Graphics Programming Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

This document is meant for graphics programmers who have experience with fixed-function pipeline graphics architecture, such as used by the PSP™ (PlayStation®Portable), and who may not be familiar with recent advances in graphics programming, such as vertex or fragment shaders.

The goal is to quickly bring such developers up to speed by contrasting these architectures, and by either giving concise information or pointers to relevant and readily available documentation.

This document includes the following Hardware and Software information:

- Chapter 1, Hardware: Overview of Graphics Architectures and Chapter 2, Hardware: PlayStation®Vita Graphics Architecture – a comparison of fixed-function pipeline graphics architecture to the programmable pipeline graphics architecture that can be implemented on the SGX processor.
- Chapter 3, Software: Performing Basic Graphics Tasks on PlayStation®Vita and Chapter 4, Software: Sample Walkthrough - Shadow Mapping – describe the libgxm API and other points related to software.

Related Documentation

Refer to the following materials for a detailed reference of PSP™ Graphics Programming:

- *Graphics Engine User's Manual*
- *Graphics Engine Command Reference Manual*
- *libgu Overview*
- *libgu Reference*
- General PSP™ resources, including developer support help: https://psp.scedev.net

Refer to the following materials for PlayStation®Vita Graphics Programming:

- *libgxm Overview*
- *libgxm Reference*
- *GPU User's Guide*
- *Shader Compiler User's Guide*
- *Texture Pipeline User's Guide*
- *Performance Analysis and GPU Debugging*
- General PlayStation®Vita resources, including developer support help: https://psvita.scedev.net

Refer to the following document for PlayStation®Vita Cg Shader Programming:

- *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison Wesley Professional, ISBN 0321194969

Typographic Conventions

The typographical conventions used in this guide are explained in this section.

Hints

A GUI shortcut or other useful tip for gaining maximum use from the software is presented as a 'hint' surrounded by a box. For example:

Hint: This hint provides a shortcut or tip.

Notes

Additional advice or related information is presented as a 'note' surrounded by a box. For example:

> **Note:** This note provides additional information.

Text

- Names of keyboard functions or keys are formatted in a bold serif font. For example, Ctrl, Delete, F9.
- File names, source code and command-line text are formatted in a fixed-width font. For example:

```
const float A = delta * delta;
```

Errata

Any updates or amendments to this guide can be found in the release notes that accompany the release.

1 Hardware: Overview of Graphics Architectures

This overview briefly reviews fixed-function pipeline architecture using the PSP™ as an example. It follows with a description of the more modern and feature-rich programmable pipeline model, in use on many systems today. The following chapter introduces the PlayStation®Vita graphics architecture, which is a specialized version of the programmable pipeline model.

Fixed-Function Pipeline Graphics Architecture

While fixed-function pipeline graphics architectures offer many features, they are by definition fixed. As such, it is very difficult to extend such systems using software beyond what is offered out of the box.

In the case of PSP™, the hardware itself was developed in a way that loosely mimics the OpenGL® API. So although its graphics API (called libgu) provides low-level access to hardware, programming for the PSP™ remains fairly straightforward to anyone with basic OpenGL API knowledge.

Figure 1 shows the various steps involved in rendering using the PSP™ graphics chip. Note that not all fixed-function pipeline architectures offer all the features shown.

The four overall processing stages are shown as outlines; the various operations are shown in shaded boxes; and the arrow boxes are the user-specified data made available to the system: vertex arrays, index arrays, texture data, frame buffers (or color buffers), and depth buffers. Note that render states, provided through the command buffer, are not displayed as part of the input.

The following sections review the four overall processing stages, and what their job is:

Vertex Processing Stage

The vertex processing stage (VPS) includes transformations such as skinning, blending, and lighting. Ultimately it takes in vertex data, processes it, and writes clip-space positions along with normals, colors, and texture coordinates, as necessary.

Raster Stage

The raster stage takes all data produced in the vertex processing stage and, considering the user-specified primitive (such as a line or triangle), it first performs a back-face test, then performs clipping against the near plane.

With what's left of the primitive and with the use of the viewport and depth range, the raster stage interpolates all data across the primitive. That interpolation is done once per pixel. The interpolated data is sent to the pixel processing stage.

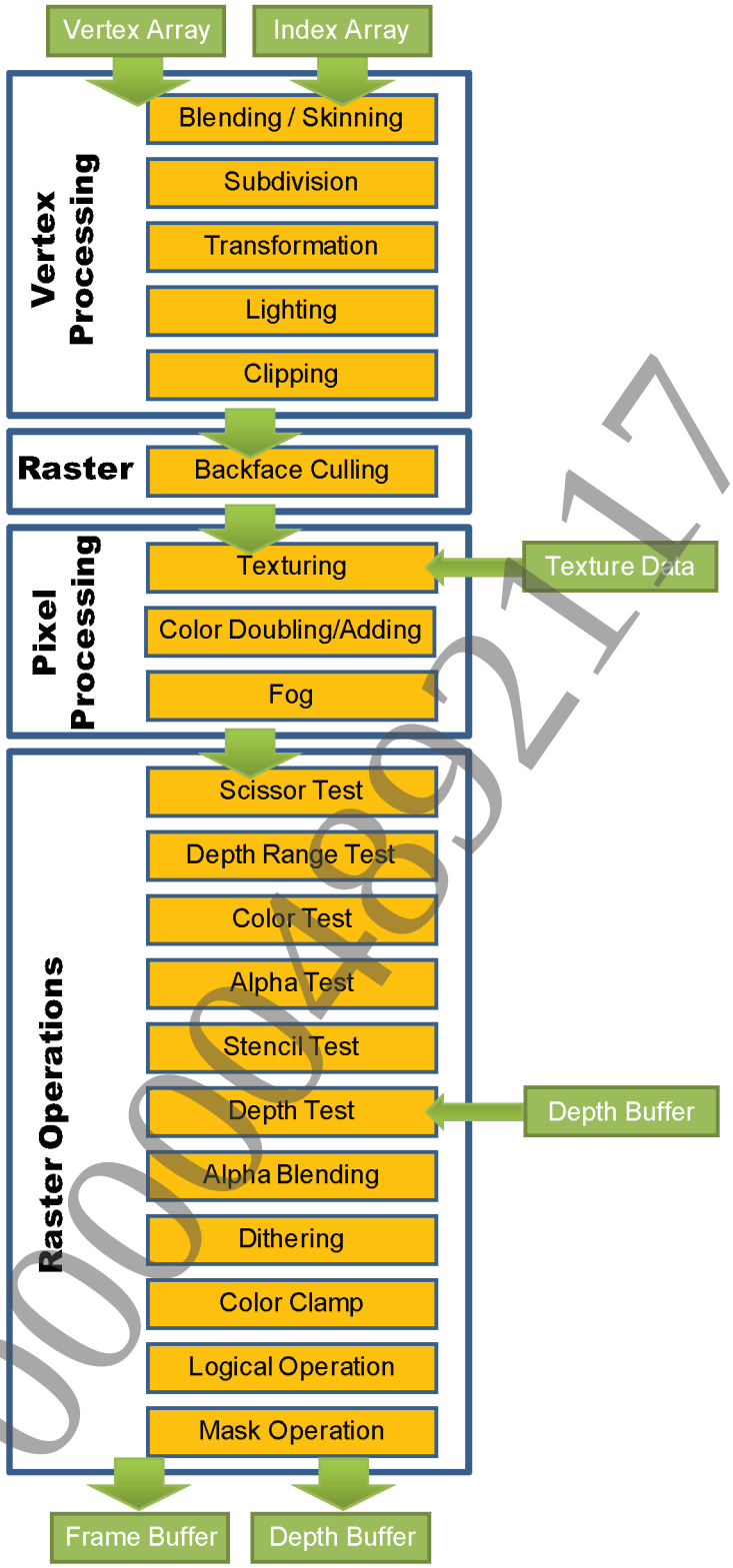
Pixel Processing Stage

The pixel processing stage (PPS) receives interpolated data for each pixel from the raster stage. Using this information, it performs texturing, color, and fog calculations. When done, a final pixel color is sent to the raster operations pipeline stage.

Raster Operations Pipeline Stage

The raster operations pipeline (ROP) stage receives a color (including an alpha component), screen position, and a depth value. It performs the series of tests as shown in Figure 1. Provided all tests are successful, the color and depth value are written into the color and depth buffers, respectively.

Figure 1 Basic PSP™ Graphics Architecture Flow



For more information about the various stages, please refer to the *PSP™ Graphics Engine User's Manual*.

Programmable Pipeline Graphics Architecture

Modern graphics architectures have moved past fixed-function pipelines. OpenGL, DirectX, and even recent consoles (including the PlayStation®3) now offer a programmable pipeline model.

Recent advances in graphics architecture convert key fixed-function pipeline stages into programmable ones. The vertex processing stage (VPS) and pixel processing stage (PPS) saw such conversions.

Despite changes, both stages' respective responsibility remains the same: The VPS reads in vertex attributes, then passes a clip-space position to the raster stage. The PPS receives interpolated data from the raster stage and produces a color. The way the stages perform those tasks is now programmable.

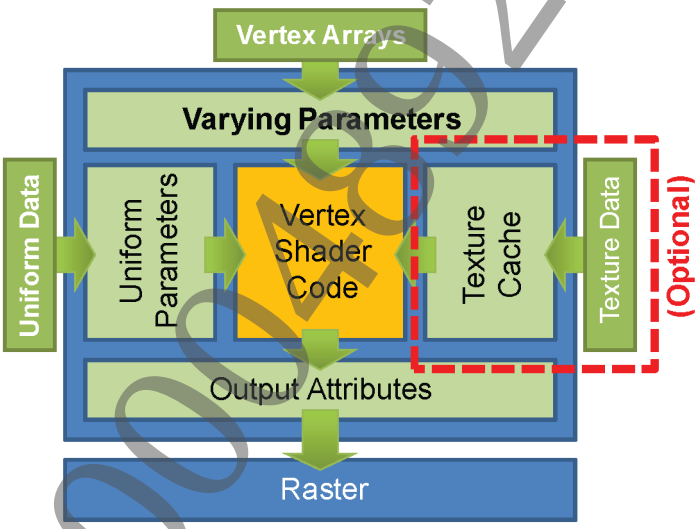
The following sections describe the new additions:

- Programmable Vertex Processing Stage
- Programmable Pixel Processing Stage
- Shader Programming Model
- New Possibilities for Graphics Programming

Programmable Vertex Processing Stage

Figure 2 shows a block diagram of the programmable VPS. In the center is the part that executes the code, called *vertex shader*. It executes code once per vertex.

Figure 2 Programmable Vertex Processing Stage



Just like for fixed-function pipeline architectures, the VPS stage receives vertex data from user-specified vertex arrays. The vertex attributes vary per vertex, and they are accessible by the vertex shader. These arrays contain positions, normals, colors, bone weight, and so on.

Any data necessary for transforming a vertex that remains constant for all vertices within a given draw call can be placed into *uniform parameters*. This data remains uniform from one vertex to another. The best example of a uniform parameter is a local-to-clip transformation matrix, but it can also contain user-defined clip plane equations, light colors, or bone matrices.

Although optional, some platforms allow the VPS to fetch data from a texture, making numerous capabilities, such as displacement mapping, available.

After the vertex shader completes its calculations, it writes *output attributes*, which are picked up by the raster stage.

