processing

Figure 5 shows a high-level view of the vertex processing pipeline.

Figure 5 Vertex Processing Pipeline



The data flow is as follows:

(1) The *VDM Command Stream* is read from memory by the *Vertex Data Master* (VDM). This is the front-end command stream for SGX, containing very high-level commands such as *Draw* or *Update State*.

(2) The VDM issues commands to the *Programmable Data Sequencer Input* unit (PDS), which is responsible for the de-indexing and DMA-fetch of geometry from memory.

Document serial number: 000004892117

©SCEI

(3) The PDS issues commands to the *Universal Scalable Shader Engine* (USSE) to start particular vertex programs.

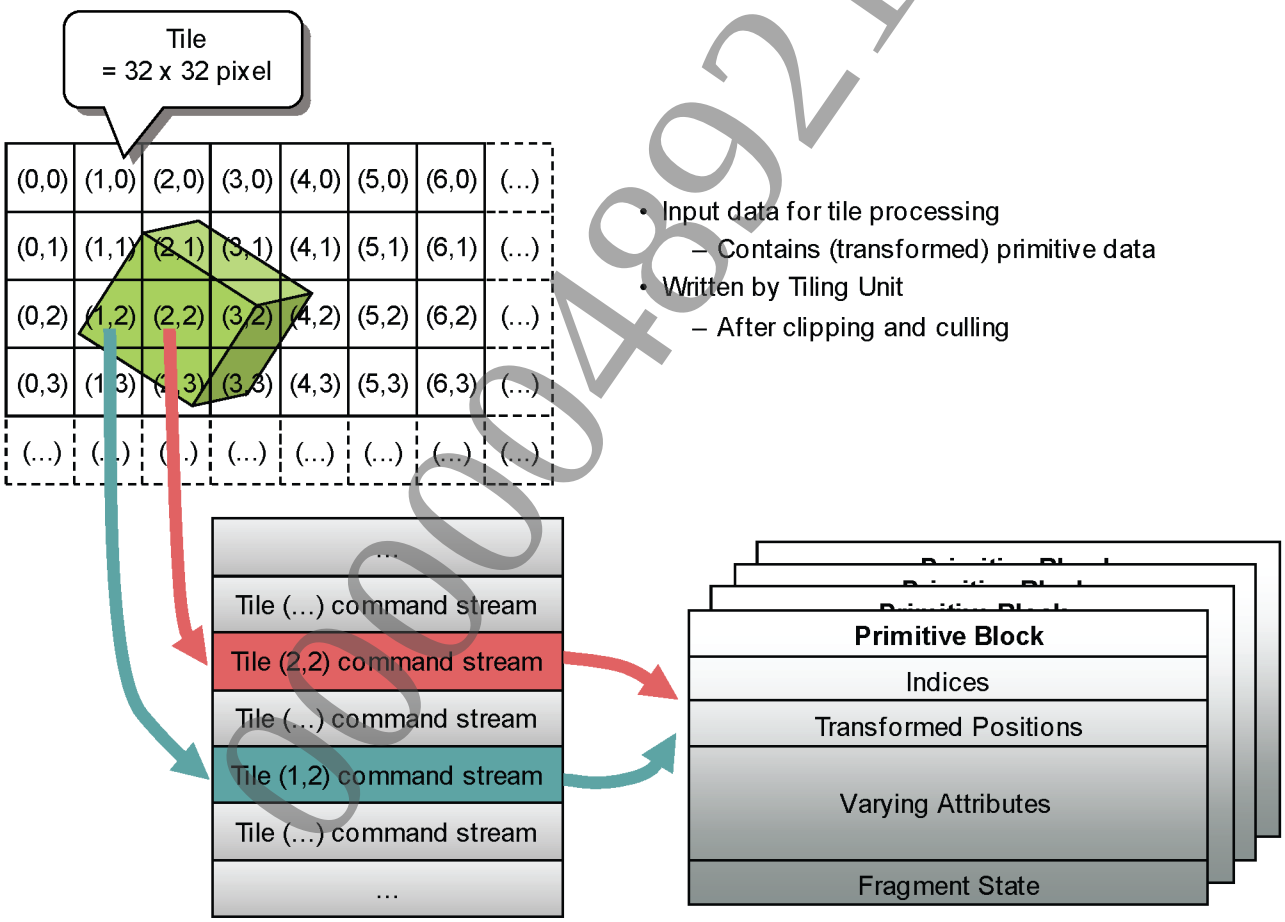
(4) The USSE outputs fully shaded vertices to the *Tiling Unit*. The tiling unit bins geometry by tile according to which tiles the geometry intersects, and it writes all the data out to memory. This data structure of geometry-indexed-by-tile is called the *Parameter Buffer*.

Parameter Buffer

The *parameter buffer* is an area of memory used by the GPU to store the output of vertex processing, indexed by tile. This memory is not touched by the CPU after allocation; it is managed directly by the GPU firmware layer during use.

Within the parameter buffer, each tile has a command stream. As shown in Figure 6, these command streams point to serialized triangle data, called a *primitive block*. The primitive block encodes the outputs of the vertex program, including all interpolants and some state that associates the triangles with a fragment program. Storage for these data structures is allocated in pages from a parameter buffer memory area set up in advance. This page allocation mechanism (and corresponding free mechanism) is implemented in hardware; after this memory has been allocated, it does not need to be manipulated by the CPU.

Figure 6 Parameter Buffer Contents

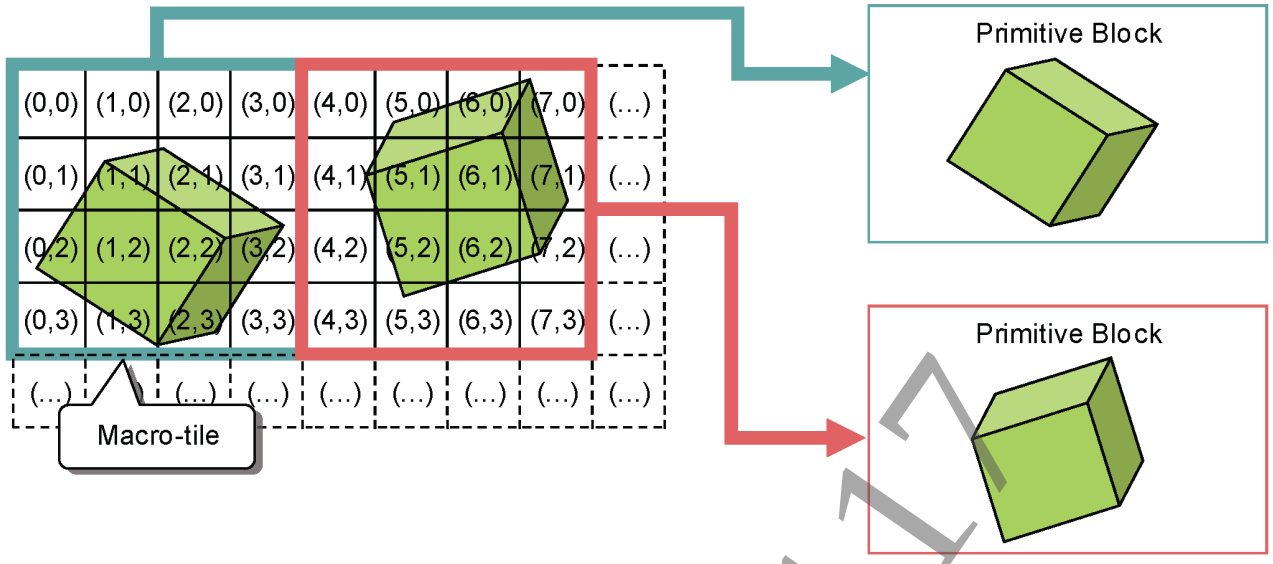


To reduce the total amount of memory required to store the parameter buffer, primitive blocks are not stored per tile, but per macro tile. Each primitive block is shared by multiple tile command streams, with a local mask for each tile to describe which primitives in the primitive block intersect with that tile.

Document serial number: 000004892117

000004892117

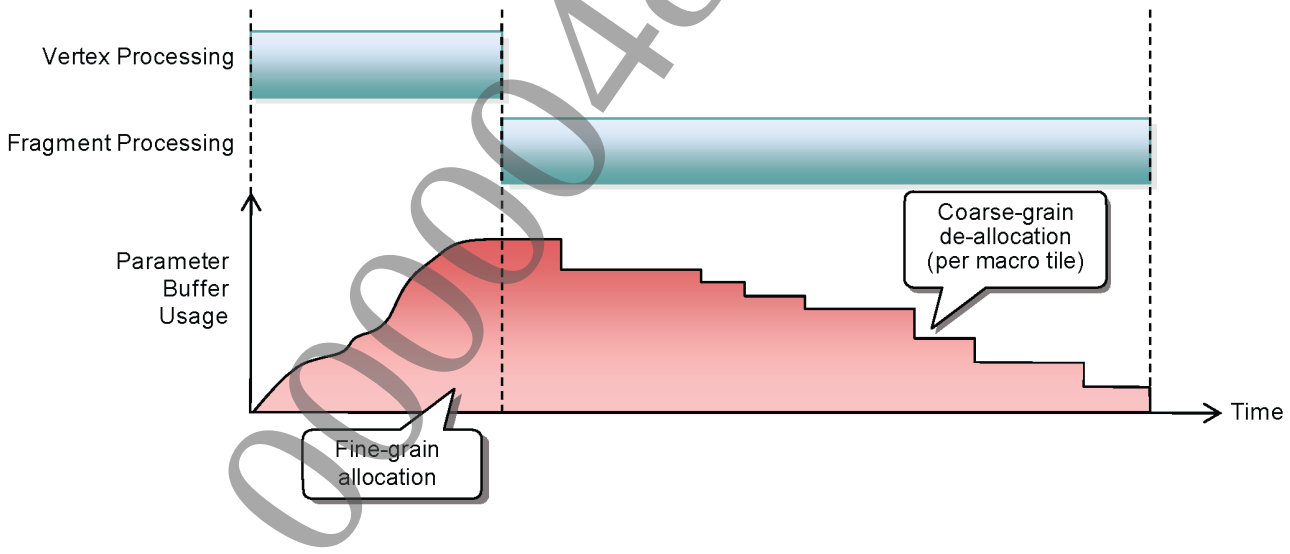
Figure 7 Primitive Blocks Shared by Multiple Tile Command-Streams



These groups of tiles are called *macro tiles* and are arranged in a higher-level grid pattern over the render target. As shown in Figure 7, triangles that span multiple tiles are not stored for each tile they intersect, but are stored for each *macro tile* they intersect.

Because primitive blocks are potentially used by every tile within a macro tile, this has a knock-on effect on how the GPU manages the de-allocation of pages within the parameter buffer. In short, primitive blocks can only be de-allocated when *all* the tiles of a given macro tile have been processed, leading to the graph of memory usage shown in Figure 8 as all the draw calls to a particular render target are processed.

Figure 8 Memory Usage for Draw Call Processing



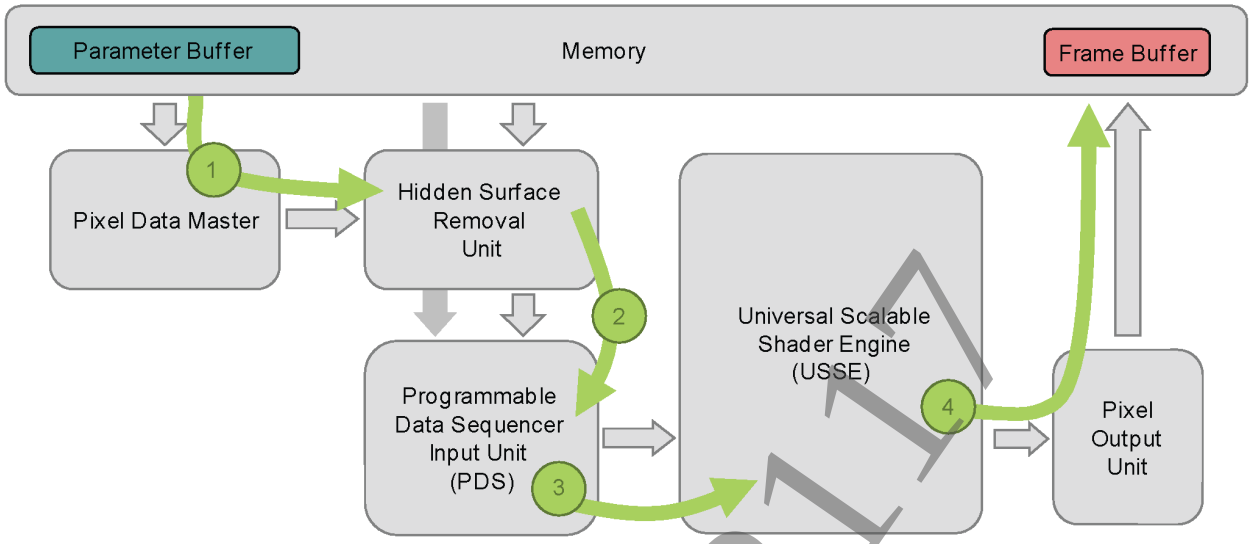
Document serial number: 000004892117

000004892117

Fragment Processing

Figure 9 shows a high-level view of the fragment processing pipeline.

Figure 9 Fragment Processing Pipeline



The data flow is as follows:

(1) The *Pixel Data Master* (PDM) executes the tile command stream from the parameter buffer. All active primitives are sent to the hidden surface removal unit for processing.

(2) The hidden surface removal unit first performs depth and stencil testing. For the pixels that pass depth and stencil, it decides which are actually visible, groups these into *spans*, and sends the spans to the PDS. All pixels within a particular span use the same fragment program.

(3) The PDS is responsible for starting the fragment program for all the pixels of the span.

(4) After the entire tile has been rendered, it is sent to the *Pixel Output Unit* to be sent to memory in a single transfer. Internally to this unit, the tile can be downscaled (if using MSAA) and converted to a particular format.

000004892117

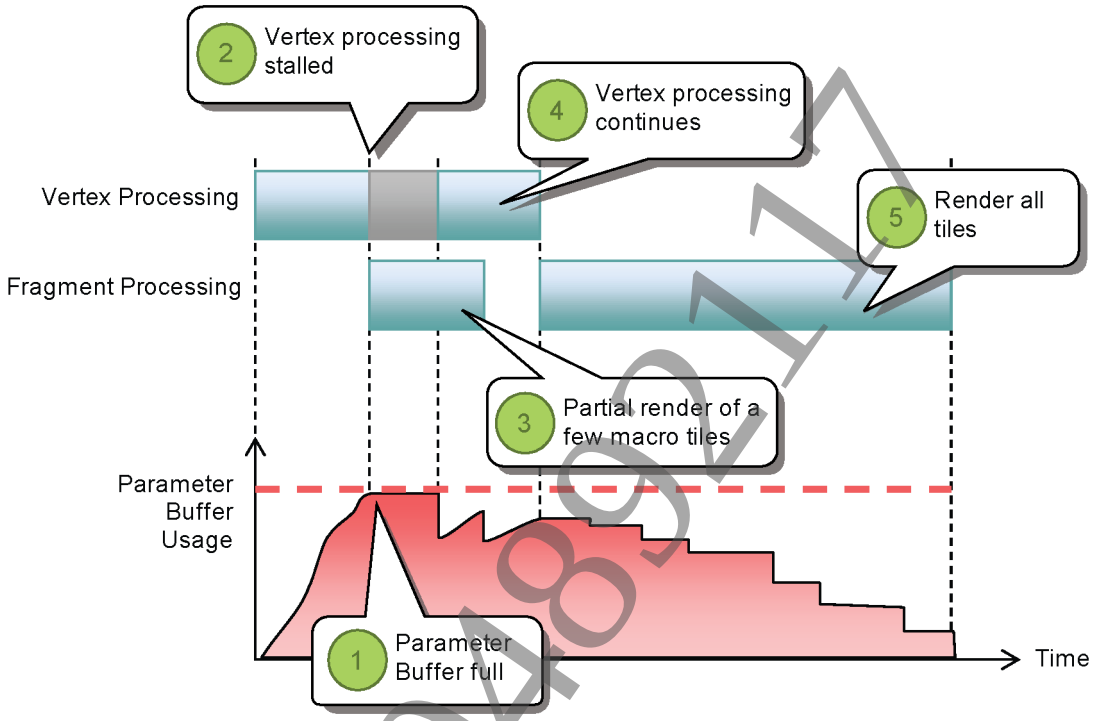
Document serial number: 000004892117

Partial Renders

For best performance, the parameter buffer should be large enough to store the vertex processing output for all of the draw calls in a particular scene. This ensures that hidden surface removal has all the information it needs to avoid shading invisible pixels, and means that the system only has to store the pixel outputs to memory once per tile.

However, the hardware is capable of handling the case where the parameter buffer becomes full, as shown in the memory usage diagram in Figure 10.

Figure 10 Memory Usage During Partial Render



The time line is as follows:

(1) Vertex processing starts and quickly fills the parameter buffer

(2) The firmware layer (described later) detects that vertex processing is stalled and starts rendering pixels early.

(3) After a full macro tile has been processed, the memory is freed.

(4) Vertex processing can now continue writing into the parameter buffer.

(5) After all vertices have been processed, rendering starts as normal. Because some tiles have already been processed, this means that some tiles will be processed for a second time.

For this *partial render* mechanism to guarantee correct output, memory must be used for depth/stencil values to preserve them between separate executions of each tile. This leads to a general recommendation on SGX to provide memory for the depth/stencil data, but to configure rendering such that the memory is **only** used if a partial render occurs. This is the default behavior of the libgxm API.

Document serial number: 000004892117

000004892117

Key Platform Differences

In many areas the SGX architecture is quite different from the standard forward-rendering model you may be used to. This section describes the important differences.

Current Tile Pixels Are On-Chip

The pixel values for the current tile are held on-chip while that tile is processed (also see blending in USSE Responsibilities). This means that blending is done in the format of the on-chip storage, not the format in memory.

In addition to handling precision differences, the GPU can downscale from sample level to pixel level when storing tiles to memory after fragment shading is complete.