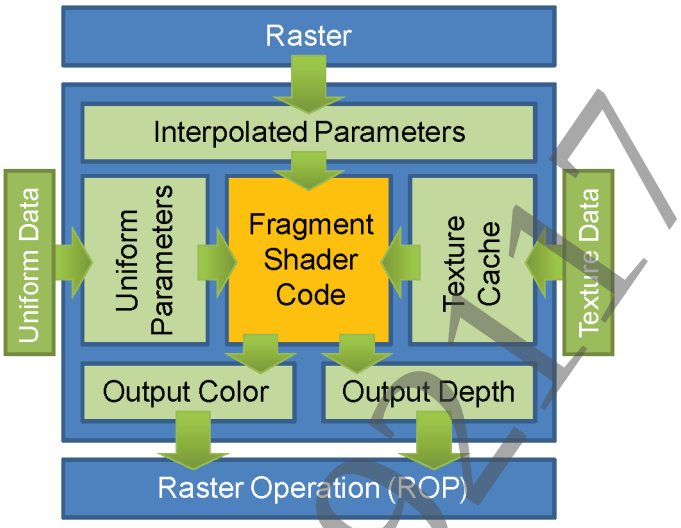
Vertex attributes, uniform parameters, and the texture cache are all read-only, whereas output attributes are write-only. No other data is readable or writable from the vertex shader. This stream-like processing

limitation allows hardware implementations to have multiple vertex shaders running in parallel, without concern that they will interfere with each other.

Programmable Pixel Processing Stage

Figure 3 shows a block diagram of the programmable PPS. The main difference between the PPS and the VPS is that the PPS is executed for each pixel deemed visible by the raster stage, rather than being executed for each vertex.

Figure 3 Programmable Pixel Processing Stage



Interpolated parameters are the values coming in from the raster stage, and are specific for a given pixel. Uniform parameters and the texture cache serve the same role as within the VPS. The *fragment shader* (center block) executes the same code for each pixel. After the shader processing is over, it produces a color that is sent to the raster operation (ROP) stage for various tests before writing into a color buffer. Similarly, if needed for visual effects, the fragment shader can output a depth value and write it to the *output depth*.

Note that writing a depth value from a fragment shader has adverse performance implications. Programmable pipeline architectures typically use the depth value calculated by the raster stage to perform an "early out" test. Overwriting the raster value from a fragment shader disables this optimization.

Note that in the general programmable pipeline architecture, a pixel processing stage does not have access to the destination color buffer unless it is mapped as a texture beforehand. The process described here generates a *source color*, with blending operations performed by the ROP stage.

As in the VPS, interpolated parameters, uniform parameters, and the texture cache are all read-only, whereas output color and output depth are write-only. Also, no other data is readable or writable from the fragment shader, allowing hardware implementations to have multiple fragment shaders running in parallel, without concern that they will interfere with each other. Using this technique, it is not uncommon for graphics chips to process hundreds or even thousands of pixels at once.

Shader Programming Model

The previous sections described vertex shaders and fragment shaders as executing code on each vertex and pixel, respectively. This section addresses the capabilities of the code itself. Although the instruction set of programmable pipeline graphics processors is not as rich as that of CPUs, it is nonetheless quite capable:

- It focuses on 4-element vector operations, as those are so prevalent in graphics processing, including: XYZW Positions, XYZW normals, and RGBA colors.

- It supports more complex requirements using higher-level languages. DirectX has HLSL, OpenGL has GLSL, and there is also Cg (C for graphics), which is available for various platforms including PCs, the PlayStation®3, and now PlayStation®Vita.

The high-level languages used for programming shaders share several characteristics:

- Currently all are based on the C language.
- Additions and limitations to the C language are required to fully expose all shader features.
- All require a compiling phase, either offline or at runtime.

This section shows a basic example demonstrating vertex and fragment shader code.

Vertex Shader

The following vertex shader performs the basic work for rendering a non-skinned model with per-pixel lighting. The position is transformed by the model's local-to-clip transformation matrix. It transfers the normal and texture coordinate over to the fragment shader, where the lighting and texturing calculation are performed.

Note that the `inputVertex` structure elements are coming in as vertex attributes (therefore from vertex buffers), whereas the local-to-clip transformation matrix is marked as `uniform`, indicating that it will remain constant between all the model's vertices. Finally, notice that the `outputVertex` structure elements are written as output attributes, because the `output` variable has the `out` modifier.

```
//--------------------------------------------------------------
// Example of a vertex shader for performing per-pixel lighting
//--------------------------------------------------------------

// Local-to-clip transformation matrix
uniform float4x4 gLocalToClipMatrix;

typedef struct
{
    float4 position     :       POSITION;
    float3 normal       :       NORMAL;
    float2 texCoord     :       TEXCOORD0;
} inputVertex;

typedef struct
{
    float4 position     :       POSITION;
    float2 texCoord     :       TEXCOORD0;
    float3 normal       :       TEXCOORD2;
} outputVertex;

//-----------------------------------
// Main vertex shader entry point
//-----------------------------------
void main (    inputVertex  input,     // Input attributes
          out outputVertex output)     // Output attributes
{
    // Transform the position from local-space to clip-space
    output.position = mul (gLocalToClipMatrix, input.position);

    // Transfer normal and texture coordinate as-is
    output.normal   = input.normal;
    output.texCoord = input.texCoord;
}
```

Fragment Shader

The following fragment shader takes as input the interpolated attributes in `inputFragment`. It uses the various lighting and material properties (defined as `uniform`) to calculate the final terms of lighting in the same method as the OpenGL API (to remain concise, the actual lighting calculations have been omitted, but can be found in many computer graphics references).

Note that the normal (`input.normal`) is normalized to put it into a format appropriate for the raster.

```
sampler2D  gDiffuseMap : TEXUNIT0;

uniform float4    gLightTypeAmbient;
uniform float4    gLightDiffuse;
uniform float4    gLightSpecular;
uniform float4    gLightPosition;
uniform float3    gLightDirection;
uniform float3    gLightAttenuation;

uniform float4    gMaterialAmbient;
uniform float4    gMaterialDiffuse;
uniform float4    gMaterialSpecular;
uniform float     gMaterialShininess;

uniform float4    gGlobalAmbient;
uniform float4    gEyePosition;

typedef struct
{
    float4  position         :      POSITION;
    float2  texCoord         :      TEXCOORD0;
    float3  localPosition    :      TEXCOORD1;
    float3  normal           :      TEXCOORD2;
} inputFragment;

typedef struct
{
    float4 color : COLOR0;
} outputFragment;

//-----------------------------------------------
// Main fragment shader entry point
//-----------------------------------------------
void main (inputFragment input, out outputFragment output)
{
    float3  finalNormal = normalize (input.normal);
    float4  ambientTerm;
    float4  diffuseTerm;
    float4  specularTerm;

    // Calculate lighting's terms here (ambient, diffuse, and
    // specular) based on the fragment's local-space normal
    {
        // CALCULATIONS HERE...
    }

    // Calculate final color based on the lighting
    // terms, material properties, and texture value
    float4  totalLightColor = (diffuseTerm * gMaterialDiffuse) +
        (ambientTerm * gMaterialAmbient);
    totalLightColor = min (totalLightColor, 1.f) *
        tex2d (gDiffuseMap, input.texCoord);
```

```
        // Specular lighting is added in the end as it isn't affected by texture value
        output.color = totalLightColor + (specularTerm * gMaterialSpecular);
    }
```

New Possibilities for Graphics Programming

The addition of programmable stages into the graphics pipeline opens up many possibilities:

- The biggest one is the ability to perform per-pixel calculations. This means Phong shading is now possible where only flat or Gouraud shading was possible before.
- Bump mapping is also possible, along with highly advanced lighting models.
- On the vertex shader side, real displacement mapping is now possible.
- Skinning can also be done with more complex skeletons.
- Coupled with the ability to read multiple textures at once, it is now possible to devise many new visual effects.

At this point, performance becomes the only real bottleneck.

2 Hardware: PlayStation®Vita Graphics Architecture

PlayStation®Vita's graphics processor is the SGX543MP4+ (referred to as "SGX" in this document), a multi-processor chip using tile-based rendering. It is capable of running user-defined vertex and fragment shaders.

To optimize performance, PlayStation®Vita provides libgxm, which is a lower-level API than OpenGL. This API exposes low-level features and components that would normally be hidden within the driver-level of a general-purpose rendering API. Those components are often new to developers, even to those familiar with very recent programming models, so it is particularly important to understand them before proceeding further.

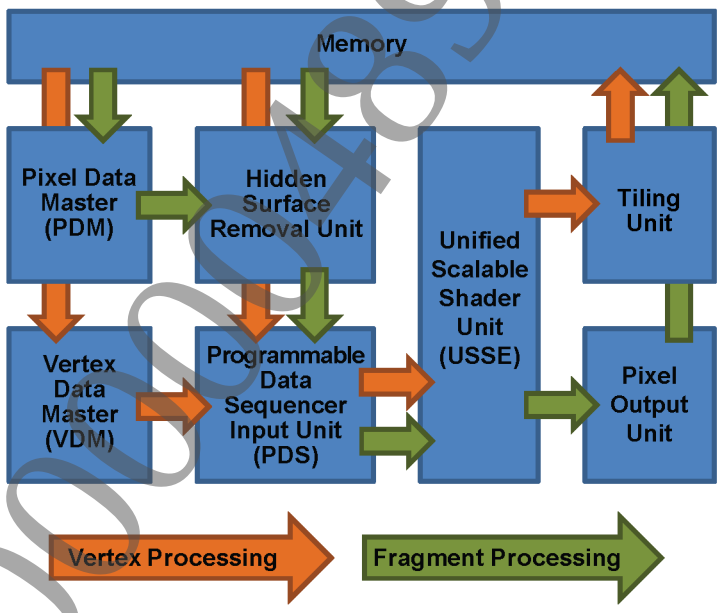
Basic Data Flow

Figure 4 shows the data flow for the SGX. The important thing to note is that many components are used for both vertex and pixel processing. The data flows through the GPU twice: once for vertex processing, where data is stored in tiles, and then again as each tile is rendered.

The Unified Scalable Shader Unit (USSE in Figure 4) performs both vertex and fragment shading.

Note: It's important to know this processing architecture has repercussions in ways the rendering API works. The section libgxm Rendering Pipeline in this document introduces the various repercussions. The *libgxm Overview* has further information on the subject.

Figure 4 SGX Basic Data Flow



Shaders on SGX

PlayStation®Vita uses Cg because it is a high-level shading language; programming directly in assembly is not offered. Therefore all shader programs must be compiled offline into binaries, using a provided command-line tool.

Each binary contains information regarding attributes and uniform parameters. This allows an application to load the binary file and query it using various functions provided in libgxm.

Although general graphics API usually decouple shaders from render states, the way SGX works internally requires it to know some extra information when setting up a shader, which the application may or may not know until a given shader is to be used.

On typical hardware, including PlayStation®3, the reading of vertex data and blending are done by parts of the hardware that are separate from shader code. But on this platform, those things are done as part of the shader code itself. For example, on SGX, a vertex shader must know the format of vertex attributes it will pull data from, and a fragment shader must know the blending equation, because it is ultimately the hardware part which performs blending tasks. The section Patching Shaders describes how this is exposed to developers.

Texture Formats

The SGX offers a rich variety of texture formats. Alongside traditional formats like RGBA8888 (32-bit-per-pixel), it offers PVR lossy compression formats and UBC 1/2/3 formats. fp16x4, fp32x4, and CLUT-based formats are also offered.

Although textures in fixed-function pipeline architectures, including PSP™, are essentially limited to represent actual texture images, the advent of programmable processing stages now allows textures to contain rendering-related data. You can think of textures as data buffers instead of simply texture images.

With textures as data buffers, you can perform a fully-custom gamma correction using a 1D texture. Or implement reflection mapping using a pixel's normal as the texture coordinate into a 2D environment texture. Or devise a scheme where one texture lookup drives another texture lookup.

Typically, fixed-point formats offer better speed, whereas floating-point formats offer better data range representation. If you need to represent a given range of data which is outside of the typical 0.0 – 1.0 mapping, it's possible to use fixed-point formats and perform a linear remapping within the shader code.

For example, an RGBA8888 texture could still be used to represent a range of -1.0 to +1.0, by applying a (2x – 1.0) calculation, though some step precision will be lost. Often such precision is not important for rendering purposes.

Texture Storage Format

libgxm includes a standard texture format for PlayStation®Vita, called GXT, along with functions to interface with it.

Just like for DirectDraw Surface (DDS) files, GXT files are data-format agnostic. That said, they are targeted to libgxm in that they include ready-to-use `SceGxmTexture` structures, and their memory layout is ready-to-use after it is loaded in memory.

It is therefore recommended to use the GXT format, at least at the start of a project.

libgxm Rendering Pipeline

libgxm manages and encapsulates many low-level data structures for you. For example, command-buffer creation is fully hidden from developers. Also, libgxm internally manages various ring-buffers, for which all you will have to do is provide a size. The following sections detail these features.

Command Buffer Creation

Whereas embedded systems like PSP™ allow for the direct creation of command buffers, libgxm handles all of it internally. Command buffers are much lighter on this architecture than others. On SGX, most commands do not sit in the command buffer itself, but are instead copied into context structures which are then referred to from the command buffer, through pointers.

Concept of a Scene

In standard, programmable-pipeline graphics hardware architectures, submitting draw-calls to the GPU with associated vertex pointers, texture sampling information, and shader constants is straightforward. The GPU hardware consumes the submitted vertex and fragment data commands in sequence, so the commands can be stored in a simple ring-buffer that is fed sequentially to the GPU. The draw-call routines look like the following example pseudo-code:

```
// Pseudo
while(rendering)
{
    SetRenderTarget(s)();
    ClearScreen();
    forEachObject()
    {
        SetShaders();
        SetTextures();
        SetUniforms();
        SetGeometryPointers();
        DrawIndexLists();
    }
    Swap();
}
```

There are, however, a few differences between general programmable pipeline architecture and what's needed to support and optimize for the SGX internal architecture.

In the SGX architecture, the internal graphics commands are not all processed at one time. As the fragment processing code and data needs to remain resident in memory until the vertex stage is complete, a more involved memory management scheme is required.

libgxm manages resources in the background in an efficient manner, while providing an intuitive programming interface for submitting graphics state and draw call information to the SGX. This API abstracts many of the internal mechanisms of the SGX, so from a programming perspective it can be treated very similarly to the generic programmable pipeline draw-call routines.

If we return to the GPU vertex and fragment stages as mentioned previously, the GPU at some point has to switch between vertex and fragment processing to complete the rendering of a given color surface. To know when it is safe to switch from vertex to fragment processing, the SGX needs to know which draw calls correspond to a color surface. libgxm has the concept of a *scene* to enclose a set of draw calls for rendering. The actual processing of render commands will not start until a scene is completed.

The scene concept is exposed in libgxm two function calls: `sceGxmBeginScene()` and `sceGxmEndScene()`. With those calls in place, the previous pseudo-code becomes:

```
// Pseudo
while(rendering)
{
    // Start of draw-call submission to render-target
    BeginScene(RenderTarget *pRenderTarget);
    ClearScreen();
    forEachObject()
    {
        SetShaders();
        SetTextures();
        SetUniforms();
        SetGeometryPointers();
        DrawIndexLists();
    }
    // End of current draw call submission ('Kick' the GPU)
    EndScene();
    Swap();
}
```

Other differences relate to the way data is managed internally due to the architecture. The management of shaders and uniform data is much more involved on the SGX than on other shading architectures.

Because this architecture is capable of overlapping vertex and fragment processing calls over different kicking boundaries, there will be many vertex and fragment operations in flight at any given time.

Display Queue

After a frame is completely rendered, it is necessary to let the hardware know it's time to display the image on screen. To avoid visual glitches and/or artifacts, such as tearing, it is necessary to synchronize the display with either the horizontal or vertical sync.

libgxm offers the concept of a *display queue* to perform this synchronization. When the CPU completes the submission of rendering commands, it can add an entry into the queue, passing in the sync object corresponding to a scene.

When that scene is finished rendering, libgxm calls a user-provided callback function to safely flip to the new display buffer.

Various Ring Buffers

Because SGX commands are not all executed at one time, vertex and fragment shaders, as well as their uniform parameters, must remain available until draw calls using them are complete. libgxm buffers the commands internally. The only thing developers have to provide is the size of the ring buffers.

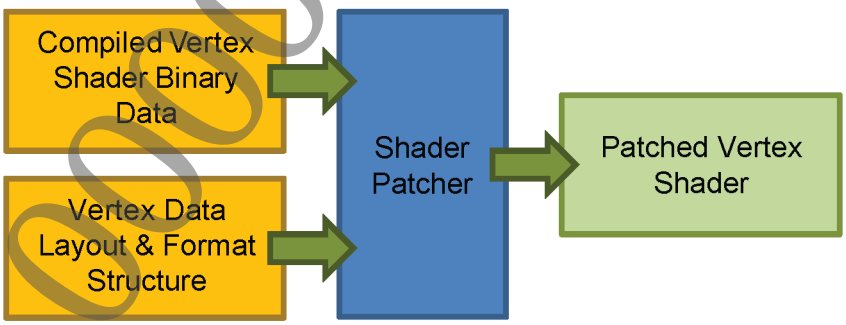
Note that libgxm performs the necessary synchronization to avoid overflows and deadlocks related to those buffers.

Patching Shaders

Shaders on SGX perform more work than on other programmable pipeline graphics architectures. This forces a binding between shaders and other render states which might not be immediately obvious. For performance reasons, this is exposed to developers by libgxm.

When you compile a Cg shader, you may not know which blending equation or vertex attribute formats will be used. The PlayStation®Vita Cg compiler produces a *pre-patched shader binary data* as shown in Figure 5. At runtime, it is necessary to perform a patching step that incorporates the inputs prior to using shaders.

Figure 5 Data Necessary for Patching a Vertex Shader



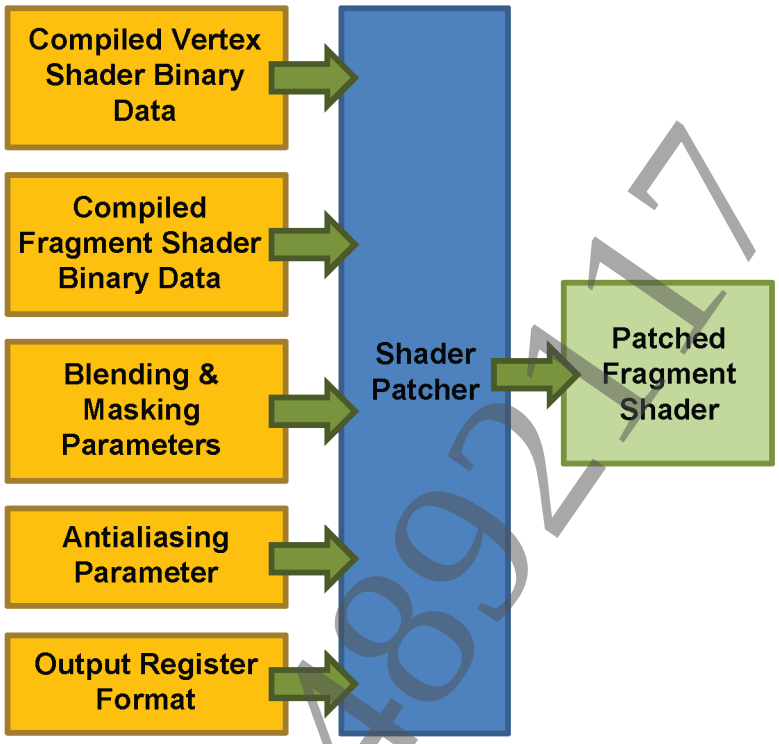
Because the input may be different depending on how the shader binary is used, you may need to generate multiple patched shaders, one for each combination of shader binary and input needed at runtime.

- Vertex shaders are involved in the process of reading vertex attributes, and must therefore know where vertex arrays are, and what their layout and format is. Considering this, if a given compiled vertex shader is used to render 20 meshes using 3 different layout/format combinations, you then need to generate 3 patched vertex shaders.

- Fragment shaders are involved in blending, anti-aliasing, write masking, and in reading data output from vertex shaders. For this, a patched fragment shader will need to know that information. For example, in order to let the fragment shader know about the output from a vertex shader, one must simply provide a pointer to the patched vertex shader. If you are using consecutive `TEXCOORD` attributes, you don't even need to pass the pointer.

Figure 6 lists the input to the patching operation.

Figure 6 Data Necessary for Patching a Fragment Shader



A simple and naive renderer implementation might internally store all patching-state information (such as the vertex array format and blending equation), and right before a draw call is to be performed, it can patch the current vertex and fragment shader.

Keep in mind that the patch process is fairly expensive, so although patching immediately before the draw call is possible, it is recommended instead to pre-patch shaders at initialization time.

Graphics Debugging and Profiling

Contrary to PSP™, where SN Systems Tuner is available for CPU-only profiling, PlayStation®Vita offers a tool called Razor for PlayStation®Vita (Razor), which offers graphics debugging and profiling, alongside CPU-side profiling. Razor is integrated into the Visual Studio Integration (VSI), alongside the compiler and debugger.

Razor allows developers to quickly get an overview of runtime performance on the target using the head-up display (HUD), and offers the ability to capture a game frame and inspect its content offline, including libgxm commands, textures, shaders, and vertex arrays.

Razor also offers *conditional profiling*, which allows you to try various optimization scenarios on a captured scene to look for performance gains when implemented in the target-side application.

Razor allows you to debug vertex and fragment shaders. You specify a given vertex or pixel, and single-step Cg instructions and inspect variables along the way.

For more information on the subject, refer to *Performance Analysis and GPU Debugging*.

3 Software: Performing Basic Graphics Tasks on PlayStation®Vita

This section describes specific examples of how to program using libgxm. You may recognize some of these code snippets from various low-level samples in the PlayStation®Vita SDK.