Depth/Stencil Bandwidth Is Optional

The GPU keeps the current depth/stencil/mask values on-chip for the current tile; therefore, while a tile is being processed, there is no external bandwidth cost for depth/stencil/mask tests. As a result, the depth, stencil, and mask tests are always enabled. On-chip depth and stencil values are always at F32 and U8 precision respectively, regardless of the memory format being used. The on-chip mask value is 1 bit.

If the depth, stencil, or mask values are not needed for a future pass, they do not need to be stored to memory. Similarly, if the existing depth/stencil/mask values in memory are not needed for the current scene – for example, if they are going to be reset to known values – they should not be loaded from memory.

If both of these conditions are met (that is, depth/stencil/mask values are not needed for either a future pass or the current scene), depth/stencil/mask processing has zero bandwidth cost. This is the default behavior of depth/stencil/mask surfaces in libgxm.

Memory for depth/stencil/mask values should always be provided to ensure that correct output is preserved even if a partial render occurs. The GPU firmware layer enables depth/stencil/mask reads and writes for just those tiles that could be rendered multiple times.

No Hardware Clear

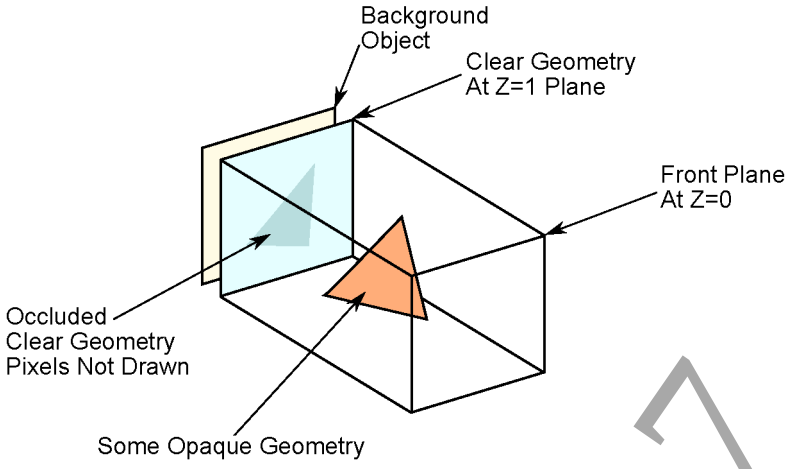
When a tile starts to be rendered, the current pixels are undefined. In addition, the hardware does not have any commands for a "clear" of the tile; everything is geometry.

When a tile starts to be rendered, the contents of the on-chip storage for the current pixels is undefined. To ensure that these undefined values are never read, the hardware uses a concept called the background object. The *background object* is a textured quad that uses a trivial shader program that reads the color buffer as a texture using texture coordinates that map texels to pixels. This background object is injected into the tile before any tile geometry is processed. In this way, where the current pixel value is needed, it can be read using standard fragment shading. In the general case, this background object will be occluded and never incur a shading cost.

If you require a clear, send opaque, depth-always geometry that sits on the far plane to "clear" the buffer. Hidden surface removal ensures parts of this "clear" geometry that are occluded by future primitives are not rendered.

Document serial number: 000004892117

Figure 11 Background Object Occluded By Clear and Other Geometry



In the example shown in Figure 11, some clear geometry is rendered (explicitly by the caller) on the far plane. This immediately occludes the background object (shown moved slightly back), avoiding reading the existing framebuffer pixels. This clear geometry is in turn occluded (shown shaded) by some other opaque object within the tile; therefore, it is not run for all pixels.

The depth and stencil values of the background object can be set by the caller, meaning that the background object in the above diagram could exist anywhere in the [0,1] depth range. By default, it exists at the Z=1 plane. Note that this depth value is only used if the depth/stencil surface has **not** been configured to load its values from memory at the start of each tile. The depth value of Z=1 can be interpreted as an easy way to clear depth/stencil values for the scene. For a depth/stencil surface that loads its values from memory at the start of each tile, the background object exists at the depth values that were loaded in.

USSE Responsibilities

Various parts of the graphics pipeline that are usually assumed to be handled by fixed function hardware are actually handled in shader code on this GPU. These include (but are not limited to) the responsibilities listed in Table 1.

Table 1 Shader Code Responsibilities

Responsibility	Description
Vertex Unpack	When shader programs are compiled, they assume that the input vertex attributes can be read as 32-bit floating point values from input registers. Because vertex attributes are usually compressed in memory, and the PDS unit is only doing a DMA from memory into the USSE, this is not always the case. Therefore, vertex programs usually require extra code to unpack each attribute from its compressed representation. This code can be patched in at runtime when the shader patcher API is used (see Chapter 12, Using the Shader Patcher API). It is not yet possible to specify the vertex unpack in shader source code to avoid this patching process, but support for this is planned.

Document serial number: 000004892117

000004892117

Responsibility	Description
Blending	Many hardware architectures for rendering 3D graphics include a fixed function unit that blends the outputs of the fragment program with the color values already stored in memory. The SGX architecture removes the need for this unit, because every fragment program has the color values for the current pixel on-chip while each tile is processed. As a result, blending is simply standard register-to-register arithmetic performed by the shader core. Preliminary support for programmable blending was added in SDK 0940. Please refer to the *Shader Compiler User's Guide* for details. When using programmable blending, shader code may read the current pixel value as an input for custom blending operations, but it is not possible to specify additional blending at runtime using the shader patcher.

000004892117

Document serial number: 000004892117

5 The libgxm Rendering Pipeline

Rendering Context

All rendering is performed via a rendering context, of which there are two types: immediate contexts and deferred contexts. Only one immediate context per game process is supported, but an application may create as many deferred contexts as necessary to achieve the desired level of multi-threading. Chapter 8, Threading describes multi-threaded rendering.

The rendering context exposes the pipeline as shown in Figure 12.

Figure 12 Rendering Pipeline

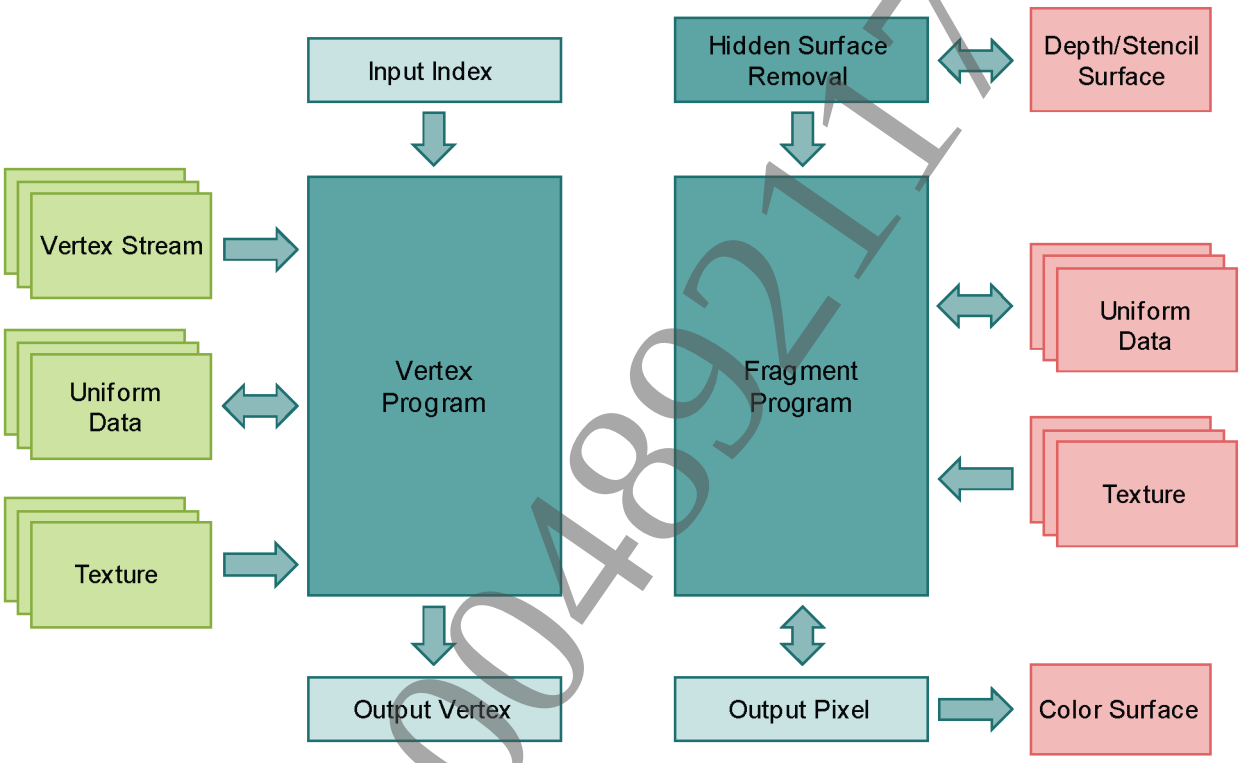


Table 2 describes each item in the rendering pipeline.

Table 2 Rendering Pipeline Components

Item	Description
Input Index	Originally a value in an index buffer, the Input Index represents the input to a vertex program. Part of the vertex program fetches the vertex data for this index value.
Vertex Stream	The base address of a vertex stream. The libgxm context supports up to 4 vertex streams, containing up to 16 vector attributes among the 4 streams.
Uniform Data	Each pipeline exposes two types of uniform data: user-declared uniform buffers (declared in shader code), and the default uniform buffer. Both types of uniform data are loaded from memory. The libgxm context supports up to 14 vertex and 14 fragment uniform buffers in addition to the default buffers. See Uniforms. Writable uniform buffers are also supported, allowing vertex and fragment programs to store arbitrary data to memory.

Document serial number: 000004892117

Item	Description
Texture	A texture is defined by its texture control words. These are set by value on the rendering context for use by the vertex or fragment program. The libgxm context supports up to 16 vertex and 16 fragment textures.
Output Vertex	The outputs of the vertex program, to be picked up by the tiling unit and serialized into the parameter buffer.
Hidden Surface Removal	The unit that will ultimately cause pixels to be shaded, after they have passed the depth/stencil test.
Depth/Stencil Surface	The description of when and how a depth/stencil surface is stored to memory. You need to provide memory to properly handle partial renders, but the hardware need not use the memory during normal rendering.
Output Pixel	The output of the fragment program. **Note**: While a tile is being shaded, the value is stored on-chip and manipulated by the fragment program using a register.
Color Surface	The description of how the fully shaded tiles are stored to memory.

Due to the SGX architecture, there are various items not in this pipeline that you might expect in other rendering APIs such as OpenGL, as listed in Table 3.

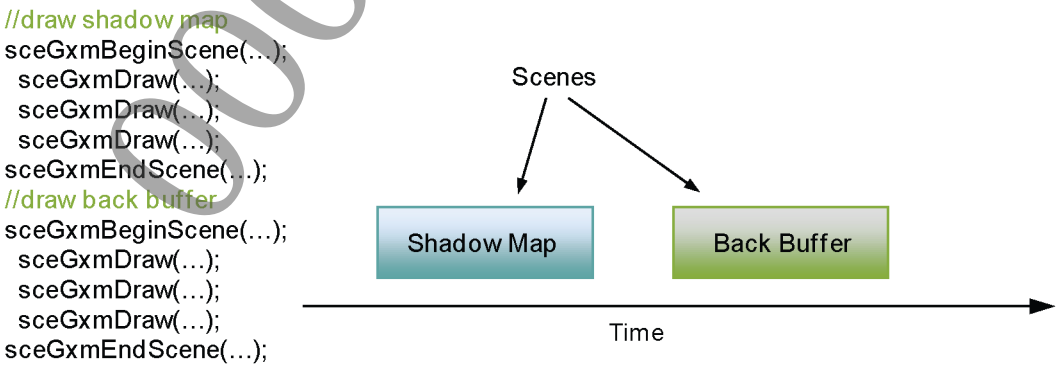
Table 3 Pipeline Differences from Open GL

Item	Description
Blend state	As mentioned in USSE Responsibilities, blending is shader code and not state. As such, the libgxm context expects shaders to already contain the relevant code for blending. For convenience, the shader patcher API allows for blending code to be specified at runtime rather than at compile time.
Vertex Formats/ Declaration	Also mentioned in USSE Responsibilities, vertex unpack is shader code and not state. As such, the libgxm pipeline only requires stream base addresses, assuming that the vertex program already contains code to unpack the data. For convenience, the shader patcher API allows for this unpack code to be specified at runtime rather than compile time.

Scenes

Rendering in libgxm is organized into *scenes*, where a scene is defined to be a collection of draw calls to the same render target. Figure 13 shows an example of a game frame with two scenes.

Figure 13 Example Game with Two Scenes



This structure allows the caller to explicitly control how jobs are generated and submitted to the GPU firmware layer.

Scenes may only be created using the immediate context. Under normal conditions, when the immediate context buffers are large enough to contain data for all draw calls within the scene, exactly one job is

Document serial number: 000004892117

created per scene. If the immediate context runs out of memory, then the scene is split into multiple jobs, and GPU performance may degrade as a result. Details on how to set these sizes and detect these out-of-memory conditions are described in Internal Ring Buffers.

Deferred contexts create *command lists*, which can be later executed from with a scene on the immediate context. For more details see Chapter 8, Threading.

Firmware Overview

Firmware Layer Responsibilities

The firmware layer is loaded by the OS during initialization of the GPU. This layer is privileged, and responsible for various tasks:

- Power management.
- Suspend/resume.
- Receiving and executing jobs.
- Sharing of the GPU between multiple processes.

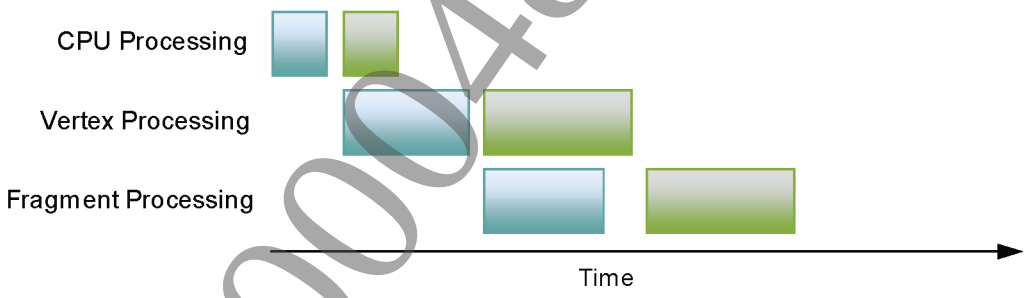
Jobs and Overlap

When jobs are submitted to the firmware layer, they enter a queue. This queue allows for the CPU to continue submitting more jobs without being blocked by GPU processing. Each job in the queue is executed in a pipelined manner as GPU resources become available. The pipeline has two distinct stages:

- Vertex processing: all geometry is vertex shaded and written to the parameter buffer, binned by tile.
- Fragment processing: each tile is executed in turn to shade the visible pixels.

Figure 14 shows how the stages of multiple jobs overlap.

Figure 14 Job Overlap



Note that this leads to some natural overlap between scenes: it is likely that the vertex processing of one scene will overlap with the fragment processing of the previous scene.

000004892117

Document serial number: 000004892117

6 Using the Rendering API

Basic Procedure

Initialization

Call `sceGxmInitialize()` to initialize the library. Specify as arguments the size of the parameter buffer, and the configuration of the display queue callback for synchronizing display flip operations with the GPU.

Call `sceGxmCreateContext()` to create a rendering context. Specify as arguments the memory regions and sizes for the context ring buffers and memory for context object itself.

Call `sceGxmCreateRenderTarget()` to create a render target object. Specify as arguments the memory for the render target CPU data structures, and a memory block for the GPU data structures. A render target is required to start a scene and begin making draw calls.

Set up a color surface using `sceGxmColorSurfaceInit()`, passing in a pointer to memory for the pixels. To ensure rendering is correct even if a partial render occurs, set up a depth-stencil surface using `sceGxmDepthStencilSurfaceInit()`, passing in a pointer to memory for the depth and stencil values. By default, this memory is only used if a partial render occurs.

Starting a Scene

Call `sceGxmBeginScene()` to start collecting draw calls to a given render target. Specify as arguments the scene description, a color surface, and depth/stencil surface.

Set Vertex/Fragment Program

Call `sceGxmSetVertexProgram()` or `sceGxmSetFragmentProgram()` to set programs for rendering. It is only necessary to set a program if it changes; vertex and fragment programs persist indefinitely, even between scenes.

Reserve and Write the Vertex/Fragment Default Uniform Buffer

If the vertex or fragment program has uniforms in the default uniform buffer, then call `sceGxmReserveVertexDefaultUniformBuffer()` or `sceGxmReserveFragmentDefaultUniformBuffer()` to reserve memory from a suitable internal ring buffer. The uniform values must be written in full before the draw call.

When reserved, the default uniform buffer does not persist indefinitely. Setting a new program or ending the scene results the reservation being lost. This is covered in more detail in the Uniforms section later in this document.

Set Vertex Streams

Vertex streams only need to be set if they have changed since the previous draw call. Call `sceGxmSetVertexStream()` to set a stream base address by index. No other information about the stream is required by the rendering API, because libgxm expects that the unpacking instructions are part of the vertex program. See Chapter 12, Using the Shader Patcher API for more details on how this shader code can be patched in at runtime. Vertex streams persist indefinitely, so they only need to be set if they have changed since the previous draw call.

Set Textures

Textures only need to be set if they have changed since the previous draw call. Call `sceGxmSetVertexTexture()` or `sceGxmSetFragmentTexture()` to set a texture for either the

Document serial number: 000004892117

vertex or fragment pipeline. Textures persist indefinitely, so they only need to be set if they have changed since the previous draw call.

Set (User Declared) Uniform Buffers

Uniform buffers are exposed as base addresses. There is a separate set of uniform buffers for the vertex and fragment pipelines. Uniform buffer addresses persist indefinitely, so they only need to be set if they have changed since the previous draw call. These can be set by calling `sceGxmSetVertexUniformBuffer()` or `sceGxmSetFragmentUniformBuffer()`.

If a user-declared uniform buffer is marked as writable then a pointer to the appropriately mapped memory should be set as the base address of the uniform buffer.

Set State

All state (such as depth mode or polygon mode) persists indefinitely. If any state needs to be changed, it should be set before the draw call.

Draw Call

SGX architecture supports only indexed primitives. Call `sceGxmDraw()` or `sceGxmDrawInstanced()` to draw geometry.

Instancing is performed using a wrap count: after this number of indices has been rendered, the hardware increments an *instance count* and jumps back to the start of the index buffer. Vertex streams can be indexed using either the index value or this instance count.

Ending the Scene

After all draw calls to a given render target are complete, call `sceGxmEndScene()` to end the scene and submit the job to the GPU.

Internal Ring Buffers

Table 4 lists the buffers that need to be given sizes when creating the immediate context .

Table 4 Internal Ring Buffer Allocations

Buffer	Description
VDM Ring Buffer	Used for VDM commands (the front-end of the GPU). Typically needs to be a few hundred Kb.
Vertex Ring Buffer	Generic vertex data ring buffer, used for generated PDS programs and vertex default uniform buffer reservations. Typically needs to be a few Mb.
Fragment Ring Buffer	Generic fragment data ring buffer, used for generated PDS programs and fragment default uniform buffer reservations. Typically needs to be a few Mb.
Fragment USSE Ring Buffer	Dynamically generated fragment USSE code, used internally by the rendering context at scene boundaries. Typically very small, tens of Kb.

In the normal case, one scene maps to exactly one job. If the internal buffers of the libgxm context are too small, then multiple jobs will be issued for a scene in order to avoid an out-of-memory deadlock. Warnings are printed in TTY if this occurs, such as:

```
[libgxm] WARNING: The vertex ring buffer high-water mark has been passed,
the scene is being split into multiple jobs. Consider making this ring
buffer larger.
```

The buffers are checked in this way after each draw call, and if judged to be above a high-water mark the job is created early.

Document serial number: 000004892117

000004892117

To prevent loss of performance, choose ring buffer sizes to ensure that scenes are not split into multiple jobs under normal conditions. Note that this out-of-memory condition from ring buffer sizes being too small for a single scene is different to the partial-render case that occurs when the parameter buffer is too small to store the outputs of vertex processing. The expected performance loss is different between the events, as listed in Table 5.

Table 5 Out-Of-Memory Events and their Performance Impact

Event	Expected Performance Loss
VDM/Vertex Ring Buffer High-Water Mark	Low impact on performance. libgxm needs to free up memory in ring buffers used by the vertex pipeline, so submits a job that does only vertex processing work for draw calls so far. This job adds data to the parameter buffer, but does not trigger the transition to fragment processing. When the scene is ended, libgxm subm its the remaining draw calls. The fragment pipeline is unchanged because the parameter buffer still contains all draw calls, and each tile is still processed only once.
Fragment/USSE Ring Buffer High-Water Mark	High impact on performance. libgxm needs to free up memory in ring buffers used by the fragment pipeline but cannot perform fragment processing without first doing vertex processing. It also cannot know which tiles are active during fragment processing, so libgxm must do fragment processing over the entire render target. As such, libgxm will submit a job to do both vertex and fragment processing for draw calls so far. To the GPU firmware layer, this looks like an additional scene. Every tile is processed more than once.
Partial Render	Medium impact on performance. The GPU firmware layer will render only as many tiles as needed to free up parameter buffer memory. This is likely to be less than the whole render target, only a subset of the tiles are processed more than once.

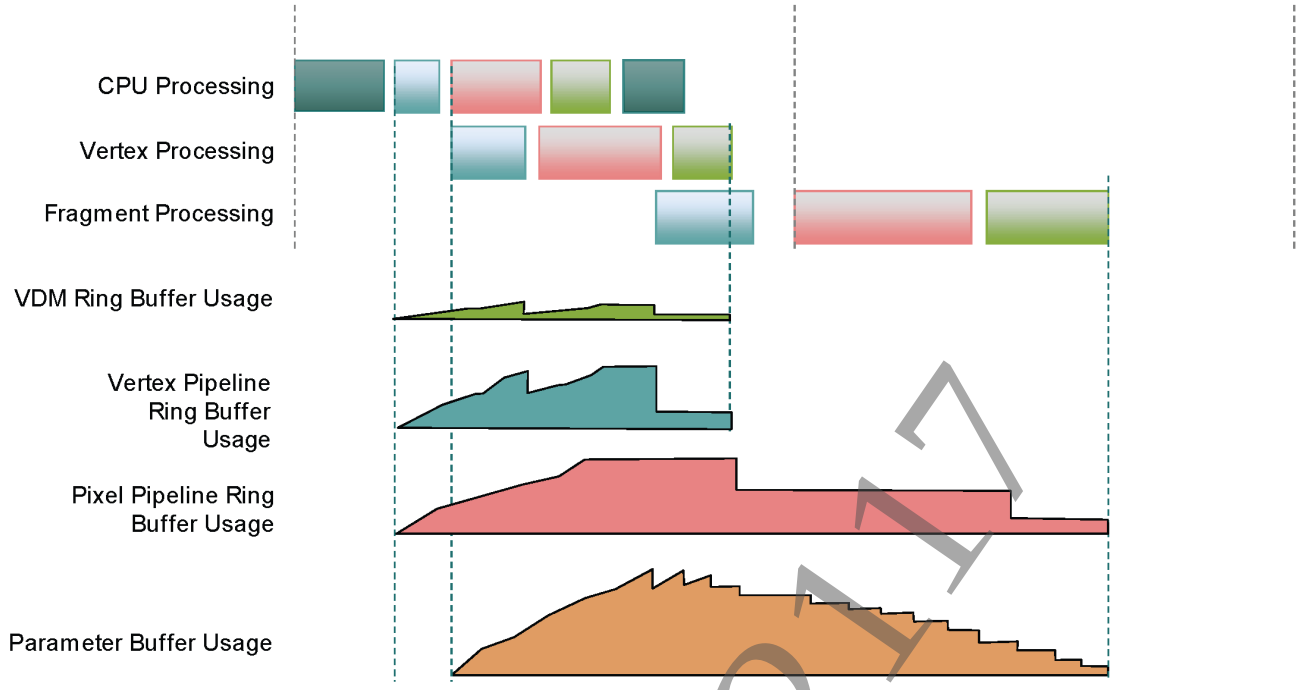
Buffering and Latency

For a typical game, we expect that CPU, vertex, and fragment processing produce the usage patterns for the libgxm ring buffers and the hardware parameter buffer shown in Figure 15.

000004892117

Document serial number: 000004892117

Figure 15 Usage Patterns for Ring Buffers and Parameter Buffer



In Figure 15, ring buffers are allocated at fine grain by the CPU and marked as consumed after the job completes on the particular pipeline the data is associated with. The parameter buffer is allocated at fine grain by the GPU in hardware, and de-allocated in hardware after macro tiles complete.

In the steady state, you can expect that:

- The VDM and vertex pipeline ring buffer usage will likely not overlap between frames, so size these buffers around the data necessary for a single frame or less.
- The fragment pipeline ring buffer and parameter buffer usages will likely overlap between frames, so size these buffers assuming more than one frame's data will be stored at the same time.

Uniforms

Uniform data on SGX is always loaded from memory. Part of the design of libgxm is to allow the game code direct access to the final memory location from which the GPU reads the data, to avoid CPU overheads from copying the data internally for every draw call. For more information about reducing CPU overheads further, see Precomputation.

Uniform data can be read by the GPU using two paths:

(1) When the shader is changed, the PDS unit issues a DMA, loading the uniform data into registers before any USSE program is started. These registers persist until the shader changes.

(2) The USSE loads the data from memory directly. This operation can sometimes be performed in the secondary USSE program, which only runs when the shader changes, but usually has to be performed for each vertex or pixel (for example if it depends on a dynamic index).

Uniforms variables in shader code can be declared either as part of a uniform buffer or not. For example, consider the following declarations:

```
uniform float4 boneMatrices[3*96] : BUFFER[0];
uniform float4x4 wvp;
uniform float4 globalParam0 : BUFFER[3];
uniform float4 globalParam1 : BUFFER[3];
uniform float3 lightDir;
uniform half4 unusedParam;    // unreferenced by shader code
```

Document serial number: 000004892117

000004892117

Live uniform parameters that are not explicitly assigned a uniform buffer ("`wvp`" and "`lightDir`" in this example) are collected into an extra uniform buffer, called the *default uniform buffer*. Unreferenced uniform parameters not explicitly assigned a uniform buffer ("`unusedParam`" in this example) are removed by the shader compiler and do not take up space in the default uniform buffer. The shader compiler may reorder parameters within the default uniform buffer to optimise the generated shader code.

Uniform parameters that are explicitly assigned to a uniform buffer always take up space in that buffer, regardless of liveness, as this allows uniform buffers to be shared between programs without memory layout changes. The order of parameters within an explicitly assigned uniform buffer is defined by the order of declaration only.

For explicitly assigned uniform buffers, the shader programmer can use a pragma to specify whether the uniforms within it should be read dynamically from memory by the USSE (the default) or pre-loaded into registers by DMA before the shader runs. See the *Shader Compiler User's Guide* for more details of uniform buffer pragmas.

The memory allocations for both explicit uniform buffers and the default uniform buffer can be fully controlled by the user, providing only a base address to libgxm. Call `sceGxmSetVertexUniformBuffer()` or `sceGxmSetFragmentUniformBuffer()` to set the base address of an explicit uniform buffer by index. Call `sceGxmSetVertexDefaultUniformBuffer()` or `sceGxmSetFragmentDefaultUniformBuffer()` to set the default uniform buffer base address.

In addition to this full control usage model, the libgxm context provides a mechanism to reserve the default uniform buffer from its internal ring buffers, but with strict rules about the lifetime of this buffer.

Default Uniform Buffer Reservations

Any live uniforms that are not explicitly assigned to a uniform buffer in the shader source code are gathered together into what is called the "default" uniform buffer, which can be reserved from an internal ring buffer if required. This mechanism can be used to avoid having to manually allocate memory for the default uniform buffer and synchronize its lifetime. While every other item of state persists indefinitely on the libgxm context, a default uniform buffer reservation has a strictly controlled lifetime to ensure that it can be handled properly if any sort of out-of-memory condition occurs internally.

The default uniform buffer (for the vertex or fragment pipeline) must be reserved from within a scene or command list, and persists until one of the following conditions is met:

- The program for that pipeline is changed by `sceGxmSetVertexProgram()` or `sceGxmSetFragmentProgram()`.
- The default uniform buffer for that pipeline is overwritten by calling `sceGxmSetVertexDefaultUniformBuffer()` or `sceGxmSetFragmentDefaultUniformBuffer()`.
- The scene or command list is ended by `sceGxmEndScene()` or `sceGxmEndCommandList()`.

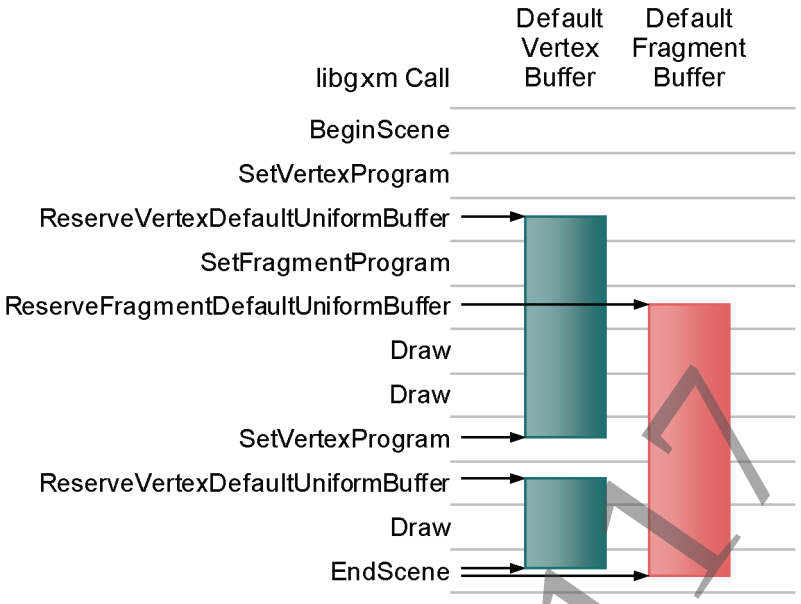
The values in the default uniform buffer must be completely written by the caller before the next draw call, because libgxm may have to create a job to start rendering the scene immediately after this draw call in order to handle an out-of-memory condition.

After the first draw call to use this buffer, the values must not be changed by the caller, but there is no limit to the number of draw calls that can share the same reservation provided the program is not changed and the scene is not ended. Figure 16 shows an example of these cases in action.

Document serial number: 000004892117

000004892117

Figure 16 Example Lifetime of Default Uniform Buffers



In Figure 16, the vertex default uniform buffer is shared by two sequential draw calls. The fragment default uniform buffer is shared by all draw calls in the scene.

> **Note:** In both cases, the data must be fully written to the uniform buffer between the reservation and the first draw call to use it.

If the default buffer is not properly set up for a draw call, the following error message will be printed on the TTY, and the draw call will return the following error code:

```
SCE_GXM_ERROR_UNIFORM_BUFFER_NOT_RESERVED:
    libgxm: ERROR: The fragment default uniform buffer has not been reserved
```

Writable Uniform Buffers

A user-declared uniform buffer may be marked as writable by using the `readwrite_buffer` pragma. Marking a user-declared uniform buffer as writable allows a vertex or fragment program to store data to the corresponding area in memory that the buffer represents. See the *Shader Compiler User's Guide* for more details of uniform buffer pragmas. These buffers can be aliased by other rendering resources such as vertex buffers or textures as long as the appropriate synchronization mechanisms are used. See Writable Uniform Buffers.

Textures

Textures are completely defined by the `SceGxmTexture` structure, which is currently opaque. These control words can be manipulated by calling the `sceGxmTextureXXX` family of functions. See the *libgxm Reference* for more details.

YUV Profiles

The GPU can select from two color-space conversion matrices per scene when converting from YUV to RGB in the texture unit. These conversion matrices are managed as part of the fragment pipeline, so it is only possible to use YUV textures with the fragment pipeline (i.e., they may not be used as vertex textures).

To set the YUV conversion profile for CSC (color space conversion) matrix 0 and 1, call `sceGxmSetYuvProfile()`. This function may only be called outside of a scene.

Each texture selects which CSC matrix to use for conversion as part of the texture format swizzle.

Document serial number: 000004892117

Render Targets

A render target is required to start a scene. It encapsulates various GPU data structures used by the firmware layer during job scheduling. Memory for these data structures can either be automatically allocated by the GPU driver or an uncached LPDDR memblock can be provided explicitly. To query the minimum size of this memblock required by a render target, call `sceGxmGetRenderTargetMemSize()`, providing the render target width, height, multisample mode, and renders per frame.

A render target can be created by calling `sceGxmCreateRenderTarget()` and destroyed by calling `sceGxmDestroyRenderTarget()`. If a memblock was provided explicitly, it can be freed after the render target has been destroyed.

Creating render targets is a costly operation, so they should be created at game initialization time if possible. In addition, there is a limit to the number of scenes that may be queued against a single render target before the CPU is blocked by the GPU. To ensure a sufficient number of scenes can be submitted before the CPU blocks, a suitable number should be specified as the *scenes per frame* value when the render target is created, corresponding to the expected maximum number scenes that will be submitted for each frame that use this render target. The amount of memory required by a render target increases as this number increases, so it should be set to the minimum value required.

By default, starting a scene using a given render target will render to the entire render target. This can be limited to a rectangular sub-region by specifying a *valid region* during `sceGxmBeginScene()`. Tiles outside of the valid region never have fragment processing performed on them, so this can be used to safely render to a subset of the render target, similar to a viewport region on other platforms. The valid region cannot be changed during a scene.

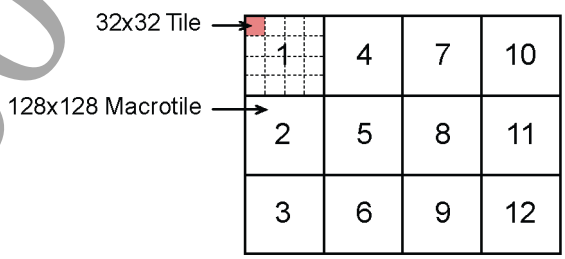
Macrotile Synchronization

In SDK 1.0, a feature was added to allow the adding of a synchronization point between each macrotile of a render target, allowing color surface or depth/stencil surface data associated with previous macrotiles to be safely used as texture data when shading the current macrotile. In particular this feature can be used to efficiently implement post processing chains that would otherwise require a large number of expensive render target switches.

All render targets split into macrotiles for processing on the hardware. Each macrotile is the same size, and each macrotile is always a multiple of 128 pixels in width and height. When configuring the GPU, every render target is either made up of 4 macrotiles in a 2x2 grid or 16 macrotiles in a 4x4 grid, but other configurations are implemented in the GPU firmware by skipping empty macrotiles.

Macrotiles are always started in a fixed order, forming vertical scanline through the render target. For example, for a 512x384 render target with 128x128 macrotiles in 4x3 configuration, the following order shown in Figure 17 will be used:

Figure 17 Macrotile Order Within A Render Target



When `SCE_GXM_RENDER_TARGET_MACROTILE_SYNC` is specified during render target creation, the GPU firmware will add a synchronization point between each macrotile to ensure tiles from the current macrotile do not start until all tiles from the previous macrotile have fully finished. It is this synchronization point that allows the previous macrotiles to be aliased as texture data.

Note that all draw calls for all macrotiles are still rendered as a single scene, the only change is to specify macrotile synchronization for the render target.

Document serial number: 000004892117

Since 2x2 is a supported mode on the hardware, this arrangement should be used if 4 or fewer macrotiles are required. For more than 4 macrotiles, an arrangement should be chosen that minimizes the number of unused macrotiles, which in turn will keep the number of synchronization points to a minimum.

When using macrotile synchronization, MSAA cannot be used and a custom valid region may not be used with `sceGxmBeginScene()`. These restrictions may be lifted in a future SDK release.

Color Surface

The `SceGxmColorSurface` structure describes how color data is stored to memory after the fragment pipeline for each tile is completed. When MSAA is used, the GPU can be configured to downscale the results *before* they are stored to memory, effectively removing the memory and bandwidth cost of MSAA color data. Extra care must be taken to avoid partial render in this case, because the samples will be resolved to pixels each time the tile is stored to memory. If partial render occurs, the quality of edge anti-aliasing could be reduced as a result.