Initializing libgxm

The following code initializes libgxm. We recommend you use the default values provided for the various buffer sizes initially and modify them as you become more familiar with your application's needs.

```
// Allocate a block of memory the size specified by size and size-alignment
void *graphicsAlloc(SceKernelMemBlockType type, uint32_t size, uint32_t
    alignment, uint32_t attribs, SceUID *uid)
{
    int err = SCE_OK;

    // Page align the size
    if (type == SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RW) {
        // CDRAM memblocks must be 256 KiB aligned
        SCE_DBG_ASSERT(alignment <= 256*1024);
        size = ALIGN(size, 256*1024);
    } else {
        // LPDDR memblocks must be 4 KiB aligned
        SCE_DBG_ASSERT(alignment <= 4*1024);
        size = ALIGN(size, 4*1024);
    }

    // Allocate some memory
    *uid = sceKernelAllocMemBlock("common", type, size, NULL);
    SCE_DBG_ASSERT(*uid >= SCE_OK);

    // Retrieve the base address
    void *mem = NULL;
    err = sceKernelGetMemBlockBase(*uid, &mem);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Map for the GPU
    err = sceGxmMapMemory(mem, size, attribs);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Done
    return mem;
}

// (Somewhere in an initialization sequence)

#define  DISPLAY_MAX_PENDING_SWAPS  2

// Set up parameters
SceGxmInitializeParams initializeParams;
memset(&initializeParams, 0, sizeof(SceGxmInitializeParams));
initializeParams.flags = 0;
initializeParams.displayQueueMaxPendingCount = DISPLAY_MAX_PENDING_SWAPS;
initializeParams.displayQueueCallback = displayCallback;
initializeParams.displayQueueCallbackDataSize = sizeof(DisplayData);
initializeParams.parameterBufferSize = SCE_GXM_DEFAULT_PARAMETER_BUFFER_SIZE;
```

```
// Start libgxm
err = sceGxmInitialize(&initializeParams);
SCE_DBG_ASSERT(err == SCE_OK);

// Create a rendering context
// Allocate host memory
g_contextHostMem = malloc(SCE_GXM_MINIMUM_CONTEXT_HOST_MEM_SIZE);

// Allocate ring buffer memory using default sizes
void *vdmRingBuffer = graphicsAlloc(
    SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
    SCE_GXM_DEFAULT_VDM_RING_BUFFER_SIZE,
    4,
    SCE_GXM_MEMORY_ATTRIB_READ,
    &g_vdmRingBufferUid);
void *vertexRingBuffer = graphicsAlloc(
    SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
    SCE_GXM_DEFAULT_VERTEX_RING_BUFFER_SIZE,
    4,
    SCE_GXM_MEMORY_ATTRIB_READ,
    &g_vertexRingBufferUid);
void *fragmentRingBuffer = graphicsAlloc(
    SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
    SCE_GXM_DEFAULT_FRAGMENT_RING_BUFFER_SIZE,
    4,
    SCE_GXM_MEMORY_ATTRIB_READ,
    &g_fragmentRingBufferUid);
uint32_t fragmentUsseRingBufferOffset;
void *fragmentUsseRingBuffer = fragmentUsseAlloc(
    SCE_GXM_DEFAULT_FRAGMENT_USSE_RING_BUFFER_SIZE,
    &g_fragmentUsseRingBufferUid,
    &fragmentUsseRingBufferOffset);

// Set up parameters
SceGxmContextParams contextParams;
memset(&contextParams, 0, sizeof(SceGxmContextParams));
contextParams.hostMem = g_contextHostMem;
contextParams.hostMemSize = SCE_GXM_MINIMUM_CONTEXT_HOST_MEM_SIZE;
contextParams.vdmRingBufferMem = vdmRingBuffer;
contextParams.vdmRingBufferMemSize = SCE_GXM_DEFAULT_VDM_RING_BUFFER_SIZE;
contextParams.vertexRingBufferMem = vertexRingBuffer;
contextParams.vertexRingBufferMemSize =
    SCE_GXM_DEFAULT_VERTEX_RING_BUFFER_SIZE;
contextParams.fragmentRingBufferMem = fragmentRingBuffer;
contextParams.fragmentRingBufferMemSize =
    SCE_GXM_DEFAULT_FRAGMENT_RING_BUFFER_SIZE;
contextParams.fragmentUsseRingBufferMem = fragmentUsseRingBuffer;
contextParams.fragmentUsseRingBufferMemSize =
    SCE_GXM_DEFAULT_FRAGMENT_USSE_RING_BUFFER_SIZE;
contextParams.fragmentUsseRingBufferOffset = fragmentUsseRingBufferOffset;

// Create the context
err = sceGxmCreateContext(&contextParams, &g_context);
SCE_DBG_ASSERT(err == SCE_OK);
```

Setting up Display Buffers

Looking at various SDK samples, you will notice that a triple-buffering scheme is used for display buffers. This is not a hardware requirement; you can choose to use a double-buffering scheme instead.

Because the render process writes its results into display buffers, you need to associate a color surface structure with each buffer. A given display buffer and its associated color surface both write to the same memory space. Whereas color buffers only require an 8 byte alignment, display buffers must be 256 byte-aligned. Therefore you need to set up the memory space on 256 byte alignment. Failure to do so will result in errors while flipping display buffers.

While setting up display buffers, it is a good idea to also set up a *sync object* for each buffer. This is required to synchronize between rendering and flip operations.

```
// (Somewhere in an initialization sequence)
    #define  DISPLAY_BUFFER_COUNT  3
    #define  MSAA_MODE SCE_GXM_MULTISAMPLE_NONE
    #define  DISPLAY_STRIDE_IN_PIXELS  1024
    #define  DISPLAY_COLOR_FORMAT SCE_GXM_COLOR_FORMAT_A8B8G8R8
    #define  DISPLAY_WIDTH  960
    #define  DISPLAY_HEIGHT  544
    int err = SCE_OK;
    void *displayBufferData[DISPLAY_BUFFER_COUNT];
    SceUID displayBufferUid[DISPLAY_BUFFER_COUNT];
    SceGxmColorSurface displaySurface[DISPLAY_BUFFER_COUNT];
    SceGxmSyncObject displayBufferSync[DISPLAY_BUFFER_COUNT];

    for (uint32_t i = 0; i < DISPLAY_BUFFER_COUNT; ++i) {
       // Allocate memory with large alignment to ensure physical contiguity
       displayBufferData[i] = graphicsAlloc(
          SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RW,
          ALIGN (4*DISPLAY_STRIDE_IN_PIXELS*DISPLAY_HEIGHT, 1*1024*1024),
          SCE_GXM_COLOR_SURFACE_ALIGNMENT,
          SCE_GXM_MEMORY_ATTRIB_READ | SCE_GXM_MEMORY_ATTRIB_WRITE,
          &displayBufferUid[i]);
       // Memset the buffer to a noticeable debug color (optional)
       for (uint32_t y = 0; y < DISPLAY_HEIGHT; ++y) {
          uint32_t *row = (uint32_t *)displayBufferData[i] +
             y*DISPLAY_STRIDE_IN_PIXELS;
          for (uint32_t x = 0; x < DISPLAY_WIDTH; ++x) {
             row[x] = 0xffff00ff;
          }
       }

       // Initialize a color surface for this display buffer
       err = sceGxmColorSurfaceInit(
          &displaySurface[i],
          DISPLAY_COLOR_FORMAT,
          SCE_GXM_COLOR_SURFACE_LINEAR,
          (MSAA_MODE == SCE_GXM_MULTISAMPLE_NONE)
             ? SCE_GXM_COLOR_SURFACE_SCALE_NONE
             : SCE_GXM_COLOR_SURFACE_SCALE_MSAA_DOWNSCALE,
          SCE_GXM_OUTPUT_REGISTER_SIZE_32BIT,
          DISPLAY_WIDTH,
          DISPLAY_HEIGHT,
          DISPLAY_STRIDE_IN_PIXELS,
          displayBufferData[i]);
          SCE_DBG_ASSERT(err == SCE_OK);
       // Create a sync object that we will associate with this buffer
       err = sceGxmSyncObjectCreate(&displayBufferSync[i]);
       SCE_DBG_ASSERT(err == SCE_OK);
    }
```

Setting up Color and Depth Surfaces

For very simple applications, it may be enough to use the display buffers' color surface, but as more visual effects are added, you may need to add extra color surfaces.

To ensure correct behavior during partial render:

- If depth tests other than NEVER or ALWAYS are required within the scene, a depth/stencil surface must be provided that contains depth data as part of its format.
- If stencil tests other than NEVER or ALWAYS are required within the scene, a depth/stencil buffer must be provided that contains stencil data as part of its format.
- If the depth and stencil test will be NEVER or ALWAYS for all draw calls in the scene, you do not need to provide the depth/stencil surface.

The following code example shows the allocation and initialization of an additional pair of color and depth surface, which in this case is referred to as an *off-screen* surface.

```
#define  OFFSCREEN_WIDTH 256
#define  OFFSCREEN_HEIGHT 256
#define  OFFSCREEN_COLOR_FORMAT SCE_GXM_COLOR_FORMAT_A8B8G8R8
#define  OFFSCREEN_DEPTH_FORMAT SCE_GXM_DEPTH_STENCIL_FORMAT_DF32
#define  MSAA_MODE SCE_GXM_MULTISAMPLE_NONE

int err = SCE_OK;

// Allocate memory
g_offscreenColorBufferData = graphicsAlloc
(SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RW
    OFFSCREEN_WIDTH*OFFSCREEN_HEIGHT*4,
    MAX (SCE_GXM_TEXTURE_ALIGNMENT,
        SCE_GXM_COLOR_SURFACE_ALIGNMENT),
    SCE_GXM_MEMORY_ATTRIB_READ |
    SCE_GXM_MEMORY_ATTRIB_WRITE,
    &g_offscreenColorBufferUid);
g_offscreenDepthBufferData = graphicsAlloc
(SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RW
    OFFSCREEN_WIDTH*OFFSCREEN_HEIGHT*4,
    MAX (SCE_GXM_TEXTURE_ALIGNMENT,
    SCE_GXM_DEPTHSTENCIL_SURFACE_ALIGNMENT),
    SCE_GXM_MEMORY_ATTRIB_READ |
    SCE_GXM_MEMORY_ATTRIB_WRITE,
    &g_offscreenDepthBufferUid);

// Set up the color surface
err = sceGxmColorSurfaceInit(&g_offscreenColorSurface,
    OFFSCREEN_COLOR_FORMAT,
    SCE_GXM_COLOR_SURFACE_LINEAR,
     (MSAA_MODE != SCE_GXM_MULTISAMPLE_NONE)
        ? SCE_GXM_COLOR_SURFACE_SCALE_MSAA_DOWNSCALE
        : SCE_GXM_COLOR_SURFACE_SCALE_NONE,
    SCE_GXM_OUTPUT_REGISTER_SIZE_32BIT,
    OFFSCREEN_WIDTH,
    OFFSCREEN_HEIGHT,
    OFFSCREEN_WIDTH,
    g_offscreenColorBufferData);
SCE_DBG_ASSERT(err == SCE_OK);

// Set up the depth surface
err = sceGxmDepthStencilSurfaceInit(&g_offscreenDepthSurface,
    OFFSCREEN_DEPTH_FORMAT,
    SCE_GXM_DEPTH_STENCIL_SURFACE_TILED,
    OFFSCREEN_WIDTH,
```

```
    g_offscreenDepthBufferData,
    NULL);
SCE_DBG_ASSERT(err == SCE_OK);
```

Extracting Parameter Information from a Shader

It is possible to extract information out of the binary data using libgxm helper functions. All related function names start with `sceGxmProgram`, whereas specific functions for extracting parameter information start with `sceGxmProgramParameter`. The following example prints out the name and register index for each parameter from the binary vertex shader. It assumes that the shader is already compiled using the PlayStation®Vita SDK Cg compiler and loaded in memory.

```
extern const SceGxmProgram _binary_basic_v_gxp_start;
uint32_t paramCount = sceGxmProgramGetParameterCount
(&_binary_basic_v_gxp_start);

// Go through all of the shader's parameters
for (uint32_t paramIndex = 0; paramIndex < paramCount; paramIndex++)
{
    const SceGxmProgramParameter *param;
    const char *paramName;
    uint32_t paramResourceIndex;

    // Get a pointer to the shader's Nth parameter
    param = sceGxmProgramGetParameter (&_binary_basic_v_gxp_start, paramIndex);

    // Get parameter name and resource index
    paramName = sceGxmProgramParameterGetName (param);
    paramResourceIndex = sceGxmProgramParameterGetResourceIndex (param);

    printf ("Param[%d]: Name=%s ResIndex=%d\n", paramIndex, paramName,
        paramResourceIndex);
}
```

Typically the application extracts this information at initialization or load time, and it stores the results in its own data structure for ease and speed of access.