The color surface can also be configured to gamma correct the results of the fragment program from linear to sRGB before storing to memory. More details can be found in the *GPU User's Guide*, as gamma correction in particular has some restrictions on the color formats and shader output formats it can be used with.

Using a color surface is optional, and can be omitted to perform depth-only rendering. This can be achieved by either passing in a NULL color surface to `sceGxmBeginScene()`, or by initializing a color surface using `sceGxmColorSurfaceInitDisabled()`.

Background Object

To handle the case where the first time a pixel is shaded it uses a translucent shader that blends with existing pixels, the GPU initializes each tile with an opaque fullscreen primitive called the "background object". As a result, a translucent object being rendered first causes these pixels to be flushed for shading using the background object shader. The background object shader fetches the existing color surface data using a texture read, and writes it to the on-chip color registers, ready for blending with the translucent object.

If the first primitive to affect a given pixel uses an opaque shader, such as an inexpensive quad drawn over the whole render target to clear the screen, hidden surface removal overwrites the background object pixels before shading, resulting in them having zero cost.

For non-MSAA scenes or MSAA scenes that use the color surface downscale, the background object uses a non-dependent texture read to load the existing color data from memory at pixel rate. For MSAA scenes that disable the color surface downscale, the background object must be loaded at sample rate to ensure that sample-level color differences are not lost. For this case the background object must use a dependent texture read (actually at sample rate). If the color surface pixels are not required for this case, an inexpensive opaque shader should be used with a fullscreen quad as the first draw call to allow hidden surface removal to skip the background object shader. The background object is not used for scenes that use depth-only rendering.

Depth/Stencil/Mask Surface

The `SceGxmDepthStencilSurface` structure describes how depth, stencil, and mask data is loaded and stored to memory. On SGX, depth, stencil, and mask testing is always on and processing at full precision within each tile: it is only the load and store to memory that is configured as part of this surface.

Depth/stencil/mask surfaces are always stored in memory at a sample rate, so a 4x MSAA depth buffer will consume 4x the memory of a single-sample depth buffer.

By default, when a depth/stencil/mask surface is initialized with a format and memory area to use, it only loads and stores to that memory if a partial render occurs. This allows the hardware to completely avoid the bandwidth cost of depth/stencil/mask testing, but assumes that your code does not need to read or write the values to memory for other reasons. To explicitly load the existing depth/stencil/mask

Document serial number: 000004892117

data from memory for all tiles, or to explicitly store the depth/stencil data to memory for all tiles, call `sceGxmDepthStencilSurfaceSetForceStoreMode()` or `sceGxmDepthStencilSurfaceSetForceLoadMode()` on the depth/stencil surface object.

Mask Testing

Each pixel (or sample, when rendering with MSAA) has a 1 bit on-chip value called the *mask bit*. Samples where the mask bit is 0 are assigned zero coverage, just as if they were outside of the bounds of the primitive during rasterization.

The mask bit test is performed immediately after rasterization, so rejected samples do not perform the depth test or stencil test (or depth write or stencil operations).

Mask Update

The mask bit is updated by drawing geometry using a *mask update* fragment program, which is created using the libgxm shader patcher function `sceGxmShaderPatcherCreateMaskUpdateFragmentProgram()`.

When geometry is drawn using this fragment program, the behavior of the GPU is changed as follows:

- The mask test is bypassed, all pixels or samples covered by the geometry update the mask bit.
- The depth and stencil tests are bypassed, all pixels or samples covered by the geometry are considered to pass the depth and stencil tests.
- Depth writes and stencil operations do not occur.
- The meaning of the stencil function is changed. When set to `SCE_GXM_STENCIL_FUNC_NEVER`, the mask bit is set to 0 for pixels or samples covered by the geometry. When set to `SCE_GXM_STENCIL_FUNC_ALWAYS`, the mask bit is set to 1 for pixels or samples covered by the geometry.

For example, a scissor window can be implemented using two draw calls using the mask update fragment program. The first draw call should be fullscreen and use `SCE_GXM_STENCIL_FUNC_NEVER` to set a 0 mask bit everywhere. The second draw call should use `SCE_GXM_STENCIL_FUNC_ALWAYS` and should cover the region where the mask bit should be set to 1.

Depth and Stencil Testing

Pixels that pass the mask test are processed for depth and stencil testing. The depth test is always at F32 precision regardless of the memory format of the depth data.

Background Depth/Stencil/Mask Values

If a surface does not use `SCE_GXM_DEPTH_STENCIL_FORCE_LOAD_ENABLED`, the initial values for the mask, depth, and stencil data for all pixels are taken from the surface background values. These may be set by calling `sceGxmDepthStencilSurfaceSetBackgroundMask()`, `sceGxmDepthStencilSurfaceSetBackgroundDepth()` and `sceGxmDepthStencilSurfaceSetBackgroundStencil()`.

Separate Load And Store Surfaces

The SGX supports using separate surfaces for loading and storing depth/stencil/mask data. This is exposed in libgxm using the `sceGxmBeginSceneEx()` function, which has identical behaviour to `sceGxmBeginScene()` but accepts two depth/stencil/mask surface descriptions.

If partial render occurs, the scene must be processed by the GPU using multiple fragment processing passes, resulting in tiles being processed more than once. In this case, the GPU firmware will ensure that only the first pass of a tile will use the load surface for reading and store surface for writing, with subsequent passes using the store surface for both reading and writing. This ensures that the load surface is never written to, even if partial render occurs. As a result, the format of the store surface must be a

Document serial number: 000004892117

000004892117

superset of the format of the load surface to ensure all data can be correctly preserved through partial render.

If separate load and store surfaces are used, the load surface should set a load mode of `SCE_GXM_DEPTH_STENCIL_FORCE_LOAD_ENABLED` to actually be used by the hardware, otherwise the background depth/stencil/mask values will be used instead. The debug version of libgxm will warn if separate surfaces are provided but this load mode is not set.

Clipping

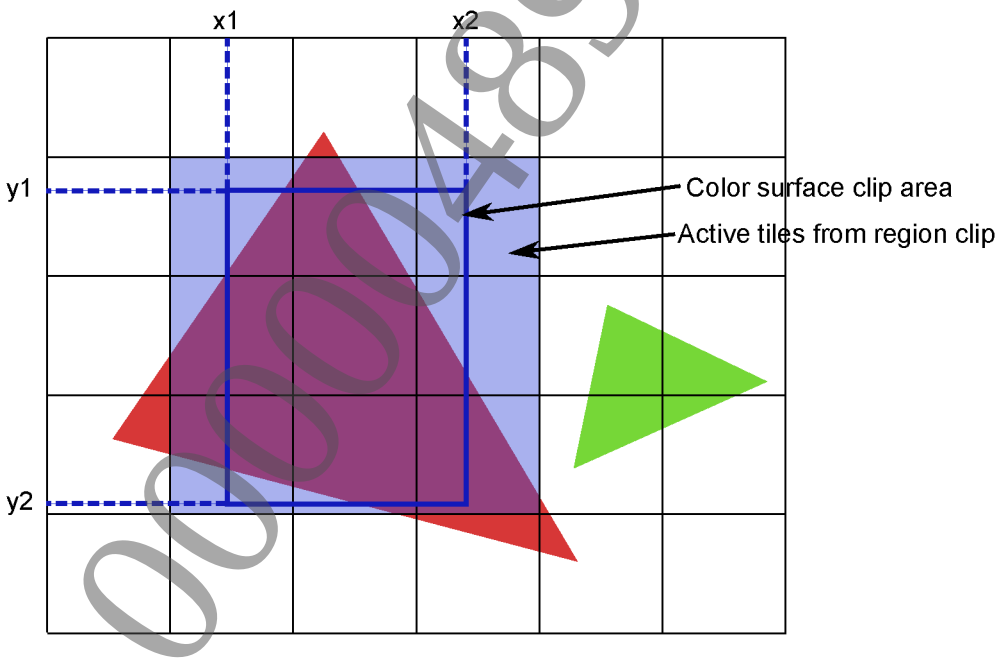
The GPU performs frustum clipping and supports up to 8 user clip planes. Clipped geometry is then transformed to screenspace using the offsets and scaling factors set in `sceGxmSetViewport()`. These results are then scissored using a combination of *region clip* and *color surface clip*.

- **Region clip:** a rectangle that defines which tiles are active during tiling. Tiles that are not active do not receive geometry and do not write data to the parameter buffer.
- **Color surface clip:** a rectangle that defines which pixels are active when a completed tile is stored back to the color surface in memory.

> **Note:** Color surface clip does not save any shading cost within a tile, but can be used to clip the output at pixel granularity.

In Figure 18, both region clip and color surface clip are set to the region bounded by x1,y1 and x2,y2. Only pixels within the color surface clip (shown as a thick rectangle) are stored to memory, and only the tiles that contain this region are active (shown shaded).

Figure 18 Clipping using Region Clip and Color Surface Clip



When `sceGxmBeginScene()` is called, region clip is reset to the valid region (which defaults to the whole render target if not specified). To change the region clip, call `sceGxmSetRegionClip()` within the scene. Region clip is used only during tiling; therefore, the values can be changed between draw calls.

When a color surface is initialized using `sceGxmColorSurfaceInit()`, the clip is initialized to the whole color surface. To change the clip rectangle after initialization, call `sceGxmColorSurfaceSetClip()`.

The mask bit can be used to clip at pixel or sample granularity. Unlike color surface clip, the mask bit test is performed before shading, so pixels or samples outside the region of interest do not have any shading

Document serial number: 000004892117

cost. Note that because the mask bit is updated using geometry, any shape may be used for the mask area; it is not restricted to a rectangular region as is the region clip or color surface clip.

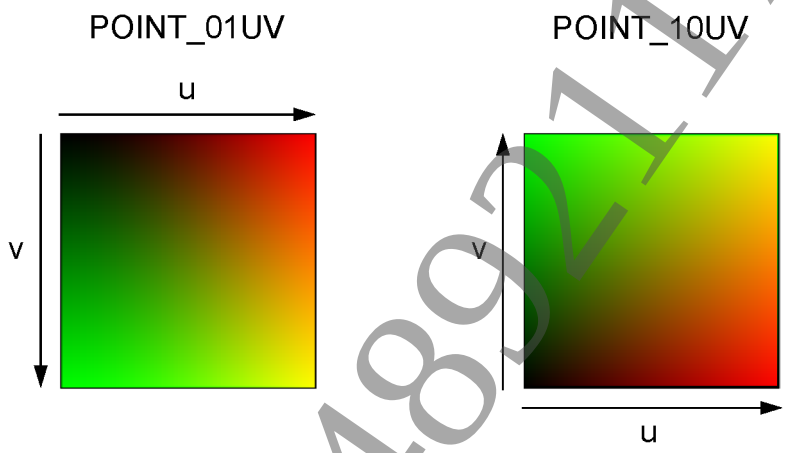
If the viewport transform is disabled using `sceGxmSetViewportEnable()`, frustum clipping and user clip planes are also disabled, but region clip and color surface clip can still be used.

Point Sprites

When rendering using a primitive type of `SCE_GXM_PRIMITIVE_POINTS`, the GPU can generate texture coordinates over the point primitive. These coordinates can be read using the `SPRITECOORD` semantic in the fragment program.

Generated texture coordinates can be used only if the polygon mode is `SCE_GXM_POLYGON_MODE_POINT_01UV` or `SCE_GXM_POLYGON_MODE_POINT_10UV`. For these supported modes, the generated coordinate will have the values over the sprite shown in Figure 19.

Figure 19 Sprite Coordinates for Supported Polygon Modes



For polygon modes other than `SCE_GXM_POLYGON_MODE_POINT_01UV` or `SCE_GXM_POLYGON_MODE_POINT_10UV`, the `SPRITECOORD` semantic cannot be read in the fragment program.

The `SPRITECOORD` semantic can be read independently of any other texture coordinate. All other texture coordinates are constant over the point primitive. It is not necessary to write any texture coordinates in the vertex program to be able to use `SPRITECOORD` in the fragment program.

Visibility Testing

The GPU can accumulate visibility counters in memory.

> **Note:** These counters are updated purely based on the pixels that pass the depth/stencil test, and are not affected by hidden surface removal.

The visibility counters are implemented by each of the 4 GPU cores having its own visibility result buffer with up to 16384 entries. These buffers are updated independently by each GPU core during fragment processing.

Each draw call may be associated with an offset within these per-core buffers, by calling `sceGxmSetFrontVisibilityTestEnable()` to enable visibility testing, and `sceGxmSetFrontVisibilityTestIndex()` to set the counter index. The operation to perform when a pixel passes the depth/stencil test can be set by calling `sceGxmSetFrontVisibilityTestOp()`. There are two possible operations:

Document serial number: 000004892117

000004892117

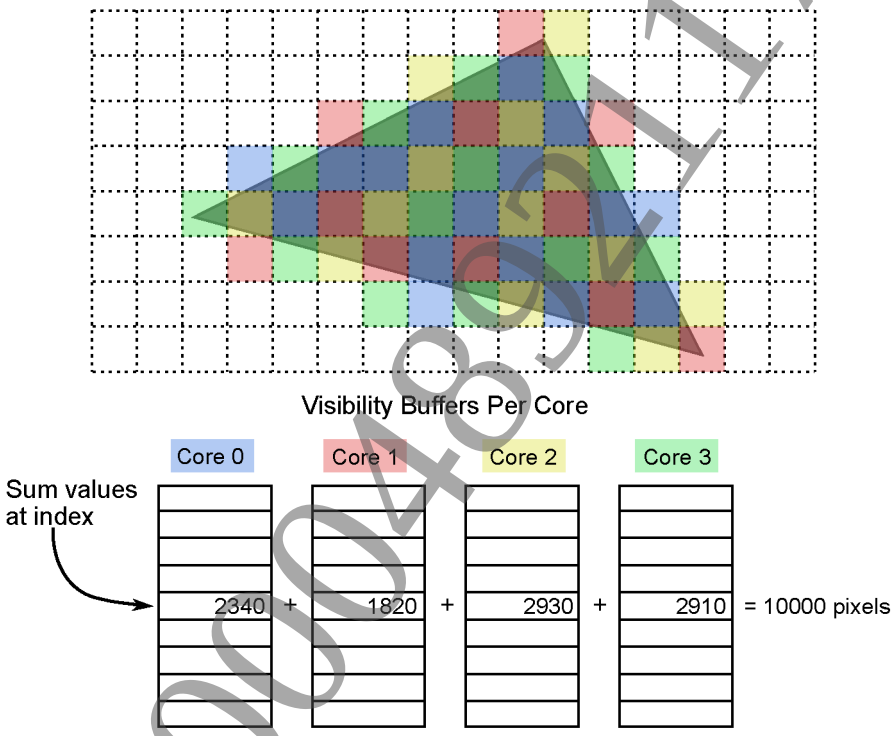
- `SCE_GXM_VISIBILITY_TEST_OP_INCREMENT` increments the counter by 1 for each sample that passes the depth/stencil test. This mode may only be used if the fragment program does not use discard or depth replace.
- `SCE_GXM_VISIBILITY_TEST_OP_SET` sets the counter to 1 for each non-discarded sample that passes the depth/stencil test. This mode may be used with any fragment program.

When fragment processing is complete, all 4 buffers must be read to compute correct visibility results.

> **Note:** Tiles are assigned to GPU cores based on demand during fragment processing, so even if the same scene is rendered the counters values per core could vary frame to frame. However, the sum of the counters from each core should remain constant.

Figure 20 shows visibility counters being updated for a triangle that covers exactly 10000 pixels, where cores are allocated to tiles non-deterministically, but the total is still the correct number of pixels.

Figure 20 Summing Visibility Counters across GPU Cores



The base address and stride between GPU cores for the visibility buffers can be set by calling `sceGxmSetVisibilityBuffer()`. This function may only be called outside of scene.

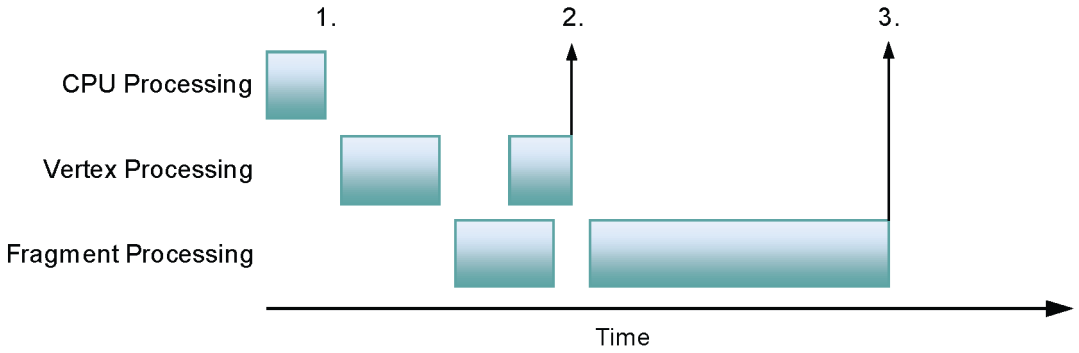
Scene Notifications

There is a fixed size *notification region* provided by libgxm. The base address of this region can be obtained by calling `sceGxmGetNotificationRegion()`, and the number of 32-bit entries is defined by `SCE_GXM_NOTIFICATION_COUNT`, which is currently set to 512 32-bit entries.

Pointers to individual 32-bit entries within this region may be used within `SceGxmNotification` structures, which describe a single 32-bit address/value pair. Each call to `sceGxmEndScene()` accepts one such notification for the vertex pipeline and one for the fragment pipeline, with the 32 bit write performed by the firmware layer when that pipeline completes processing on the GPU.

Document serial number: 000004892117

Figure 21 Program Notifications



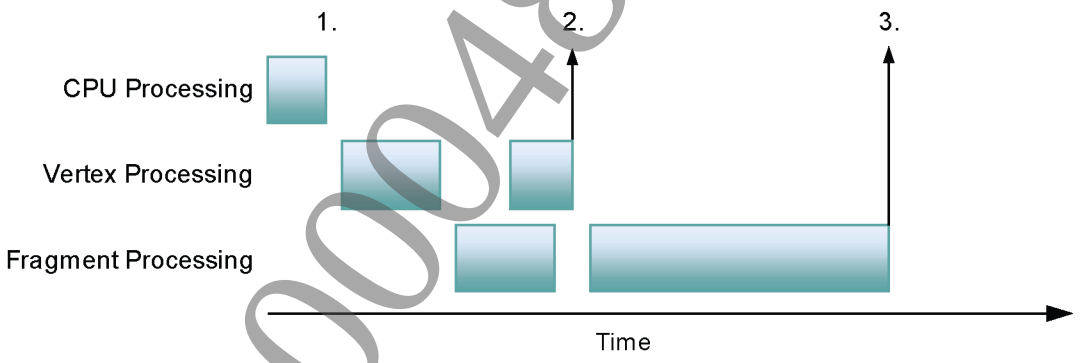
(1) The CPU builds the scene and calls `sceGxmEndScene()`. The job is created and submitted to the GPU firmware layer for processing.

(2) The vertex pipeline finishes processing all vertices and writes its notification to memory. Any data used by the vertex pipeline (such as vertex/index buffers or vertex texture data) can now be safely overwritten by the CPU.

(3) The fragment pipeline finishes processing all tiles and writes its notification to memory. Any data used by the fragment pipeline (such as fragment texture data) can now be safely overwritten by the CPU.

Note that notifications behave the same way if either the libgxm context has split the scene into multiple jobs due to a CPU-side out-of-memory condition or the GPU firmware performs a partial render due to the parameter buffer running out of space during vertex processing. Figure 22 shows an example of the pipeline notifications for the partial render case.

Figure 22 Program Notifications During a Partial Render



As can be seen from Figure 22, a notification still occurs when processing is finished, but there may be significantly increased latency.

To ensure that all scenes submitted to the GPU have finished on both the vertex and fragment processing pipelines, the function `sceGxmFinish()` can be called.

Scene Dependencies

Scenes are always processed in order by the GPU firmware layer, but there is some natural overlap between vertex and fragment processing.

In some situations it is necessary to enforce a dependency between the fragment processing of a scene with the vertex processing of a future scene. You can have such a dependency for render-to-vertex texture effects.

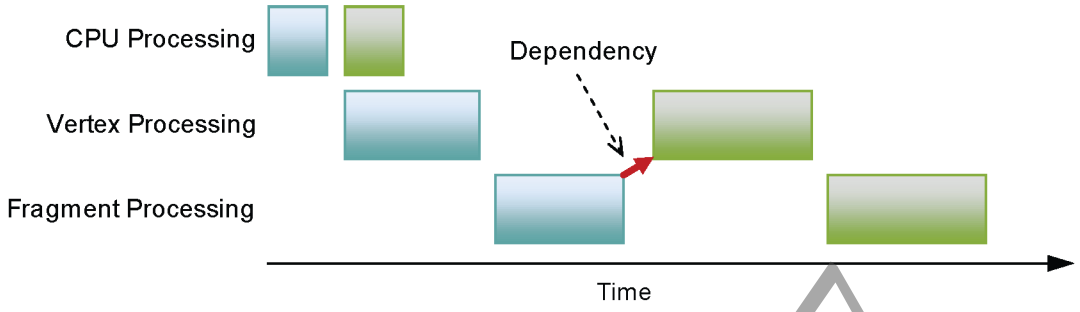
The dependency can be set up by flagging the first scene as `SCE_GXM_SCENE_FRAGMENT_SET_DEPENDENCY` in its scene description, and

Document serial number: 000004892117

000004892117

`SCE_GXM_SCENE_VERTEX_WAIT_FOR_DEPENDENCY` in the second scene. Figure 23 shows the timeline for processing the dependency.

Figure 23 Dependencies Between Vertex and Fragment Processing



The scene that waits does not have to be immediately consecutive to the scene that sets the dependency: it is valid to insert other scenes between them to ensure that the GPU always has a mixture of vertex and fragment processing to work on.

Scenes can also be synchronized with transfer operations, as described in Chapter 7, Transfers.

Additionally, scenes that render to a surface that is also used for display need to synchronize their rendering with flip operations. This can be achieved by pointing to a sync object with the `fragmentSyncObject` parameter of `sceGxmBeginScene()`, and is described in Chapter 9, Synchronization with Display.

Threading

A libgxm context cannot be used simultaneously by multiple threads. To use libgxm from multiple threads, a deferred context can be created for each thread, with each deferred context used to create one or more command lists. Only the immediate context may create the final scene that will execute the command lists on the GPU.

Deferred contexts allow for coarse grain parallelization of rendering by preparing draw calls as command lists, which can spread CPU overheads effectively over multiple CPU cores. However, it is also possible to bypass most of the CPU overheads involved in preparing data for rendering by using precomputation. Precomputation is supported by both immediate and deferred contexts. See Precomputation.

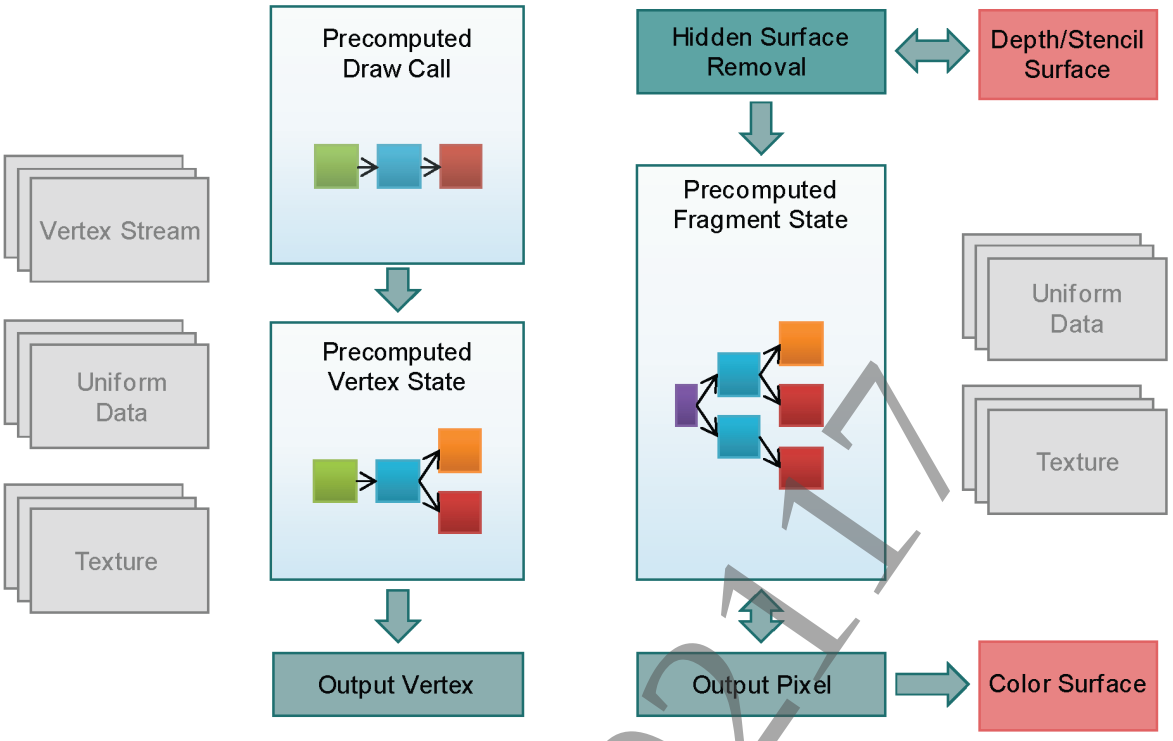
Precomputation

When using precomputation, parts of the rendering API pipeline can be bypassed in favor of precomputed data structures in memory.

Document serial number: 000004892117

000004892117

Figure 24 Rendering Pipeline with Precomputed Data Structures



Usage Model

In general, using precomputation trades memory for performance. Using precomputed objects also requires the caller to synchronize any patching operations with GPU usage.

The finest level of synchronization supported by the GPU firmware layer is via the *Notification* mechanism currently exposed. Patching at a finer level than scene granularity is not currently possible, so developers are advised to use simple scene-based or frame-based double or triple buffering of precomputed structures.

As shown in Figure 24, using precomputed objects allows you to bypass setting many types of objects on the libgxm context. Instead, this state is patched on the precomputed object itself. Because the precomputed object has no dependencies, it may be *patched* at any time it is not being used by the GPU. This patching process can be spread over many independent threads to patch many objects in parallel. In contrast, using the API without precomputation requires that rendering must be performed from a single thread.

Precomputed objects must be created for a particular vertex program (in the case of precomputed draw calls or vertex state) or fragment program (in the case of precomputed fragment state). While you can omit libgxm context calls to set vertex streams, textures and uniform buffers, it is still necessary that you set vertex and fragment programs and any other render state such as depth modes.

Precomputed Draw Call

Precomputed draw calls embed the following information, which may be patched when the object is not currently being used for rendering:

- All the parameters that would otherwise be used with `sceGxmDraw()` or `sceGxmDrawInstanced()`.
- Vertex streams that would usually be set through `sceGxmSetVertexStream()`.

See `SceGxmPrecomputedDraw` and `sceGxmDrawPrecomputed()` in the *libgxm Reference* for more details.

Document serial number: 000004892117

000004892117

Precomputed Vertex State

Precomputed vertex state embeds the following information, which may be patched when the object is not currently being used for rendering:

- Uniform buffers that would usually be set through `sceGxmSetVertexUniformBuffer()`.
- Textures that would usually be set through `sceGxmSetVertexTexture()`.
- The default uniform buffer that would normally be reserved through `sceGxmReserveVertexDefaultUniformBuffer()` or set through `sceGxmSetVertexDefaultUniformBuffer()`.

See `SceGxmPrecomputedVertexState` and `sceGxmSetPrecomputedVertexState()` in the *libgxm Reference* for more details.

Precomputed Fragment State

Precomputed fragment state embeds the following information, which may be patched when the object is not currently being used for rendering:

- Uniform buffers that would usually be set through `sceGxmSetFragmentUniformBuffer()`.
- Textures that would usually be set through `sceGxmSetFragmentTexture()`.
- The default uniform buffer that would normally be reserved through `sceGxmReserveFragmentDefaultUniformBuffer()` or set through `sceGxmSetFragmentDefaultUniformBuffer()`.

See `SceGxmPrecomputedFragmentState` and `sceGxmSetPrecomputedFragmentState()` in the *libgxm Reference* for more details.

Writable Uniform Buffers

Writable uniform buffers allow shader code to write to memory while vertices or pixels are processed during normal rendering.

The GPU contains multiple cores each with their own caches, so before the GPU reads data written to the uniform buffer (for example as a vertex buffer or texture), it is important to ensure that the GPU has finished these memory writes and appropriate caches have been invalidated.

Writing and Reading in Different Jobs

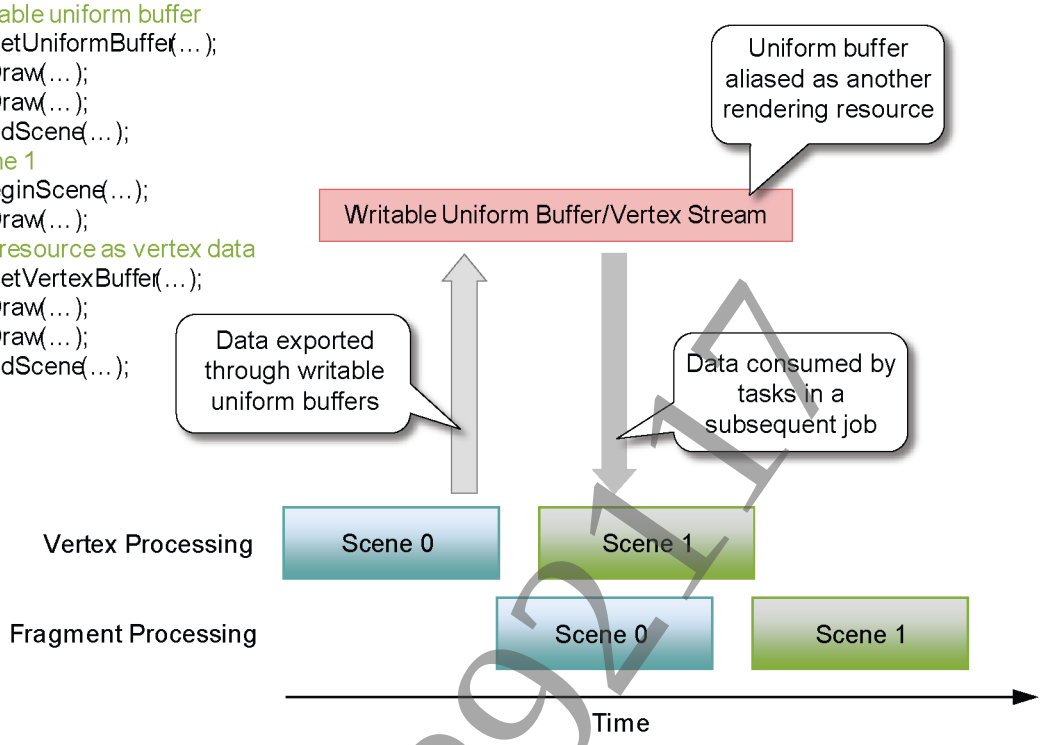
It is possible to alias the memory allocated for a writable uniform buffer as another type of rendering resource. If the data exported from a draw call using writable uniform buffers is used in a different job (this can be a separate scene or through the use of the `sceGxmMidSceneFlush()` function) then no additional synchronization is required to safely alias writable uniform buffers as other types of rendering resources.

000004892117

Document serial number: 000004892117

Figure 25 Using Resources Created in a Previous Job

```
//draw scene 0
sceGxmBeginScene(…);
  sceGxmDraw(…);
  //set writeable uniform buffer
  sceGxmSetUniformBuffer(…);
  sceGxmDraw(…);
  sceGxmDraw(…);
sceGxmEndScene(…);
//draw scene 1
sceGxmBeginScene(…);
  sceGxmDraw(…);
  // use the resource as vertex data
  sceGxmSetVertexBuffer(…);
  sceGxmDraw(…);
  sceGxmDraw(…);
sceGxmEndScene(…);
```



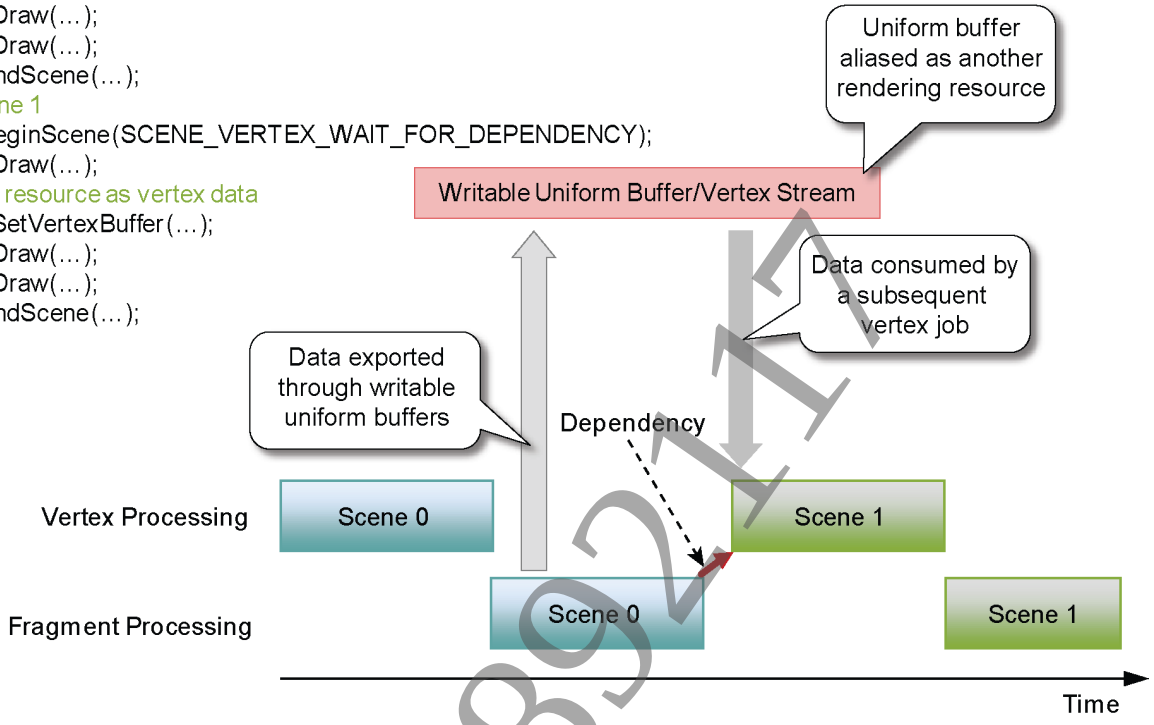
However, if data is exported from a fragment job and consumed in a subsequent vertex job, then scene dependencies must be set up accordingly to ensure that the vertex job does not overlap the fragment job. See Scene Dependencies for more details.

000004892117

Document serial number: 000004892117

Figure 26 Using Vertex Resources Created in a Fragment Job

```
//draw scene 0
sceGxmBeginScene(SCENE_FRAGMENT_SET_DEPENDENCY);
  sceGxmDraw(…);
  //set writeable uniform buffer
  sceGxmSetUniformBuffer(…);
  sceGxmDraw(…);
  sceGxmDraw(…);
sceGxmEndScene(…);
//draw scene 1
sceGxmBeginScene(SCENE_VERTEX_WAIT_FOR_DEPENDENCY);
  sceGxmDraw(…);
  // use the resource as vertex data
  sceGxmSetVertexBuffer(…);
  sceGxmDraw(…);
  sceGxmDraw(…);
sceGxmEndScene(…);
```



000004892117

Document serial number: 000004892117

7 Transfers

Basic Procedure

The following sections describe the steps for using transfers.