Patching a Vertex and Fragment Shader

The next step is *patching* the shaders to bind them to vertex data arrays and blending equations. These steps assume that the shader is already compiled using the PlayStation®Vita SDK Cg compiler and loaded in memory.

There are three steps for patching binary shaders:

(1) Create the patcher once, at initialization time:

```
// This function could allocate from any memory allocator.
static void *patcherHostAlloc(void *userData, uint32_t size)
{
    UNUSED(userData);
    return malloc(size);
}

// This function could free from any memory allocator.
static void patcherHostFree(void *userData, void *mem)
{
    UNUSED(userData);
    free(mem);
}
```

```
// Set buffer sizes for this sample
const uint32_t patcherBufferSize = 64*1024;
const uint32_t patcherVertexUsseSize = 64*1024;
const uint32_t patcherFragmentUsseSize = 64*1024;

// Allocate memory for buffers and USSE code
SceUID patcherBufferUid;
void *patcherBuffer = graphicsAlloc(
    SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
    patcherBufferSize,
    4,
    SCE_GXM_MEMORY_ATTRIB_READ | SCE_GXM_MEMORY_ATTRIB_WRITE,
    &patcherBufferUid);
SceUID patcherVertexUsseUid;
uint32_t patcherVertexUsseOffset;
void *patcherVertexUsse = vertexUsseAlloc(
    patcherVertexUsseSize,
    &patcherVertexUsseUid,
    &patcherVertexUsseOffset);
SceUID patcherFragmentUsseUid;
uint32_t patcherFragmentUsseOffset;
void *patcherFragmentUsse = fragmentUsseAlloc(
    patcherFragmentUsseSize,
    &patcherFragmentUsseUid,
    &patcherFragmentUsseOffset);

// Create a shader patcher
SceGxmShaderPatcherParams patcherParams;
memset(&patcherParams, 0, sizeof(SceGxmShaderPatcherParams));
patcherParams.userData = NULL;
patcherParams.hostAllocCallback = &patcherHostAlloc;
patcherParams.hostFreeCallback = &patcherHostFree;
patcherParams.bufferAllocCallback = NULL;
patcherParams.bufferFreeCallback = NULL;
patcherParams.bufferMem = patcherBuffer;
patcherParams.bufferMemSize = patcherBufferSize;
patcherParams.vertexUsseAllocCallback = NULL;
patcherParams.vertexUsseFreeCallback = NULL;
patcherParams.vertexUsseMem = patcherVertexUsse;
patcherParams.vertexUsseMemSize = patcherVertexUsseSize;
patcherParams.vertexUsseOffset = patcherVertexUsseOffset;
patcherParams.fragmentUsseAllocCallback = NULL;
patcherParams.fragmentUsseFreeCallback = NULL;
patcherParams.fragmentUsseMem = patcherFragmentUsse;
patcherParams.fragmentUsseMemSize = patcherFragmentUsseSize;
patcherParams.fragmentUsseOffset = patcherFragmentUsseOffset;

SceGxmShaderPatcher *shaderPatcher = NULL;
int err = sceGxmShaderPatcherCreate(&patcherParams, &shaderPatcher);
SCE_DBG_ASSERT(err == SCE_OK);
```

(2) Register a shader with the patcher:

```
extern const SceGxmProgram _binary_basic_v_gxp_start;
extern const SceGxmProgram _binary_basic_f_gxp_start;
SceGxmShaderPatcherId basicVertexProgramId;
SceGxmShaderPatcherId basicFragmentProgramId;
int err = SCE_OK;

// Register programs with the patcher
err = sceGxmShaderPatcherRegisterProgram(shaderPatcher,
&_binary_basic_v_gxp_start,
```

```
&basicVertexProgramId);
SCE_DBG_ASSERT(err == SCE_OK);
err = sceGxmShaderPatcherRegisterProgram(shaderPatcher,
&_binary_basic_f_gxp_start,
    &basicFragmentProgramId);
SCE_DBG_ASSERT(err == SCE_OK);
```

(3) Patch the shader:

If at all possible, perform this step once at initialization time or load time. Because the patching process is quite expensive, try to avoid running the patching operation right before a draw call is about to be done.

```
#define  MSAA_MODE  SCE_GXM_MULTISAMPLE_NONE

// Data structure for basic geometry
typedef struct BasicVertex {
    float x;
    float y;
    float z;
    uint32_t color;
} BasicVertex;

SceGxmVertexAttribute basicVertexAttributes[2];
SceGxmVertexStream basicVertexStreams[1];
SceGxmVertexProgram *basicVertexProgram = NULL;
SceGxmFragmentProgram *basicFragmentProgram = NULL;
int err = SCE_OK;

// Create shaded triangle vertex format. Our triangle's vertex data consists
// of interleaved positions and colors into a single vertex stream.
// Positions format are FP32x3, while colors format are A8B8G8R8
//(16 bytes per vertex)
{
    // Position data is in vertex stream 0, has format FP32x3,
    // and will be bound to shader parameter named "aPosition".
    basicVertexAttributes[0].streamIndex = 0;
    basicVertexAttributes[0].offset = 0;
    basicVertexAttributes[0].format = SCE_GXM_ATTRIBUTE_FORMAT_F32;
    basicVertexAttributes[0].componentCount = 3;
    basicVertexAttributes[0].regIndex =
        sceGxmProgramParameterGetResourceIndex (
        sceGxmProgramFindParameterByName (
        basicVertexProgramId,
        "aPosition"));

    // Color data is in vertex stream 0, has format A8B8G8R8,
    // and will be bound to shader parameter named "aColor".
    basicVertexAttributes[1].streamIndex = 0;
    basicVertexAttributes[1].offset = 12;
    basicVertexAttributes[1].format = SCE_GXM_ATTRIBUTE_FORMAT_U8N;
    basicVertexAttributes[1].componentCount = 4;
    basicVertexAttributes[1].regIndex =
        sceGxmProgramParameterGetResourceIndex (
        sceGxmProgramFindParameterByName (
        basicVertexProgramId,
        "aColor");

    // We have 1 vertex stream with a vertex stride of 16 bytes.
    // We will index into it using 16-bit indices.
    basicVertexStreams[0].stride = sizeof(BasicVertex);
    basicVertexStreams[0].indexSource = SCE_GXM_INDEX_SOURCE_INDEX_16BIT;
}
```

```
// Create shaded triangle shaders
err = sceGxmShaderPatcherCreateVertexProgram(
    shaderPatcher,              // Previously created shader patcher
    basicVertexProgramId,       // Shader ID returned by patcher
    basicVertexAttributes,      // Attribute array defined above
    2,                          // Nbr entries in attribute array
    basicVertexStreams,         // Vertex stream array defined above
    1,                          // Nbr entries in vertex stream array
    &basicVertexProgram);       // Resulting patched vertex shader
SCE_DBG_ASSERT(err == SCE_OK);
err = sceGxmShaderPatcherCreateFragmentProgram(
    shaderPatcher,              // Previously created shader patcher
    basicFragmentProgramId,     // Shader ID returned by patcher
    SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4,    // Output register format
    MSAA_MODE,                  // MSAA mode
    NULL,                       // Blend equation (NULL if none)
    &_binary_basic_v_gxp_start, // Vertex shader used with fragment shader
    &basicFragmentProgram);     // Resulting patched fragment shader
SCE_DBG_ASSERT(err == SCE_OK);
```

Using Alpha Blending

Alpha blending is a render state that is rolled into shaders. More specifically, it is specified at the time of the fragment shader patching. The previous example (Patch the shader) passed NULL as the blend equation (the fifth argument to `sceGxmShaderPatcherCreateFragmentProgram()`). Passing in NULL disables blending.

To enable blending using a shader patcher, do not pass in NULL, but instead pass a pointer to a `SceGxmBlendInfo` structure as the fifth argument. Note that this structure also contains a color mask. The fragment shader also controls color masking.

The following example replaces the previous example's `sceGxmShaderPatcherCreateFragmentProgram()` call. It sets additive blending, based on source alpha for the color component, while the source alpha is set to overwrite the destination alpha completely:

```
SceGxmBlendingInfo  blendInfo;
int err = SCE_OK;

blendInfo.colorMask = SCE_GXM_COLOR_MASK_ALL;
blendInfo.colorFunc = SCE_GXM_BLEND_FUNC_ADD;
blendInfo.alphaFunc = SCE_GXM_BLEND_FUNC_ADD;
blendInfo.colorSrc = SCE_GXM_BLEND_FACTOR_SRC_ALPHA;
blendInfo.colorDst = SCE_GXM_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
blendInfo.alphaSrc = SCE_GXM_BLEND_FACTOR_ONE;
blendInfo.alphaDst = SCE_GXM_BLEND_FACTOR_ZERO;

err = sceGxmShaderPatcherCreateFragmentProgram(
    shaderPatcher,              // Previously created shader patcher
    basicFragmentProgramId,     // Shader ID returned by patcher
    MSAA_MODE,                  // MSAA mode
    &blendInfo,                 // Blend equation
    &_binary_basic_v_gxp_start, // Vertex shader used in conjunction with
                                // fragment shader
    &basicFragmentProgram);     // Resulting patched fragment shader
SCE_DBG_ASSERT(err == SCE_OK);
```

Beginning and Ending a Scene

The concept of a scene is usually not present in fixed-function pipeline rendering architectures. But in libgxm, every render state setting and draw call must occur between a `sceGxmBeginScene()` and `sceGxmEndScene()` pair, or else an error will occur.

To begin a scene, you must first decide which color and depth buffers you want to draw into, if any. You can omit one or the other, but not both. You must then set a `SceGxmRenderTarget` (although this is only the tiling setup piece of the render target information) and `SceGxmScene` structure (contains synchronization-related items, like a sync object).

Before beginning to render a scene, the previous scene must be complete.

```
int err = SCE_OK;

// Set up render target parameters
SceGxmRenderTargetParams renderTargetParams;
memset(&renderTargetParams, 0, sizeof(SceGxmRenderTargetParams));
renderTargetParams.flags = 0;
renderTargetParams.width = DISPLAY_WIDTH;
renderTargetParams.height = DISPLAY_HEIGHT;
renderTargetParams.scenesPerFrame = 1;
renderTargetParams.multisampleMode = MSAA_MODE;
renderTargetParams.multisampleLocations = 0;
renderTargetParams.driverMemBlock = SCE_UID_INVALID_UID;

{
    // Compute sizes
    uint32_t  driverMemSize;
    sceGxmGetRenderTargetMemSizes(&renderTargetParams, &driverMemSize);

    // Allocate driver memory
    renderTargetParams.driverMemBlock = sceKernelAllocMemBlock(
        "SampleRT",
        SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
        driverMemSize,
        NULL);
}

// Create the render target
SceGxmRenderTarget *renderTarget;
err = sceGxmCreateRenderTarget(&renderTargetParams, &renderTarget);
SCE_DBG_ASSERT(err == SCE_OK);

// Start rendering to the main render target
err = sceGxmBeginScene(
    context,                            // libgxm context
    0,                                  // Sync flags
    renderTarget,                       // Ptr to render target tiling setup
    NULL                                // Valid region, NULL if none
    NULL,                               // Vertex sync object, NULL if none
    displayBufferSync[backBufferIndex], // Fragment sync object, NULL if none
    &displaySurface[backBufferIndex],   // Ptr to color surface, NULL if none
    &depthSurface);                     // Ptr to depth surface, NULL if none
SCE_DBG_ASSERT(err == SCE_OK);

// ...
// Perform render state changes and draw calls here...
// ...
```

```
// Stop rendering to the color/depth buffer
sceGxmEndScene(
    context,                                    // libgxm context
    NULL,                                       // Vertex notification, NULL if none
    NULL);                                      // Fragment notification, NULL if none
```

Creating and Setting Up Textures

Ultimately, a texture is simply data in memory. How the texture data makes it to memory is up to the application. For example, the libgxm samples currently link GXT textures into the executable for simplicity. Loading textures from files is a better way to proceed for more complex applications.