Initialization

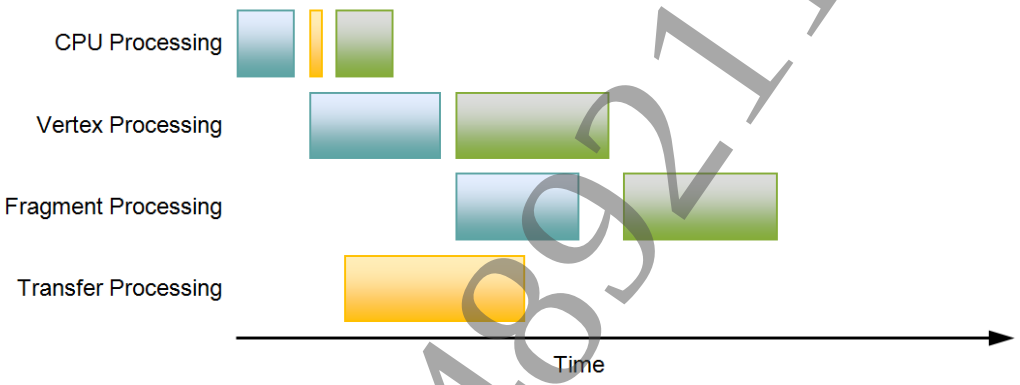
Transfers can be used after the library has been initialized by calling `sceGxmInitialize()`.

Memory for transfer operations must be mapped for use by the GPU before operations are submitted.

Submitting Transfer Operations

Transfer operations are performed in the order in which they are submitted, asynchronously to the CPU, GPU vertex processing, and GPU fragment processing by default.

Figure 27 Job and Transfer Overlap



The transfer operations types shown in Table 6 are supported:

Table 6 Transfer Operation Types

Transfer API Call	Description
`sceGxmTransferCopy()`	Copies memory from one location to another. This operation supports: – Conversion between RGB formats (e.g., `SCE_GXM_TRANSFER_FORMAT_U8U8U8U8_ABGR` to `SCE_GXM_TRANSFER_FORMAT_U1U5U5U5_ABGR`). – Conversion between YUV formats (e.g., `SCE_GXM_TRANSFER_FORMAT_YUYV422` to `SCE_GXM_TRANSFER_FORMAT_UYVY422`). – Conversion between tiled, swizzled and linear memory layouts. – Color keying of RGB/YUV data.
`sceGxmTransferDownscale()`	A fixed 50% downscale operation in both the x and y dimensions using a box filter. Only RGB data in a linear memory layout is supported. Conversion between RGB formats is supported during the downscale operation.
`sceGxmTransferFill()`	Fills RGB/YUV data with a specific value.

All transfer commands support sub-rectangles to only read and write part of the data.

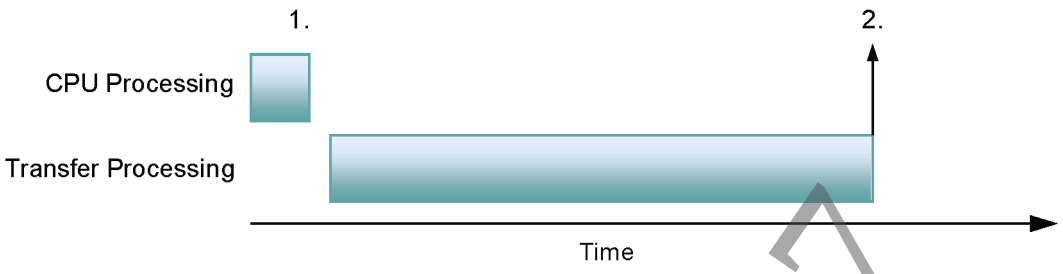
Document serial number: 000004892117

000004892117

Notifications

All transfer commands support an optional notification to indicate that the transfer has completed. This functions in the same way as scene notifications, with the GPU firmware writing a specific value to a location in the *notification region* once the transfer is complete.

Figure 28 Transfer Notifications



The steps are numbered in Figure 28 as follows:

(1) The CPU creates and submits a transfer operation by calling one of the transfer API functions. This is submitted to the GPU firmware for processing.

(2) The GPU completes the transfer operation and writes the notification to memory. Memory used by the transfer operation may now be safely overwritten by the CPU.

To ensure that all submitted transfer operations have finished, the function `sceGxmTransferFinish()` can be called.

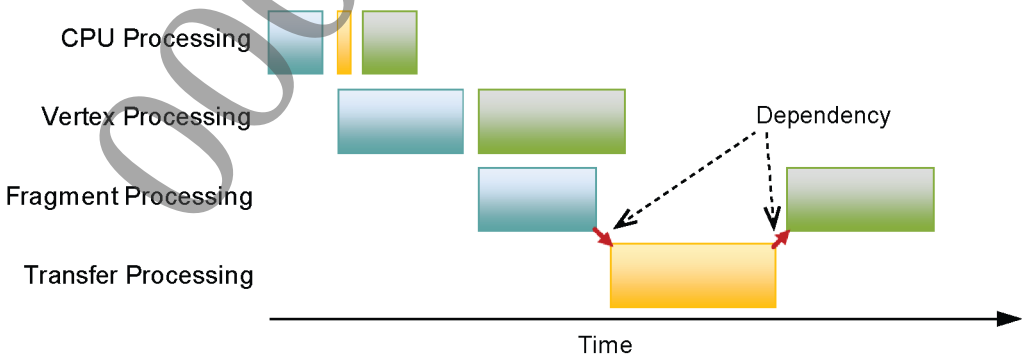
Dependencies

By default, transfers are processed by the GPU asynchronously to vertex and fragment processing.

To support scenarios where the output of a transfer operation is required as input data for a scene, or the output data of a scene is required as input for a transfer operation, synchronization flags can be specified to all transfer commands and `sceGxmBeginScene()`.

To synchronize fragment processing with transfer operations, the scenes and transfers should be submitted in the order in which they should be processed by the GPU, with scenes specifying `SCE_GXM_SCENE_FRAGMENT_TRANSFER_SYNC` in their flags and transfers specifying `SCE_GXM_TRANSFER_FRAGMENT_SYNC` in their flags. An example for this case of generating a downscaled version of a render-to-texture effect is shown in Figure 29.

Figure 29 Dependencies Between Transfers and Fragment Processing



In this example both scenes would set `SCE_GXM_SCENE_FRAGMENT_TRANSFER_SYNC` and the transfer operation would set `SCE_GXM_TRANSFER_FRAGMENT_SYNC`.

Synchronization with vertex processing is also possible via the use of the `SCE_GXM_SCENE_VERTEX_TRANSFER_SYNC` and `SCE_GXM_TRANSFER_VERTEX_SYNC` flags with the

Document serial number: 000004892117

scene and transfer operations respectively. These flags can be specified at the same time as the fragment synchronization flags.

Transfers to a surface that is also used for display need to synchronize with display flip operations. This can be achieved by pointing to a sync object with the `syncObject` parameter to the transfer functions, and is described in Chapter 9, Synchronization with Display.

000004892117

Document serial number: 000004892117

8 Threading

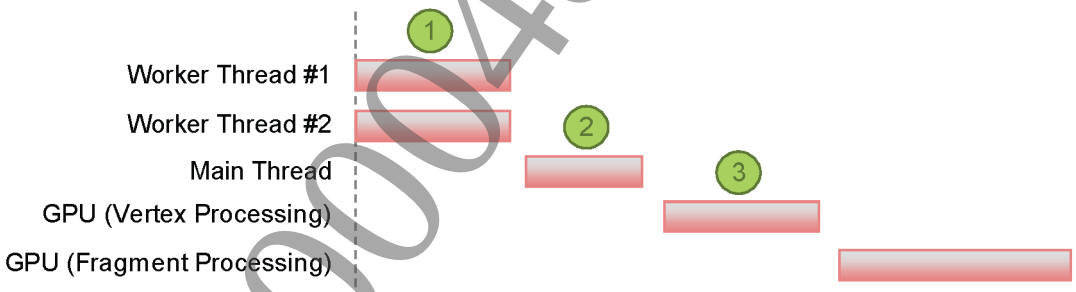
It is important to understand that libgxm structures and the functions that operate on them should typically not be viewed as thread safe unless the structures being operated on are exclusively used by the thread in question. In cases where these structures are being used within another thread, it is the responsibility of the application to use system synchronization primitives or other arbitration logic to ensure that other threads are not accessing those structures simultaneously.

So far this document has dealt mainly with the idea of an *immediate context*, which cannot be used simultaneously by multiple threads. To support multiple threads, an application may create one or more *deferred contexts*. Deferred contexts are independent from each other and the immediate context so they may be used to prepare data in parallel across multiple threads by assigning one deferred context to each thread. A deferred context cannot be used to submit work directly to the GPU. Instead the deferred context allows rendering commands to be recorded into one or more command lists, which can be executed by the immediate context at a later time.

Figure 30 Deferred Contexts Create Command Lists

As each deferred context is a separate construct managing separate regions of memory, it is possible to initialize and build command lists from multiple deferred contexts in parallel.

Figure 31 Command Lists can be Recorded in Parallel with Multiple Deferred Contexts



In Figure 31:

(1) The application's two worker threads are constructing command lists in parallel as each is rendering with a unique deferred context.

(2) When the command lists are constructed, the immediate context (owned by the main thread) can then execute these command lists in addition to performing any rendering of its own.

(3) When the scene is finished, the GPU begins rendering, starting with vertex processing as normal.

Basic Procedure

The basic procedure for rendering with a deferred context is as follows:

Creating a Deferred Context

Call `sceGxmCreateDeferredContext()` to initialize a deferred context. Specify as arguments:

- The memory region for the deferred context object itself.

Document serial number: 000004892117

- Callback functions for each buffer type required by the deferred context.

After a deferred context is initialized it can be used to generate multiple command lists.

Beginning a Command List

Call `sceGxmBeginCommandList()` to start collecting draw calls for a command list.

Performing Rendering Commands

Rendering can now be performed with the deferred context with the results being collected in the command list. Almost all rendering functions can be used with a deferred context as normal. For specific details about which APIs cannot be used, refer to the *libgxm Reference*.

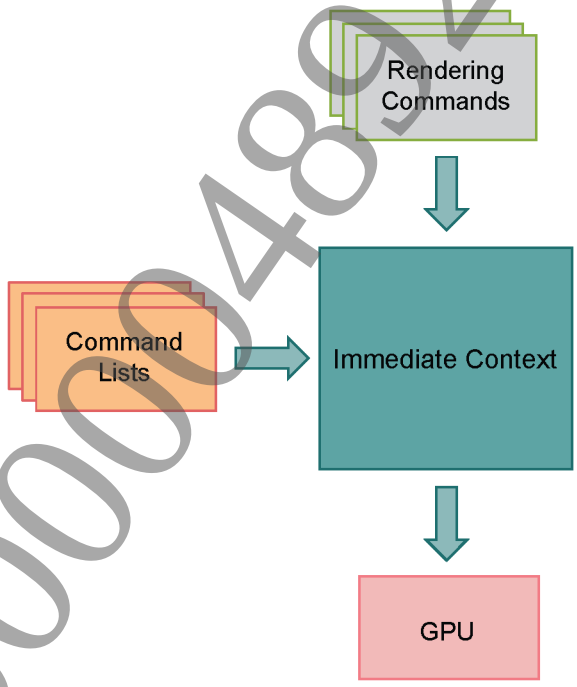
Ending a Command List

Call `sceGxmEndCommandList()` to finish collecting draw calls and finalize the creation of the command list. The command list is now ready to be executed by the immediate context.

Executing a Command List

To submit these rendering commands to the GPU, call `sceGxmExecuteCommandList()` on the immediate context specifying the command list that was generated by the deferred context. This function must be called between calls to `sceGxmBeginScene()` and `sceGxmEndScene()`.

Figure 32 Immediate Context Executes Command Lists



Command Lists

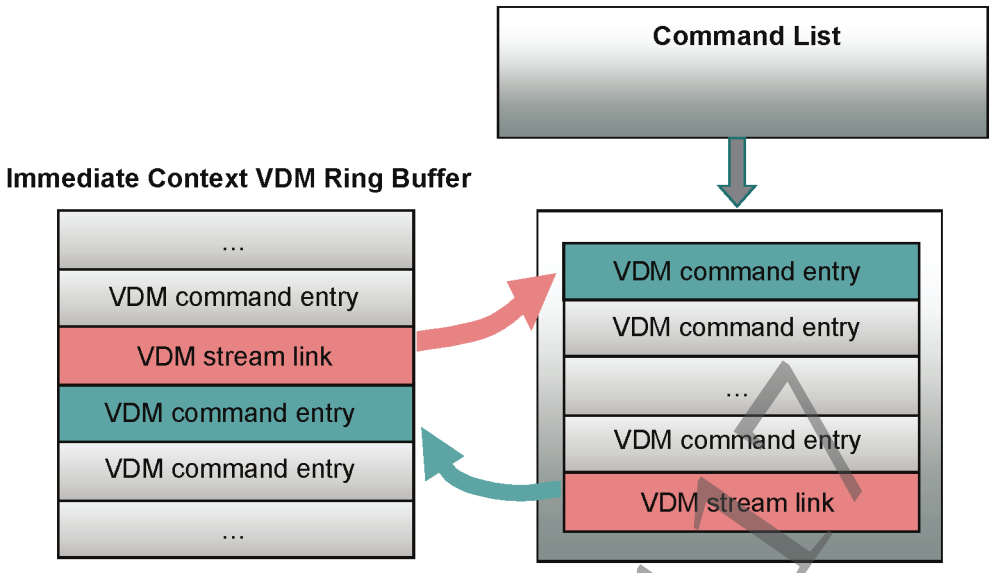
A `SceGxmCommandList` is an opaque data structure that encapsulates everything required for the immediate context to execute rendering commands that have been collected by a deferred context.

The structure contains (among other things) the start address of the memory region used for the VDM commands that form the command list. When the command list is executed from the immediate context, these entries are linked into the VDM ring buffer, as shown in Figure 33.

000004892117

Document serial number: 000004892117

Figure 33 Command Lists are Linked into the Immediate Context's VDM Ring Buffer



The last VDM command of the command list is patched to jump back to the immediate context. As such, it is important to ensure that the GPU has finished vertex processing of any previous scenes that use this command list. See Memory Management for this and other aspects of managing the memory for a command list.

Each libgxm context maintains an independent model of the full GPU state used by a draw call, which includes hardware state, such as culling and primitive modes and depth/stencil state, but also software state, such as the currently bound textures and uniform buffers. As a result, the VDM commands within a command list will always reset the full state of the GPU as the first VDM command, and the immediate context will add a VDM command to set the full state of the GPU back to the immediate context state after the command list returns.

Memory Management

Deferred contexts differ from immediate contexts in terms of the level of memory management they provide. Instead of using a ring buffering scheme to manage the memory required for GPU data structures constructed during the course of rendering, a deferred context uses a more straightforward linear allocation strategy. The application is responsible for both providing and managing the lifetimes of the memory regions used by a deferred context. libgxm provides several different ways of doing this, including:

- Providing Memory Regions During Initialization
- Setting and Getting Memory Regions Manually
- Callback Functions

The buffers that need to be managed by the application during the lifetime of the deferred context are described in Table 7.

Table 7 Deferred Context Buffers

Buffer	Description
VDM Buffer	Used for VDM commands (the front-end of the GPU).
Vertex Buffer	Generic vertex data linear buffer used for generated PDS programs, internal buffers, and vertex default uniform buffer reservations.
Fragment Buffer	Generic fragment data linear buffer used for generated PDS programs, internal buffers, and fragment default uniform buffer reservations.

Document serial number: 000004892117

000004892117

When a memory region is provided to the deferred context, the memory will be consumed linearly from beginning to end as draw calls are added to a command list. When no more memory is available for a particular buffer type, a callback function corresponding to the buffer type will be called, allowing additional memory to be provided to the deferred context to enable it to continue rendering. This allows virtually any type of allocation scheme to be implemented with deferred contexts. If no additional memory is provided via the callback mechanism, then subsequent rendering commands that cause internal allocations (such as draw calls) will fail with an out-of-memory error. They will not be present in the command list. See Out of Memory for further details.

Providing Memory Regions During Initialization

The simplest way of managing memory is to use single buffering and provide memory regions for the VDM, vertex, and fragment buffers at deferred context creation time. This is achieved by passing in memory region base pointers and sizes through the `SceGxmDeferredContextParams` structure. See the *libgxm Reference* for details.

Setting and Getting Memory Regions Manually

An alternative method of managing a deferred context's buffers is by using the collection of functions that allow the current state of a deferred context's buffers to be either set or queried. These functions are described in Table 8.

Table 8 Functions to Set or Query the State of Buffers in a Deferred Context

Function Name	Description
`sceGxmGetDeferredContextVdmBuffer()`	Get the address of the current position in the VDM buffer.
`sceGxmSetDeferredContextVdmBuffer()`	Set a new memory region for the VDM buffer.
`sceGxmGetDeferredContextVertexBuffer()`	Get the address of the current position in the vertex buffer.
`sceGxmSetDeferredContextVertexBuffer()`	Set a new memory region for the vertex buffer.
`sceGxmGetDeferredContextFragmentBuffer()`	Get the address of the current position in the fragment buffer.
`sceGxmSetDeferredContextFragmentBuffer()`	Set a new memory region for the fragment buffer.

These functions may only be called when a command list is not in progress on the deferred context.

Callback Functions

When a deferred context has exhausted all the available memory in one of its buffers (VDM, vertex, or fragment) it will call the corresponding callback function provided to the deferred context during its creation. The callback function can supply a new memory region, which in turn allows rendering to continue. A separate callback function can be specified per buffer type, allowing more sophisticated allocation strategies, such as double or ring buffering, to be implemented.

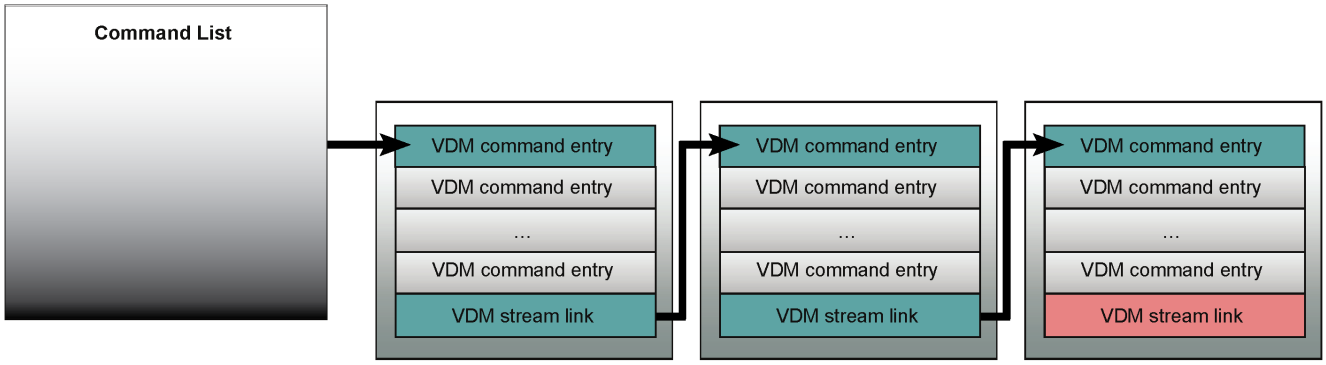
Deferred contexts do not support re-entrancy within their callback functions. It is not valid to call functions on the deferred context from within one of its callback functions; doing so will result in undefined behaviour. As a result, callback functions should simply acquire memory (or fail to acquire memory) from the game application heaps and return immediately, allowing the draw call or reservation currently in progress on the deferred context to continue.

In the case of the VDM stream, additional stream links are automatically written by the deferred context to chain together multiple memory regions to form a single command. Conceptually this is similar to a linked list data structure.

Document serial number: 000004892117

000004892117

Figure 34 Stream Links are Automatically Written by the Deferred Context

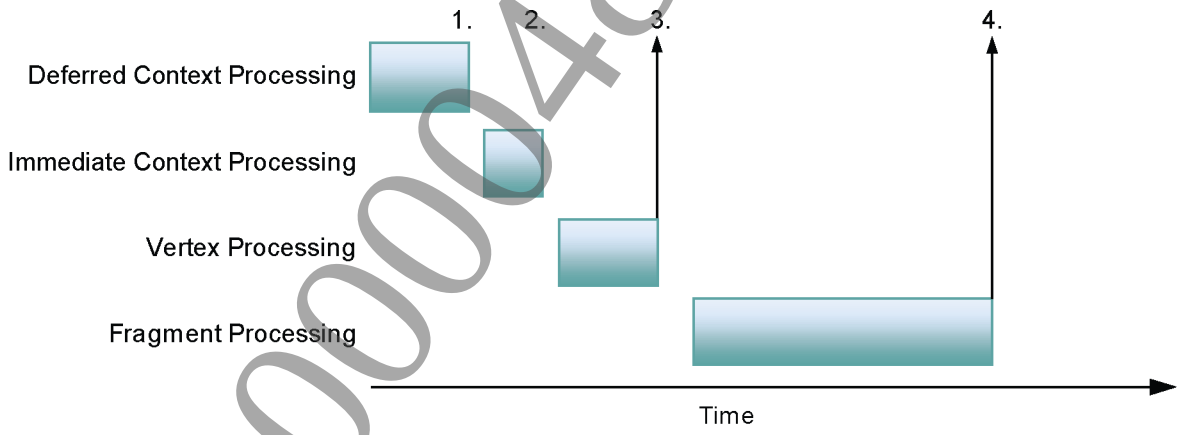


In Figure 34, the green VDM stream link boxes are the stream links created by the deferred context after a new memory region has been supplied to it via the callback mechanism. However, in the final VDM memory region (the rightmost region on the diagram), the red VDM stream link is only reserved and is patched during the call to `sceGxmExecuteCommandList()` so that the VDM returns back to the immediate context. See Figure 33.

Memory Lifetimes

Each of the buffers used by a deferred context has a distinct lifetime associated with it. Memory allocated for the deferred context should not be overwritten (or freed) until the GPU has completed all processing that requires access to this memory. Both the VDM and vertex buffers must persist until the vertex processing for the scene that executes the command list is completed. The fragment buffer must persist until fragment processing for the scene that executes the command list is completed. libgxm provides notifications that can be used to ensure that the GPU has completed all processing. See Scene Notifications.

Figure 35 Deferred Context Buffer Lifetimes



The steps are numbered in Figure 35 as follows:

(1) The CPU creates a command list by calling the rendering API functions on a deferred context.

(2) The CPU submits the command list (and any other rendering performed on the immediate context) to the GPU firmware for processing. The `SceGxmCommandList` structure does not need to persist after the call to `sceGxmExecuteCommandList()`.

(3) The GPU completes vertex processing and writes the notification to memory. Memory used by the command list's VDM or vertex buffers may now be safely overwritten or freed by the CPU.

(4) The GPU completes fragment processing and writes the notification to memory. Memory used by the command list's fragment buffer may now be safely overwritten or freed by the CPU.

Document serial number: 000004892117

000004892117

Out of Memory

An important difference between an immediate and a deferred context is that deferred context rendering commands may fail due to out-of-memory conditions. While this will never happen if memory is always provided to a deferred context when the callback functions request it, it is important to understand the implications of running out of memory if the application's design allows out-of-memory conditions to arise.

Memory is requested through the callbacks every time it is needed, even if previous requests failed. If the callback function fails to make more memory available, the remaining memory for that buffer is still kept and could still be used if smaller allocations from subsequent draw calls can use it.

If `sceGxmBeginCommandList()` succeeds, then the command list can always be ended through `sceGxmEndCommandList()` and executed from an immediate context. However, if during the course of creating the command list the application encounters an out-of-memory condition then one or more draw calls within the command list may fail and will not be included within the command list.

000004892117

Document serial number: 000004892117

9 Synchronization with Display

libgxm is separated from display behavior: it does not provide a full swap chain implementation. Instead, it provides a generic *display queue* that synchronizes a custom callback function with the fragment processing pipeline of the GPU.

This callback function is called from an internal thread as soon as the relevant fragment or transfer processing operations complete, independent of any threads the application itself is running. This callback function is triggered from an internal GPU interrupt, which ensures that the callback is run with minimal latency, while not running inside the interrupt handler.

The implementation of this callback function should perform the flip operation and associated VSYNC handling by calling functions from the display library.

Basic Procedure

The following sections describe the steps for using the display queue.

Creating the Display Queue

The display queue is created during `sceGxmInitialize()`. The parameters specific to the display queue are:

- Callback function to use for the flip operation.
- Maximum number of display flips to allow in the queue.
- Size of the data that needs to be serialized with each display queue entry.

The size parameter allows for arbitrary data to easily be passed from the application to the display queue callback (from its separate thread).

Creating the Sync Objects

Synchronization objects are required to synchronize access to a particular resource between the GPU and the flip operation. Each resource should have exactly one sync object associated with it. For example, an application that intends to use triple buffering should create three sync objects and always associate the same sync object with each buffer.

Create sync objects when the application starts by calling `sceGxmSyncObjectCreate()`.

Using the Sync Objects and Display Queue

Any buffer that is both rendered to and displayed must be used with its associated sync object to ensure these operations are correctly ordered. This is necessary in the following places:

- As the *`fragmentSyncObject`* parameter to `sceGxmBeginScene()`.
- As the *`syncObject`* parameter to transfer API functions `sceGxmTransferCopy()`, `sceGxmTransferDownscale()`, and `sceGxmTransferFill()`.
- When adding a swap operation using `sceGxmDisplayQueueAddEntry()`.

The call to `sceGxmDisplayQueueAddEntry()` queues up a display operation to flip between two buffers, and requires the sync object for each buffer as an argument. This function also copies data to pass to the callback function on its callback thread during the flip operation. This data is usually just the buffer address to flip to, but could also contain the flip mode or width and height if using dynamic scaling.

Within the callback, the caller should do the following.

- Trigger the flip as soon as possible.
- Block until the flip is completed and the old buffer is no longer displayed.

Document serial number: 000004892117

000004892117

The blocking behavior is important, because there may be pending fragment pipeline or transfer operations that wish to write to the buffer that is currently being displayed, especially if the application is using double buffering. After the callback returns, the GPU will be allowed to start writing to this buffer, so it is crucial that the flip operation has finished before returning from the callback.

Destroying the Sync Objects

When the application has finished with the sync objects and they are no longer being used with the GPU or CPU, they should be destroyed using `sceGxmSyncObjectDestroy()`.

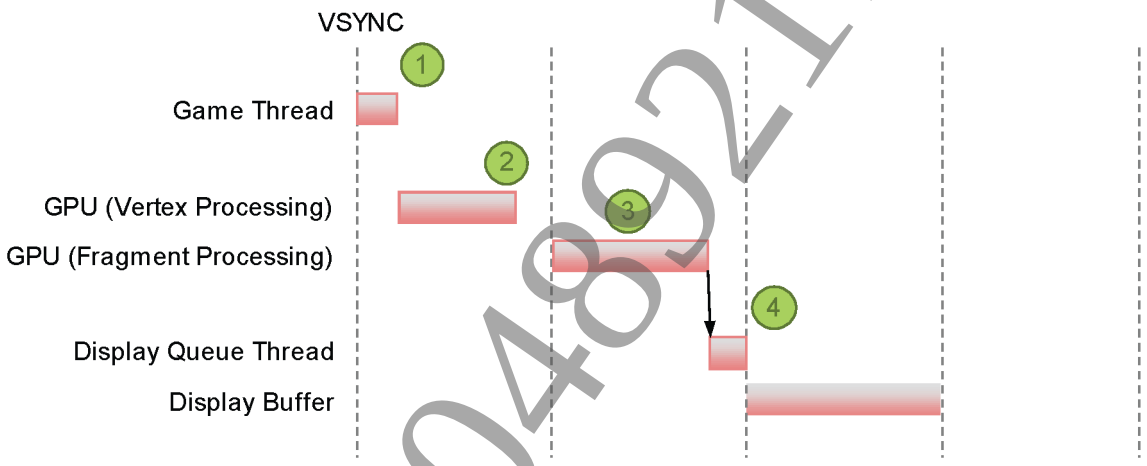
Destroying the Display Queue

The display queue is destroyed as part of `sceGxmTerminate()`. To simply wait for all pending display queue entries to complete without destroying the queue itself, call `sceGxmDisplayQueueFinish()`.

Example Timeline

Figure 36 shows the timeline for a single frame, using a display callback that flips on the next VSYNC.

Figure 36 Example Display Timeline For A Single Frame



In Figure 36:

(1) The game thread begins, draws and ends a scene.

(2) The vertex processing for this scene occurs.

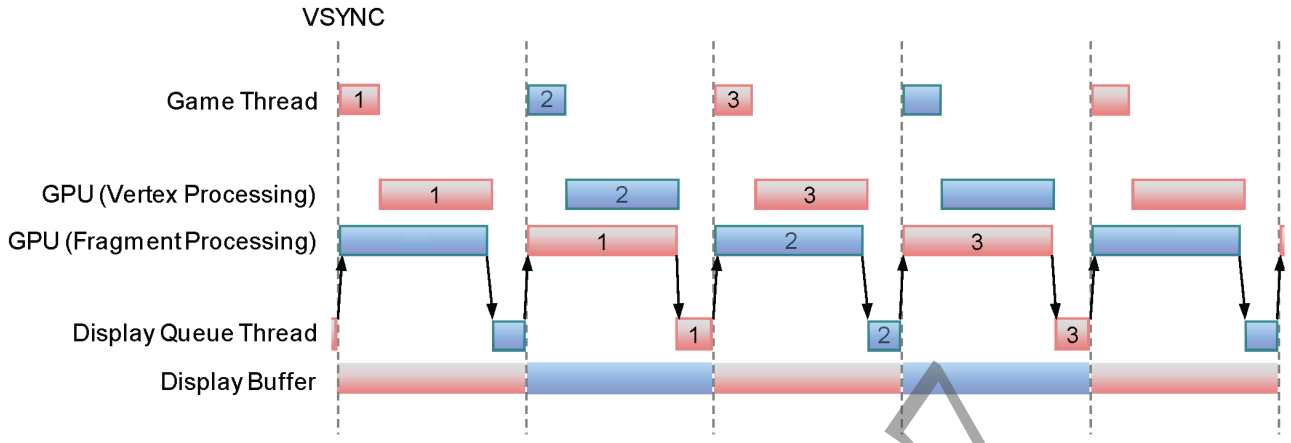
(3) After the fragment processing buffer can be written to, fragment processing starts. After fragment processing is complete, the display queue callback is triggered (the dependency is shown as an arrow in the figure).

(4) The display queue callback (in this example) sets the buffer to display on the next VSYNC, and then blocks until this operation is complete.

In the steady state in a double buffered application, this configuration allows for one frame of overlap, as shown below in Figure 37. Note how only the fragment processing is synchronized with the flip: fragment processing cannot start until its buffer is flipped away from, and it cannot trigger the flip until processing is complete. Vertex processing is unaffected by flip operations.

Document serial number: 000004892117

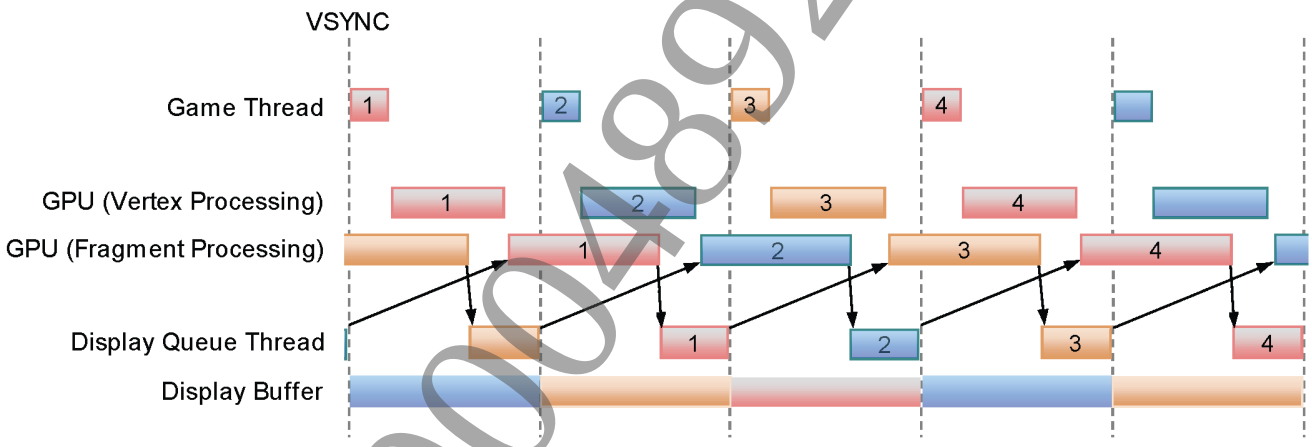
Figure 37 Example Double Buffering Steady State



In Figure 37, the blocks of processing are labeled with their frame number (frame 1, 2, or 3). The colors of each block correspond to processing or display for a particular buffer in the double-buffered setup.

In a triple buffered application, the additional buffer allows for more overlap between fragment processing and flip operations, because it can be rendered to before the flip operation affecting the other two buffers is completed. Figure 38 shows an example time line for the triple buffered steady state.

Figure 38 Example Triple Buffering Steady State



Note that the display queue callback is generic code supplied by the caller. Although the examples shown here are for VSYNC-based flipping at the display refresh rate, it is possible for the user to implement different schemes, such as always displaying a buffer for two VSYNCs. Example callbacks for these schemes will be added in a future release of this document.

000004892117

Document serial number: 000004892117

10 Memory Model

The libgxm memory model ensures that when memory is accessible to the GPU, the GPU virtual address is identical to the CPU virtual address. As such, all libgxm functions that expose pointers in GPU data structures will do so using standard CPU virtual addresses.

Basic Procedure

Memory is required for vertex data, index data, uniform buffers, color, and depth/stencil surfaces and textures. The user is expected to allocate and manage this data.

Allocating Memory

All memory is considered as cached from a GPU perspective; that is, loads and stores issued by the GPU use the GPU caches.

Memory can be allocated from either main memory or video memory. When allocating from main memory, the caller can specify whether this memory should be cached from a CPU perspective, that is, whether loads or stores issued by the CPU use the CPU caches.

To allocate memory, call the kernel API `sceKernelAllocMemBlock()`. The five memory block types supported by this function are listed in Table 9:

Table 9 Memory Block Types

Memory Block Type	Description
`SCE_KERNEL_MEMBLOCK_TYPE_USER_RW`	Main memory that the CPU reads and writes cached. The GPU also reads and writes this memory cached. If the GPU misses its caches, the CPU caches are snooped before loading from main memory. In this way, the GPU is coherent with the CPU.
`SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE`	Main memory that the CPU reads and writes uncached. **Note:** the GPU still reads and writes this memory cached.
`SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RW`	Video memory. The CPU always reads and writes video memory uncached. **Note:** the GPU still reads and writes this memory cached.
`SCE_KERNEL_MEMBLOCK_TYPE_USER_MAIN_PHYCONT_RW`	Memory blocks from a physical continuous memory area in a user space that the CPU reads and writes cached.
`SCE_KERNEL_MEMBLOCK_TYPE_USER_MAIN_PHYCONT_NC_RW`	Memory blocks from a physical continuous memory area in a user space that the CPU reads and writes uncached. **Note:** the GPU still reads and writes this memory cached.

The size of memory blocks must always be a whole number of pages. Therefore, for main memory, the size must be a multiple of 4 KB; for video memory, it must be a multiple of 256 KB.

To obtain the base address of the memory block, call `sceKernelGetMemBlockBase()`.

Document serial number: 000004892117

000004892117

Mapping Memory

Before memory can be used for GPU data, you must map it by calling `sceGxmMapMemory()`. This function may only be called with 4 KB-aligned base address and sizes to ensure whole numbers of pages are mapped.

Memory can be mapped as read-only or read-write with the GPU. It is recommended to map memory that is only required for reading by the GPU (such as texture data) as read-only to ensure hardware protection from GPU overwrites. Memory that needs to be written by the GPU, such as writable uniform buffers, color surfaces, or depth/stencil surfaces, must be mapped as read-write.

As of SDK 0940, memory can also be mapped as vertex USSE code and fragment USSE code. To perform this mapping operation, call `sceGxmMapVertexUsseMemory()` or `sceGxmMapFragmentUsseMemory()`. The shader cores cannot access memory using CPU virtual address, so these mapping operations return a 32-bit *USSE offset* for the start of the mapped region.

It is possible to map memory as both vertex USSE code and fragment USSE code. However, the *USSE offset* returned by each mapping operation will be different, so care must be taken to use the correct offset depending on whether the shader code is a vertex program or fragment program.