The following example shows how one can load and initialize a texture from a GXT file:

```
const void *GXTFileInMemory;
int err = SCE_OK;
unsigned int nbrTexturesInGXTFile;
unsigned int textureDataSize;
const void *textureDataPointer;
SceGxmTexture texture;
unsigned int textureIndexToUse = 0;

// ...
// Data is loaded into memory, one way or another, and 'GXTFileInMemory'
// will point to it.
// ...

// Extract info from GXT file buffer
{
    err = sceGxtCheckData(GXTFileInMemory);
    SCE_DBG_ASSERT(err == SCE_OK);

    // GXT files can contain more than one texture.
    // Get texture count from GXT file.
    nbrTexturesInGXTFile = sceGxtGetTextureCount(GXTFileInMemory);
    SCE_DBG_ASSERT(nbrTexturesInGXTFile > 0);

    // Get texture data size within the GXT file buffer
    textureDataSize = sceGxtGetDataSize(GXTFileInMemory);
    SCE_DBG_ASSERT(textureDataSize);

    // Get texture data pointer within the GXT file buffer
    textureDataPointer = sceGxtGetDataAddress(GXTFileInMemory);
    SCE_DBG_ASSERT(textureDataPointer);

    err = sceGxtInitTexture(&texture,  // Texture structure to be filled out
       GXTFileInMemory,                // Pointer to GXT file buffer
       textureDataPointer,             // Pointer to the actual texture data
       textureIndexToUse);             // Texture index within GXT file buffer
    SCE_DBG_ASSERT(err == SCE_OK);
}
```

SGX allows up to 16 textures to be accessible from vertex shaders and 16 textures to be accessible from fragment shaders at any given time, eliminating the need for multi-pass rendering: every “layer” can be processed within one pass, provided everything can be done using 16 textures.

Note that textures are specified separately for vertex shaders and fragment shaders, so a total of 32 textures can be enabled at any given time.

The following example shows how to set a given texture for use:

```
int err = SCE_OK;
unsigned int samplerUnit = 0;

// Set texture for use by vertex shader
err = sceGxmSetVertexTexture(g_context, samplerUnit, texture);
SCE_DBG_ASSERT(err == SCE_OK);

// Set texture for use by fragment shader
err = sceGxmSetFragmentTexture(g_context, samplerUnit, texture);
SCE_DBG_ASSERT(err == SCE_OK);
```

Initiating a Draw Call

On PlayStation®Vita, all draw calls are index-based. For geometry which is often created at runtime, such as particle geometry or font geometry, it is sometimes more practical to work with non-indexed geometry.

For such assets, it is possible to setup a common index array where entry N is equal to N, and use it in place of a "real" index array. The hardware will still work with indexed geometry, while the application still work as though it were non-indexed. Doing so will save you from having to duplicate this data for each non-indexed asset.

To perform a draw operation:

- Set the vertex and fragment shaders
- Set each vertex stream's base address in memory
- Performing the draw call itself, as the rest has already been setup at vertex shader patching time

```
sceGxmSetVertexProgram(g_context,       // libgxm context
    basicVertexProgram);                // Patched vertex shader
sceGxmSetFragmentProgram(g_context,     // libgxm context
    basicFragmentProgram);              // Patched fragment shader
sceGxmSetVertexStream(g_context,        // libgxm context
    0,                                  // Vtx stream index (same as
    basicVertices);                     // specified at patch time)
                                        // Pointer to the vertex stream
sceGxmDraw(g_context,                   // libgxm context
    SCE_GXM_PRIMITIVE_TRIANGLES,        // Primitive type
    SCE_GXM_INDEX_FORMAT_U16,           // Index type (16 or 32)
    indexArray,                         // Pointer to the index array
    3);                                 // Number of indices
```

Remember that draw calls must always fall between a `sceGxmBeginScene()` and `sceGxmEndScene()` pair. Failing to do so will result in an error.

Waiting for Rendering Completion

Sometimes it is necessary for the CPU to wait for all rendering to be done. This can be done easily in libgxm, by calling `sceGxmFinish()` for a given context. Be aware that doing so can be quite expensive, and has the undesirable side-effect of blocking a CPU thread until rendering on the graphics processor is complete.

If you are waiting for completion to allow for a vertex texture to be used for a subsequent scene rendering, then libgxm provides a better mechanism without the undesirable side-effect. Use the `flags` element of the `SceGxmScene` structure:

- When you initialize the scene whose color or depth buffer will subsequently be used as a vertex texture, set the flag element to `SCE_GXM_SCENE_FRAGMENT_SET_DEPENDENCY`.
- When you initialize the scene that will use the vertex texture, set the flag element to `SCE_GXM_SCENE_VERTEX_WAIT_FOR_DEPENDENCY`.

- When SGX processes scenes and encounters one with the `VERTEX_WAIT_FOR_DEPENDENCY` flag, it will wait until the *last* scene with a `FRAGMENT_SET_DEPENDENCY` flag is complete before proceeding further.

Note that no synchronization is necessary for cases where a scene's fragment shader uses the color or depth buffer of a previous scene as fragment texture.

Flipping Display Buffers

After a full frame of rendering has been submitted to libgxm through scenes, swap the display buffer to the final color surface. Assuming the display buffers are set up (see Setting up Display Buffers), there are two components to managing the display buffers:

- Initialize the *display queue*.

Do this one time, soon after initializing libgxm. The following code sets up a display queue callback function that is invoked when the scene associated with the sync point passed into `sceGxmDisplayQueueAddEntry()` is ready to be displayed.

```
// Data structure to pass data to the display queue
typedef struct DisplayData
{
    void  *address;             // Display buffer pointer
} DisplayData;

// A pointer to this function is passed in as a parameter
// to the sceGxmInit() function, as shown above
void displayCallback(const void *callbackData)
{
    int err = SCE_OK;
    SceDisplayFrameBuf framebuf;
    // Cast the parameters back
    const DisplayData *displayData = (const DisplayData *)callbackData;

    // Swap to the new buffer on the next VSYNC
    // (Last parameter could be _NEXTHSYNC if so desired)
    memset(&framebuf, 0x00, sizeof(SceDisplayFrameBuf));
    framebuf.size= sizeof(SceDisplayFrameBuf);
    framebuf.base= displayData->address;
    framebuf.pitch = DISPLAY_STRIDE_IN_PIXELS;
    framebuf.pixelformat = DISPLAY_PIXEL_FORMAT;
    framebuf.width = DISPLAY_WIDTH;
    framebuf.height = DISPLAY_HEIGHT;
    err = sceDisplaySetFrameBuf(&framebuf,
SCE_DISPLAY_UPDATETIMING_NEXTVSYNC);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Block this callback until the swap has occurred
    // and the old buffer is no longer displayed
    err = sceDisplayWaitVblankStart();
    SCE_DBG_ASSERT(err == SCE_OK);
}
```

- Queue the display flip.

This step is done once per frame, after all rendering has been submitted. The sync objects are necessary so rendering and displaying do not happen at the same time for a given buffer.

```
DisplayData displayData;
unsigned int backBufferIndex = 0;
unsigned int frontBufferIndex = 0;
```

```
// Queue the display swap for this frame
displayData.address = displayBufferData[backBufferIndex];
sceGxmDisplayQueueAddEntry(
   displayBufferSync[frontBufferIndex], // Front buffer is OLD buffer
   displayBufferSync[backBufferIndex],  // Back buffer is NEW buffer
   &displayData);                       // Ptr to data to be passed to
                                        // callback function
```

Stitching it All Together

Many of these examples come from the samples `api_libgxm/basic` and `api_libgxm/render_to_texture`, both of which are part of the PlayStation®Vita Software Development Kit (SDK). You should look at the sample `api_libgxm/basic` first because it illustrates many of the basic libgxm features, without the complications of multiple source files. The sample `api_libgxm/render_to_texture` takes basic concepts and builds on top of them with an extra color surface.

Next Steps

This document only begins to describe what is possible for programming graphics rendering on PlayStation®Vita.

As a next step, read about general Cg shader programming, and then study the various graphics overviews and references which are part of the PlayStation®Vita SDK.

At this point, performance becomes the only real bottleneck.

4 Software: Sample Walkthrough - Shadow Mapping

This section explains the aspects of the PlayStation®Vita SDK shadow mapping graphics sample:

```
/sample_code/graphics/api_libgxm/shadow_mapping
```

This sample renders cubes over a ground plane, and does so with shadows. It uses the well-documented shadow map technique.

In short, the algorithm requires the rendering of a *shadow map* for each light that generates shadows. This shadow map is a depth map of the scene rendered from the point of view of the light. After generating the shadow map or maps, the application performs scene rendering as usual, except that fragment shaders must perform the extra step of checking if each pixel is in light or in shadow. This check is done by transforming the pixel's position into light space and querying the position in the light's shadow map.

For more information describing the algorithm, see introductions found online or in many computer graphics books.

Cg Shaders

The shadow mapping sample contains three vertex and fragment shader pairs: two clear shaders, two shadow map generation shaders, and two cube-rendering shaders.

Building Shaders

The six shaders are part of the sample's Visual Studio Integration (VSI) project, and as such are compiled alongside other source files. As they cannot be compiled using the standard PlayStation®Vita C/C++ compiler, they are compiled using the PlayStation®Vita Cg compiler. The VSI project includes custom build steps to accommodate each compiler.

To set up the build, include the following in each vertex shader's properties:

```
"$(SCE_PSP2_SDK_DIR)/host_tools/bin/psp2cgc.exe" --cache --profile
sce_vp_psp2 "$(InputDir)\$(InputName).cg" -o "$(IntDir)\$(InputName).gxp"

"$(SCE_PSP2_SDK_DIR)/host_tools/build/bin/psp2bin.exe"
"$(IntDir)\$(InputName).gxp" -b2e
PSP2,_binary_$(InputName)_gxp_start,_binary_$(InputName)_gxp_size,4 -o
"$(IntDir)\$(InputName)_gxp.obj"
```

The first step uses the PlayStation®Vita Cg compiler to generate a GXP shader binary output file. The second step converts the shader binary output file to a standard (linkable) ELF object file. That second step converts the binary file as binary data which can be referenced using the following public symbols:

```
_binary_{vertexShaderFilename}_gxp_start
_binary_{vertexShaderFilename}_gxp_size
```

Fragment shaders are built similarly, using the following custom build step commands:

```
"$(SCE_PSP2_SDK_DIR)/host_tools/bin/psp2cgc.exe" --cache --profile
sce_fp_psp2 "$(InputDir)\$(InputName).cg" -o "$(IntDir)\$(InputName).gxp"

"$(SCE_PSP2_SDK_DIR)/host_tools/build/bin/psp2bin.exe"
"$(IntDir)\$(InputName).gxp" -b2e
PSP2,_binary_$(InputName)_gxp_start,_binary_$(InputName)_gxp_size,4 -o
"$(IntDir)\$(InputName)_gxp.obj"
```

Creating the following public symbols:

```
_binary_{fragmentShaderFilename}_gxp_start
_binary_{fragmentShaderFilename}_gxp_size
```

In both cases, the final output file is an object file; it is therefore linked in with the rest of the application. The application accesses the symbols by first declaring them as external in a C/C++ file:

```
extern const SceGxmProgram _binary_{vertexShaderFilename}_gxp_start;
extern const SceGxmProgram _binary_{fragmentShaderFilename}_gxp_start;
```

Clear Shaders

As their names suggest, these shaders clear the screen. This step is quite simple, and so is the shader code:

```
// Vertex shader 'clear_v.cg'
float4 main(float2 aPosition) : POSITION
{
    return float4(aPosition, 1.f, 1.f);
}
```

The vertex shader takes a two-element float array position in `aPosition`, converts it to a four-element float array of the form (x, y, 1, 1), and sends it to be interpolated by the raster stage.

The shader output has the semantic `POSITION`. This semantic must be provided in clip space, where each X/W, Y/W, and Z/W must be within the {-1, 1} range to be visible. Because this vertex shader does not transform the input position by a transformation matrix, those input positions must already be in clip space. To clear the whole screen, you would have to provide input values {-1, -1}, {-1, 1}, {1, 1}, and {1, -1}.

```
// Fragment shader 'clear_f.cg'
float4 main() : COLOR
{
    return 0.f;
}
```

The fragment shader takes no input parameters, because it is implicit and pre-determined by the time the fragment shader is executed for a given pixel. It returns a four-element float array of {0, 0, 0, 0}, which corresponds to black color with an alpha value of 0.

Shadow Map Generation Shaders

These shaders are used to generate the shadow map texture. A shadow map is a depth map, where each pixel represents how far the geometry it corresponds to is from the view point. A typical depth buffer is exactly this, and so the sample renders to a depth buffer without rendering a color buffer.

```
// Vertex shader 'shadow_v.cg'
void main(float3 aPosition,
    uniform float4x4 worldViewProj,
    out float4 vPosition : POSITION)
{
    vPosition = mul(float4(aPosition, 1.f), worldViewProj);
}
```

The vertex shader takes two inputs:

- `aPosition` is a three-element float array, describing a position in object local space.
- `worldViewProj` is a concatenated 4-by-4 matrix, performing local-to-world, world-to-screen, and screen-to-clip space transformation. Because this matrix will be the same for all vertices of a given model, it is marked as uniform, and need only be set once per model.

The only output is `vPosition`, a four-element clip-space position.

```
// Fragment shader 'shadow_f.cg'
float4 main()
{
    return 0.f;
}
```

The fragment shader is identical to the clear fragment shader, and its only output is black color. This is done in place of an empty fragment shader, as there is no way to fully disable fragment shader execution.

Cube Rendering Shaders

These shaders (one vertex, one fragment) are used to render objects like the ground and cubes.

The cube vertex shader has five input parameters: two are vertex attributes and three are uniform parameters. The two attributes are `aPosition` and `aNormal`, while the three uniform parameters are the following:

- `world` is a local-to-world 4-by-4 transformation matrix.
- `viewProj` is a world-to-clip 4-by-4 transformation matrix.
- `shadowTexViewProj` is a world-to-clip 4-by-4 transformation matrix to bring vertices into the shadow map texture space.

There are four output parameters:

- `vPosition` uses the semantic `POSITION`, which is as always the vertex position in clip space.
- `vWorldPos` is the vertex position in world space.
- `vWorldNormal` is the normal in world space.
- `vShadowCoord` is the vertex coordinate in shadow map's clip space.

`vWorldPos` and `vWorldNormal` are passed to the fragment shader to perform lighting calculations. `vShadowCoord` is needed by the fragment shader, because it queries the shadow map texture for each pixel to see if it is in light or in shadow.

To calculate `worldPos`, the shader multiplies the local-space vertex position by the local-to-world matrix. This variable is then used to calculate the final clip-space position `vPosition` and shadow-map-space position `shadowCoord`, and is output as-is to `vWorldPos`. To calculate `worldNormal`, the input normal is also transformed by the local-to-world matrix, although no translation is applied (a W component of zero is specified prior to the multiplication). It is then copied as-is to `vWorldNormal`.

```
// Vertex shader 'cube_v.cg'
void main(
    float3 aPosition,
    float3 aNormal,
    uniform float4x4 world,
    uniform float4x4 viewProj,
    uniform float4x4 shadowTexViewProj,
    out float4 vPosition : POSITION,
    out float3 vWorldPos : TEXCOORD0,
    out float3 vWorldNormal : TEXCOORD1,
    out float4 vShadowCoord : TEXCOORD2)
{
    float4 worldPos = mul(float4(aPosition, 1.f), world);
    float4 worldNormal = mul(float4(aNormal, 0.f), world);

    vPosition = mul(worldPos, viewProj);
    vWorldPos = worldPos.xyz;
    vWorldNormal = worldNormal.xyz;
    vShadowCoord = mul(worldPos, shadowTexViewProj);
}
```

Besides `vPosition`, all other vertex shader output values find their way as inputs to the fragment shader, in interpolated form. In addition, the fragment shader has three uniform input parameters:

- `shadowTex` is the shadow map texture.
- `worldLightPos` is the world-space position for the light that was used to generate the shadow map.
- `diffuseColor` is the object's diffuse color.

The fragment shader's only output parameter is the four-element float array representing the final pixel color.

In this shader, the lighting equation is simplified to only take into account attenuation based on "normal" in relation to the lighting direction, and it uses a hard-coded dark-gray ambient light of color {0.2, 0.2, 0.2}. It calculates the object's surface normal at the pixel position and performs a dot product with the inverse of the light direction (`lightDir`), which itself is calculated by subtracting the pixel position's world position from the light's world position. The dot product result is assigned to the variable `nDotL`.

`shadowTerm` is the value that determines whether the pixel is in shadow or not. The effort to calculate it occurs within the `f1tex2Dproj()` function call. As described in renderMain Function, because the texture is set as type "RF32M", the hardware returns either 0 or 1, based on whether the texture coordinate passed in has a depth value that is higher or lower than the value fetched from the shadow map texture. For example, let's say the application fetches the texture with the texture coordinate {0.2, 0.5, 0.7, 1}. If the depth value at {0.2, 0.5} is less than 0.7, then `shadowTerm` is assigned a value of 0. If the depth value is greater than or equal to 0.7, then it is assigned a value of 1.

`litValue` brings all of the lighting calculations together: it gives the normal-based attenuation a weight of 80% and gives the remaining 20% to the ambient light. Finally, `litValue` is multiplied by the light's diffuse color and returned as pixel output.

Note that some of the fragment shader's local variables are defined as `half`, whereas others are `float`. Under some circumstances, using `half` variables will result in higher performance, because it uses less resources and often uses specialized instructions. But it can sometimes result in worse performance, because casting between `float` and `half` is not always free. Refer to the *GPU User's Guide* for additional details.

```
// Fragment shader 'cube_f.cg'
float4 main(
    float3 vWorldPos : TEXCOORD0,
    float3 vWorldNormal : TEXCOORD1,
    float4 vShadowCoord : TEXCOORD2,
    uniform sampler2D shadowTex : TEXUNIT0,
    uniform float3 worldLightPos,
    uniform half3 diffuseColor)
{
    // Compute directions
    float3 normal = normalize(vWorldNormal);
    float3 lightDir = normalize(worldLightPos - vWorldPos);

    // Compute shadow term
    float shadowTerm = f1tex2Dproj<float>(shadowTex, vShadowCoord);

    // Diffuse lighting
    half nDotL = max(dot(normal, lightDir), 0.0f);
    half litValue = 0.8f*nDotL*shadowTerm + 0.2f;
    return float4(litValue*diffuseColor, 1.f);
}
```

Main Application Code

The main application code is spread across three source files:

- `common.cpp` contains code that is shared among many libgxm samples.
- `heap.cpp` contains heap-management code that is shared among many libgxm samples.
- `main.cpp` contains sample-specific code.

common.cpp Source File

The file `common.cpp` contains functionality that is shared among many libgxm samples. Namely, functions related to debugging fonts, basic libgxm initialization, and graphics-related memory allocation.

Note that any file using functions from `common.cpp` must include the header file `common.h`.

initDbgFont / termDbgFont Functions

The functions `initDbgFont()` and `termDbgFont()` initialize and terminate the debug font library. Debug font is quite useful for quickly and simply printing out information on the screen.

```
int initDbgFont(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Initialize structure
    SceDbgFontConfig config;
    memset(&config, 0, sizeof(SceDbgFontConfig));
    config.fontsize = SCE_DBGFONT_FONTSIZE_DEFAULT;
    err = sceDbgFontInit(&config);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Done
    return err;
}

int termDbgFont(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Exit
    err = sceDbgFontExit();
    SCE_DBG_ASSERT(err == SCE_OK);

    // Done
    return err;
}
```

createGxmContext Function

The function `createGxmContext()` creates a context to be used throughout the application for adding rendering commands, as outlined in Initializing libgxm.

```
// libgxm context
void *g_contextHostMem = NULL;
void *g_contextVdmMem = NULL;
void *g_contextVertexMem = NULL;
void *g_contextFragmentMem = NULL;
void *g_contextFragmentUsseMem = NULL;
SceGxmContext *g_context = NULL;

static void createGxmContext(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Allocate host memory
    g_contextHostMem = malloc(SCE_GXM_MINIMUM_CONTEXT_HOST_MEM_SIZE);

    // Allocate ring buffer memory using default sizes
    uint32_t fragmentUsseOffset;
    g_contextVdmMem = heapAlloc(
        HEAP_TYPE_LPDDR_R,
        SCE_GXM_DEFAULT_VDM_RING_BUFFER_SIZE,
        4);
    g_contextVertexMem = heapAlloc(
        HEAP_TYPE_LPDDR_R,
```

```
        SCE_GXM_DEFAULT_VERTEX_RING_BUFFER_SIZE,
        4);
    g_contextFragmentMem = heapAlloc(
        HEAP_TYPE_LPDDR_R,
        SCE_GXM_DEFAULT_FRAGMENT_RING_BUFFER_SIZE,
        4);
    g_contextFragmentUsseMem = heapAlloc(
        HEAP_TYPE_FRAGMENT_USSE,
        SCE_GXM_DEFAULT_FRAGMENT_USSE_RING_BUFFER_SIZE,
        4,
        &fragmentUsseOffset);

    // Set up parameters
    SceGxmContextParams contextParams;
    memset(&contextParams, 0, sizeof(SceGxmContextParams));
    contextParams.hostMem = g_contextHostMem;
    contextParams.hostMemSize = SCE_GXM_MINIMUM_CONTEXT_HOST_MEM_SIZE;
    contextParams.vdmRingBufferMem = g_contextVdmMem;
    contextParams.vdmRingBufferMemSize =
        SCE_GXM_DEFAULT_VDM_RING_BUFFER_SIZE;
    contextParams.vertexRingBufferMem = g_contextVertexMem;
    contextParams.vertexRingBufferMemSize =
        SCE_GXM_DEFAULT_VERTEX_RING_BUFFER_SIZE;
    contextParams.fragmentRingBufferMem = g_contextFragmentMem;
    contextParams.fragmentRingBufferMemSize =
        SCE_GXM_DEFAULT_FRAGMENT_RING_BUFFER_SIZE;
    contextParams.fragmentUsseRingBufferMem = g_contextFragmentUsseMem;
    contextParams.fragmentUsseRingBufferMemSize =
        SCE_GXM_DEFAULT_FRAGMENT_USSE_RING_BUFFER_SIZE;
    contextParams.fragmentUsseRingBufferOffset = fragmentUsseOffset;

    // Create the context
    err = sceGxmCreateContext(&contextParams, &g_context);
    SCE_DBG_ASSERT(err == SCE_OK);
}
```