Using Mapped Memory

After memory has been mapped, any pointer within the mapped region may be used with GPU data structures.

For the case of USSE code, the offset for a given pointer within the mapped region is computed by adding the offset in bytes from the start of the region to the *USSE offset* value returned by the mapping operation.

Unmapping Memory

Unmap memory by calling `sceGxmUnmapMemory()`, `sceGxmUnmapVertexUsseMemory()` or `sceGxmUnmapFragmentUsseMemory()`, depending on how the memory was mapped. If memory was mapped as both vertex and fragment USSE code, it needs to be unmapped from both before the memory is freed. Before unmapping memory, callers should ensure that it is not currently being used by the GPU, otherwise page faults could occur. A simple way to ensure that the GPU is not currently using the memory is to call `sceGxmFinish()` before unmapping.

Freeing Memory

After memory has been unmapped, it should be freed by calling the kernel API `sceKernelFreeMemBlock()`.

000004892117

Document serial number: 000004892117

11 Using the Shader API

libgxm exposes a compact API for interrogating the shader compiler output. This API exposes some basic information about the program itself and an array of parameters that describe how to provide inputs to the program in the rendering API and shader patcher API.

Basic Procedure

The following sections describe the steps for using the shader API.

Loading the Program

It is the responsibility of the game to load the shader compiler output into memory. This output can be interpreted as being of type `SceGxmProgram`. Call `sceGxmProgramCheck()` to test that the data is as expected and that it is from the correct version of the shader compiler.

The program can either be a vertex or fragment program. Call `sceGxmProgramGetType()` to determine the type of a program in memory.

Program Parameters

A program exposes an array of parameters to describe its inputs. The number of parameters can be queried by calling `sceGxmProgramGetParameterCount()` and each parameter can be accessed by index by calling `sceGxmProgramGetParameter()`.

The shader API also exposes the utility function `sceGxmProgramFindParameterByName()` to look up a parameter by name, and `sceGxmProgramFindParameterBySemantic()` to find a parameter by semantic:

```
SceGxmProgramParameter *param
    = sceGxmProgramFindParameterByName(program, "foo");
if (!param) {
    // handle parameter not found case
}
```

Parameters are always assigned to a category, which you can determine by calling `sceGxmProgramParameterGetCategory()` and passing in the parameter pointer. The resource index and type of a parameter has different meaning depending on the category (exposed as `SCE_GXM_PARAMETER_CATEGORY_XXX`, where *XXX* is replaced by one of the category strings in Table 10.

Table 10 SCE_GXM_PARAMETER_CATEGORY Strings

Category	Description
`ATTRIBUTE`	Vertex attributes (for vertex programs only). Note that the type of a vertex attribute parameter is always reported as `4xF32`, since this is the format expected by the shader code after unpacking. The resource index is the register index for the vertex attribute.
`UNIFORM`	A uniform parameter. The resource index of uniform parameters is the 32-bit word offset within a uniform buffer. The container index is the uniform buffer index or `SCE_GXM_PROGRAM_INVALID_INDEX` for uniforms in the default buffer. Uniforms are currently always either `F32` or `F16` format in memory, and each parameter is at most an array of vectors of this scalar type. Note that when arrays are used, each array element always starts on a word-aligned boundary.

Document serial number: 000004892117

©SCEI

Category	Description
SAMPLER	A sampler parameter. The resource index of a sampler is the texture unit to use with the rendering context. The scalar type and component count show the sampler result type expected by the shader code: the user should ensure that a compatible texture format is used at runtime (see the *GPU User's Guide* for more details).
UNIFORM_BUFFER	The parameter representing the entire uniform buffer. The resource index of a uniform buffer is the uniform buffer index to use with the rendering context functions `sceGxmSetVertexUniformBuffer()` or `sceGxmSetFragmentUniformBuffer()`. The uniform buffer parameter is of type `SCE_GXM_PARAMETER_TYPE_AGGREGATE`, and can be queried for the size of the buffer in bytes. Only user declared uniform buffers are represented by a parameter of this type, the default uniform buffer is not represented. The visibility of uniform buffers can be altered by marking uniform buffers with the `enable_buffer_symbols` and `disable_buffer_symbols` pragmas in the shader's Cg source code. See the *Shader Compiler User's Guide* for more details on the available uniform buffer pragmas.

The default uniform buffer is not part of the array of parameters. It is possible to query the size of the default uniform buffer by calling `sceGxmProgramGetDefaultUniformBufferSize()`.

000004892117

Document serial number: 000004892117

12 Using the Shader Patcher API

The shader patcher API creates final programs by combining the output of the shader compiler with a model for state that is close to what is expected by standard rendering APIs. This state generates additional shader code.

Basic Procedure

The following sections describe the steps for using the shader patcher API.

Initialization

Call `sceGxmShaderPatcherCreate()` to create the shader patcher. This function takes as an argument a structure that describes callback functions to use for memory allocations.

The allocations are split into the following categories:

- Host memory is standard CPU-cached memory, such as that returned by `malloc()`. This memory is used for CPU data structures.
- Buffer memory is memory mapped as read-write for the GPU using `sceGxmMapMemory()`. This memory is used for literal data read from memory, and for read-write spilling buffers used by programs that ran out of temporary registers.
- Vertex USSE memory is used for final vertex USSE code.
- Fragment USSE memory is used for final fragment USSE code.

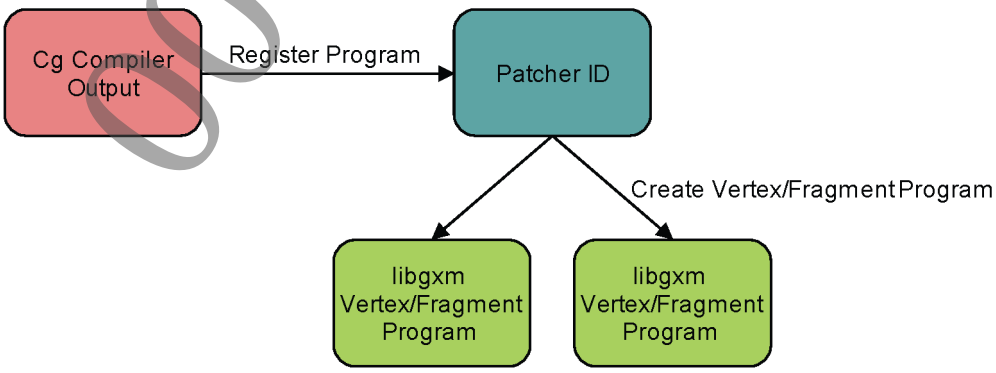
For all allocation types apart from host memory, it is possible to specify a static region of memory instead of a pair of callback functions. If a memory region is provided, a heap will be created by the shader patcher using that memory, and callbacks will be provided internally that use this heap.

For the case of vertex and fragment USSE memory, a single region may be used as a combined heap. In order to use this mode, the base address and size of the regions must match. The caller must ensure that the correct offsets are provided, as the vertex USSE offset for the memory region will be different from the fragment USSE offset for the memory region, even though the base addresses match.

Registering Programs

To use a program with the shader patcher, it must first be registered by calling `sceGxmShaderPatcherRegisterProgram()`. This produces an ID for use with all further shader patcher API calls.

Figure 39 Shader Patcher Registration



This ID is used as the root of a data structure that contains multiple vertex programs or multiple fragment programs. Each time the shader patcher is requested to create a vertex or fragment program for a given ID, this data structure is checked to see if a compatible program has already been created, and if so this

Document serial number: 000004892117

existing program has its reference count increased. Only if the parameters require the creation of a new program will a new program be created.

Creating a Vertex Program

Create a vertex program for a given program ID by calling `sceGxmShaderPatcherCreateVertexProgram()`. Call the function with the following information:

- **Vertex attribute descriptions.** These descriptions are used to generate shader code that unpacks from the memory format to the canonical float representation expected by the compiled shader code. Where the shader uses more components than are provided by the attribute in memory, the standard (0, 0, 0, 1) vector is used to supply the extra components.
  - When offline vertex unpack is used for an attribute, the shader patcher should not unpack from the memory format, since this code has already been generated by the shader compiler. All that is left for the shader patcher to do is generate a template PDS program to transfer the attribute from memory. To specify this, the attribute format `SCE_GXM_ATTRIBUTE_FORMAT_UNTYPED` should be used, along with the number of 32-bit words of data to transfer from memory.
- **Vertex stream descriptions.** These descriptions are used to generate a program template that fetches vertex data from the vertex streams. The rendering API copies and patches this program for each draw call that uses it.

Creating a Fragment Program

To create a fragment program for a given ID, call the `sceGxmShaderPatcherCreateFragmentProgram()` function with the following information:

- **The output register format of the program.** This is the format to use for pixels when they are in on-chip storage.
  - If programmable blending is used in the shader code with the `__nativecolor` extension, or the fragment program has no color output, this parameter is ignored as runtime format conversion and blending are not possible.
  - If programmable blending is not used, the fragment program color output is in `half4` format if `__regformat` is not used, or the format specified by `__regformat` if it is present. The shader patcher is capable of converting `half4` to any other format, or the parameter `SCE_GXM_OUTPUT_REGISTER_FORMAT_DECLARED` can be passed to preserve the format specified in the shader code.
- **The MSAA mode that will be used with this program.** This flag is required to account for differences in execution rate in the shader compiler output. How MSAA affects shader code will be covered in more detail in a future SDK release.
  - If programmable blending is used in the shader code with the `__nativecolor` extension, the program must have been compiled for at least the number of samples specified in the runtime MSAA mode. For example, when specifying the `__msaa` keyword offline (which is an alias for `__msaa4x`), this compiles the program for 4xMSAA, which can be used with both 4xMSAA and 2xMSAA at runtime. Additional shader syntax is not required for MSAA when programmable blending is not used.
- **The blend mode and color mask.**
  - If programmable blending is used in the shader code with the `__nativecolor` extension, or the fragment program has no color output, this parameter is ignored as runtime blending is not possible.
  - If programmable blending is not used and the fragment program has a color output, a NULL blend info generates shader code to overwrite the output register directly without blending or masking.
  - If programmable blending is not used and the fragment program has a color output, a non-NULL blend info generates shader code to implement the desired blend mode and mask operation with the existing output register contents.

Document serial number: 000004892117

000004892117

- Blending is supported only if the output register format has 4 components, in order for the blend equation to be well-defined. Having an active blend mode or non-trivial color mask when the output register format is not `SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4` or `SCE_GXM_OUTPUT_REGISTER_FORMAT_HALF4` will result in an error when trying to create the fragment program.

- **The vertex program to link with.** If provided, the fragment program texture coordinates are remapped to take account of any gaps in the vertex program outputs. The vertex program can be omitted if the caller knows that the vertex program writes to a contiguous range of `TEXCOORD` interpolants starting at `TEXCOORD0`.

To create a special case fragment program that can be used to update the mask bit, use `sceGxmShaderPatcherCreateMaskUpdateFragmentProgram()`. Creating this fragment program does not require a program ID or any other parameters. The reference count for this fragment program should be managed in the same way as fragment programs created using `sceGxmShaderPatcherCreateFragmentProgram()`.

Managing Reference Counts

The vertex and fragment programs created by the shader patcher are reference counted. If a call is made to the shader patcher API with identical arguments to previous calls, the reference count of an existing `SceGxmVertexProgram` or `SceGxmFragmentProgram` program is increased. The reference count can also be manually incremented by calling `sceGxmShaderPatcherAddRefVertexProgram()` or `sceGxmShaderPatcherAddRefFragmentProgram()` to share programs without calling a creation function.

When a program is no longer needed, it should be released using `sceGxmShaderPatcherReleaseVertexProgram()` or `sceGxmShaderPatcherReleaseFragmentProgram()`. You must ensure the program is no longer being used by:

- **The GPU:** Check by using notifications or call `sceGxmFinish()` to ensure the GPU has finished all pending operations.
- **libgxm context:** Check by setting a different vertex/fragment program or setting the program to NULL.

Instancing

In libgxm, instancing is part of the description of the vertex streams for the shader patcher. Each stream may have one of the following index sources:

Table 11 Index Sources

Index Source	Description
`SCE_GXM_INDEX_SOURCE_INDEX_16BIT`	Index into the stream using the index value, where the value is 65535 or less.
`SCE_GXM_INDEX_SOURCE_INDEX_32BIT`	Index into the stream using the index value, which could be larger than 65535, but less than the maximum index value of 16777216.
`SCE_GXM_INDEX_SOURCE_INSTANCE_16BIT`	Index into the stream using the instance number, where the instance number is 65535 or less.
`SCE_GXM_INDEX_SOURCE_INSTANCE_32BIT`	Index into the stream using the instance number, which could be larger than 65535, but less than the maximum index value of 16777216.

When using 32-bit sources, more expensive indexing code has to be generated for the PDS unit of the GPU, so 32-bit sources should only be used if actually required.

Document serial number: 000004892117

000004892117

When drawing geometry using `sceGxmDrawInstanced()`, the `indexWrap` parameter specifies how many indices are rendered per mesh. In the draw call, after the specified number of indices have been drawn, the offset within the index buffer is reset to zero and the instance number is incremented.

For example, an array of 50 cubes has the following stream layout:

- **Stream 0:** Position, normal, texture coordinate per vertex.
- **Stream 1:** Matrix per instance.

To describe the indexing into these streams, the following stream array should be used when creating the vertex program using `sceGxmShaderPatcherCreateVertexProgram()`:

```
SceGxmVertexStream streams[2];
streams[0].stride = sizeof(Vertex);
streams[0].indexSource = SCE_GXM_INDEX_SOURCE_INDEX_16BIT;
streams[1].stride = sizeof(Matrix);
streams[1].indexSource = SCE_GXM_INDEX_SOURCE_INSTANCE_16BIT;
```

Assuming each cube contains 36 indices, the 50 cubes could then be drawn using the following call:

```
sceGxmDrawInstanced(
    ctx,
    SCE_GXM_PRIMITIVE_TRIANGLES,
    SCE_GXM_INDEX_FORMAT_U16,
    pIndicesForASingleCube,
    36 * 50,
    36);
```

Note: Only the indices for a single cube need to be provided, since the index buffer offset is reset to zero each time 36 indices have been rendered.

000004892117

Document serial number: 000004892117

13 Performance Considerations

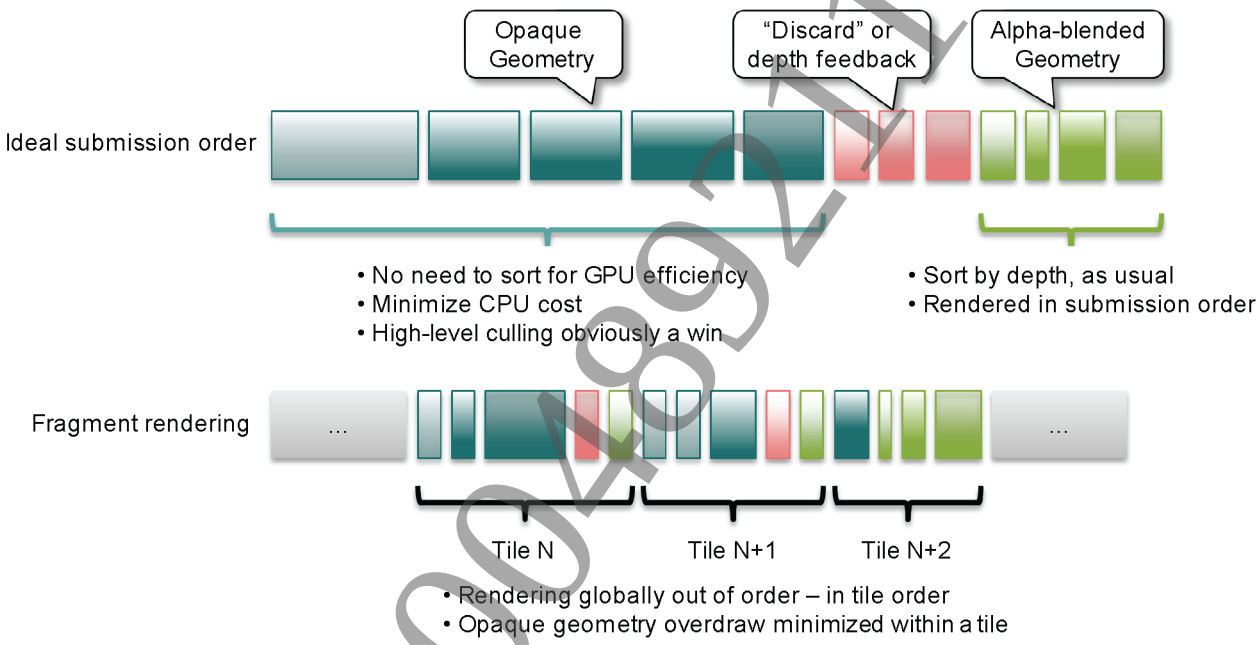
This section includes information about how you can take advantage of the tile-based architecture to improve rendering performance.

This material will be extended with more detail in a future release of this document.

Rendering Order

The ideal submission order for geometry on SGX is to submit all opaque geometry before any geometry that uses discard, depth feedback, or alpha blending. This sequence allows for hidden surface removal to remove as much opaque overdraw as possible. After opaque geometry is submitted, submit geometry that uses fragment program discard or fragment program depth writes, followed by alpha-blended geometry.

Figure 40 Rendering Order



Note that SGX hardware does not sort primitives, it merely looks ahead for opaque geometry to skip pixels that ultimately are not visible. As such, alpha geometry should be sorted by depth as per other geometries because they will be rendered in submission order.

Depth-Only Rendering

For best performance during depth-only rendering, fragment programs should be disabled on the hardware where possible. However, fragment programs must not be disabled when discard or depth replace is used, as this requires shading to produce the discard or depth result.

It is recommended to let libgxm disable the fragment program automatically, which will happen for the following cases:

- The fragment program did not declare a `COLOR` output (this is optional in the `sce_fp_psp2` profile), and there are no side-effects such as discard or depth replace.
- The blend equation has no effect on destination color and there are no side-effects such as discard or depth replace. This check occurs for both programmable blending and runtime blending using the shader patcher.

Document serial number: 000004892117