createGxmShaderPatcher Function

The function `createGxmShaderPatcher()`, as its name implies, creates a shader patcher as outlined in Patching a Vertex and Fragment Shader. It will be executed once, and that patcher will be reused multiple times.

```
// libgxm shader patcher
SceGxmShaderPatcher *g_shaderPatcher = NULL;

static void createGxmShaderPatcher(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Create a shader patcher
    SceGxmShaderPatcherParams patcherParams;
    memset(&patcherParams, 0, sizeof(SceGxmShaderPatcherParams));
    patcherParams.userData = NULL;
    patcherParams.hostAllocCallback = &patcherHostAlloc;
    patcherParams.hostFreeCallback = &patcherHostFree;
    patcherParams.bufferAllocCallback = &patcherBufferAlloc;
    patcherParams.bufferFreeCallback = &patcherFree;
    patcherParams.bufferMem = NULL;
    patcherParams.bufferMemSize = NULL;
    patcherParams.vertexUsseAllocCallback = &patcherVertexUsseAlloc;
    patcherParams.vertexUsseFreeCallback = &patcherFree;
```

```
    patcherParams.vertexUsseMem = NULL;
    patcherParams.vertexUsseMemSize = NULL;
    patcherParams.vertexUsseOffset = NULL;
    patcherParams.fragmentUsseAllocCallback = &patcherFragmentUsseAlloc;
    patcherParams.fragmentUsseFreeCallback = &patcherFree;
    patcherParams.fragmentUsseMem = NULL;
    patcherParams.fragmentUsseMemSize = NULL;
    patcherParams.fragmentUsseOffset = NULL;

    err = sceGxmShaderPatcherCreate(&patcherParams, &g_shaderPatcher);
    SCE_DBG_ASSERT(err == SCE_OK);
}
```

createRenderTarget and destroyRenderTarget Functions

The function `createRenderTarget()` creates the render target structure, which is necessary because it defines how surfaces are handled for fragment processing. This step was described in Beginning and Ending a Scene. Inversely, `destroyRenderTarget()` deletes the structure.

```
SceGxmRenderTarget *createRenderTarget(uint32_t width, uint32_t height,
    SceGxmMultisampleMode msaaMode)
{
    int err = SCE_OK;
    UNUSED(err);

    // Set up parameters
    SceGxmRenderTargetParams params;
    memset(&params, 0, sizeof(SceGxmRenderTargetParams));
    params.flags = 0;
    params.width = width;
    params.height = height;
    params.scenesPerFrame = 1;
    params.multisampleMode = msaaMode;
    params.multisampleLocations = 0;
    params.driverMemBlock = SCE_UID_INVALID_UID;

    // Create the render target
    SceGxmRenderTarget *renderTarget;
    err = sceGxmCreateRenderTarget(&params, &renderTarget);
    SCE_DBG_ASSERT(err == SCE_OK);
    return renderTarget;
}

void destroyRenderTarget(SceGxmRenderTarget *renderTarget)
{
    int err = SCE_OK;
    UNUSED(err);

    // Destroy the render target
    err = sceGxmDestroyRenderTarget(renderTarget);
    SCE_DBG_ASSERT(err == SCE_OK);
}
```

initGxmLibrary and initGxm Functions

The function `initGxmLibrary()` specifically initializes the gxm library, as outlined in Initializing libgxm.

```
// Data structure to pass through the display queue
typedef struct DisplayData
{
    void *address;                  // Framebuffer address
    uint32_t width;                 // Framebuffer width
```

```
    uint32_t height;                // Framebuffer height
    uint32_t strideInPixels;        // Framebuffer stride in pixels
    uint32_t flipMode;              // From #FlipMode
} DisplayData;

static void initGxmLibrary(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Set up parameters
    SceGxmInitializeParams initializeParams;
    memset(&initializeParams, 0, sizeof(SceGxmInitializeParams));
    initializeParams.flags = 0;
    initializeParams.displayQueueMaxPendingCount =
        DISPLAY_MAX_PENDING_SWAPS;
    initializeParams.displayQueueCallback = displayCallback;
    initializeParams.displayQueueCallbackDataSize = sizeof(DisplayData);
    initializeParams.parameterBufferSize =
        SCE_GXM_DEFAULT_PARAMETER_BUFFER_SIZE;

    // Start libgxm
    err = sceGxmInitialize(&initializeParams);
    SCE_DBG_ASSERT(err == SCE_OK);
}
```

The function `initGxm()` calls the three initialization functions `initGxmLibrary()`, `createGxmContext()`, and `createGxmShaderPatcher()` and initializes the remainder of the basic libgxm features. The sample includes four initialization steps, described here:

The first step in the function is to call the three initialization functions.

```
// libgxm display queue
void *g_displayBufferData[DISPLAY_BUFFER_COUNT];
SceGxmColorSurface g_displaySurface[DISPLAY_BUFFER_COUNT];
SceGxmSyncObject *g_displayBufferSync[DISPLAY_BUFFER_COUNT];
uint32_t g_displayFrontBufferIndex = DISPLAY_BUFFER_COUNT - 1;
uint32_t g_displayBackBufferIndex = 0;

// Depth buffer for display surface
void g_mainDepthBufferData = NULL;
void g_mainStencilBufferData = NULL;
SceGxmDepthStencilSurface g_mainDepthSurface;

// libgxm main render target
SceGxmRenderTarget *g_mainRenderTarget = NULL;

int initGxm(SceGxmDepthStencilFormat depthFormat, SceGxmMultiSampleMode
    msaaMode)
{
    int err = SCE_OK;
    UNUSED(err);

    // Initialize libgxm
    initGxmLibrary();

    // Initialize our sample heaps
    initHeaps();

    // Create a rendering context
    createGxmContext();
```

```
    // Create a shader patcher
    createGxmShaderPatcher();
```

`initGxm()` follows with the creation and initialization of display buffers and depth buffer as described in Setting up Color and Depth Surfaces. As described, it uses a triple-buffered display buffer scheme.

```
// Allocate memory and sync objects for display buffers
for (uint32_t i = 0; i < DISPLAY_BUFFER_COUNT; ++i) {
    // Allocate memory for display
    g_displayBufferData[i] = heapAlloc(
    HEAP_TYPE_CDRAM_RW,
    4*DISPLAY_STRIDE_IN_PIXELS*DISPLAY_HEIGHT),
    256);

    // memset the buffer to black
    for (uint32_t y = 0; y < DISPLAY_HEIGHT; ++y) {
    uint32_t *row = (uint32_t *)g_displayBufferData[i] +
       y*DISPLAY_STRIDE_IN_PIXELS;
          for (uint32_t x = 0; x < DISPLAY_WIDTH; ++x) {
          row[x] = 0xff000000;
          }
       }

    // Initialize a color surface for this display buffer
    err = sceGxmColorSurfaceInit(
    &g_displaySurface[i],
    DISPLAY_COLOR_FORMAT,
    SCE_GXM_COLOR_SURFACE_LINEAR,
       (msaaMode == SCE_GXM_MULTISAMPLE_NONE)
       ? SCE_GXM_COLOR_SURFACE_SCALE_NONE
       : SCE_GXM_COLOR_SURFACE_SCALE_MSAA_DOWNSCALE,
    SCE_GXM_OUTPUT_REGISTER_SIZE_32BIT,
    DISPLAY_WIDTH,
    DISPLAY_HEIGHT,
    DISPLAY_STRIDE_IN_PIXELS,
    g_displayBufferData[i]);
       SCE_DBG_ASSERT(err == SCE_OK);

    // Create a sync object that we will associate with this buffer
    err = sceGxmSyncObjectCreate(&g_displayBufferSync[i]);
    SCE_DBG_ASSERT(err == SCE_OK);
    }

    // Compute depth buffer dimensions
    const uint32_t alignedWidth = ALIGN(DISPLAY_WIDTH,
       SCE_GXM_TILE_SIZEX);
    const uint32_t alignedHeight = ALIGN(DISPLAY_HEIGHT,
       SCE_GXM_TILE_SIZEY);
    uint32_t sampleCount = alignedWidth*alignedHeight;
    uint32_t depthStrideInSamples = alignedWidth;
    if (msaaMode == SCE_GXM_MULTISAMPLE_4X) {
       // Samples increase in X and Y
       sampleCount *= 4;
       depthStrideInSamples *= 2;
    } else if (msaaMode == SCE_GXM_MULTISAMPLE_2X) {
       // Samples increase in Y only
       sampleCount *= 2;
    }

    // Compute depth buffer bytes per sample
    uint32_t depthBytesPerSample = 0;
```

```
    uint32_t stencilBytesPerSample = 0;
    switch (depthFormat)
    {
    case SCE_GXM_DEPTH_STENCIL_FORMAT_DF32:
    case SCE_GXM_DEPTH_STENCIL_FORMAT_DF32M:
       depthBytesPerSample = 4;
       break;

    case SCE_GXM_DEPTH_STENCIL_FORMAT_S8:
       stencilBytesPerSample = 1;
       break;

    case SCE_GXM_DEPTH_STENCIL_FORMAT_DF32_S8:
    case SCE_GXM_DEPTH_STENCIL_FORMAT_DF32M_S8:
       depthBytesPerSample = 4;
       stencilBytesPerSample = 1;
       break;

    case SCE_GXM_DEPTH_STENCIL_FORMAT_S8D24:
       depthBytesPerSample = 4;
       break;

    case SCE_GXM_DEPTH_STENCIL_FORMAT_D16:
       depthBytesPerSample = 2;
       break;
    }

    // Allocate memory
    if (depthBytesPerSample != 0) {
       g_mainDepthBufferData = heapAlloc(
     HEAP_TYPE_LPDDR_RW,
     depthBytesPerSample*sampleCount,
     SCE_GXM_DEPTHSTENCIL_SURFACE_ALIGNMENT);
    }
    If (stencilBytesPerSample != 0) {
       g_mainStencilBufferData = heapAlloc(
          HEAP_TYPE_LPDDR_RW,
          stencilBytesPerSample*sampleCount,

SCE_GXM_DEPTHSTENCIL_SURFACE_ALIGNMENT);

    // Initialize depth surface
    err = sceGxmDepthStencilSurfaceInit(
     &g_mainDepthSurface,
     depthFormat,
     SCE_GXM_DEPTH_STENCIL_SURFACE_TILED,
     depthStrideInSamples,
     g_mainDepthBufferData,
     g_mainStencilBufferData);
```

The third step performs the first display-buffer flip synchronized to the next VSync. It is done to avoid visual artifacts that could be visible between initialization time and the time the first frame is done.

```
    // Swap to the current front buffer with VSYNC
    // (also ensures that future calls with HSYNC are successful)
    SceDisplayFrameBuf framebuf;
    memset(&framebuf, 0x00, sizeof(SceDisplayFrameBuf));
    framebuf.size = sizeof(SceDisplayFrameBuf);
    framebuf.base = g_displayBufferData[g_displayFrontBufferIndex];
    framebuf.pitch = DISPLAY_STRIDE_IN_PIXELS;
    framebuf.pixelformat = DISPLAY_PIXEL_FORMAT;
    framebuf.width = DISPLAY_WIDTH;
```

```
    framebuf.height = DISPLAY_HEIGHT;
    err = sceDisplaySetFrameBuf(&framebuf,
        SCE_DISPLAY_UPDATETIMING_NEXTVSYNC);
    SCE_DBG_ASSERT(err == SCE_OK);
    err = sceDisplayWaitSetFrameBuf();
    SCE_DBG_ASSERT(err == SCE_OK);
```

The fourth and final step creates a render target that is used to initiate rendering to the main scene. As noted later, rendering to the shadow map texture uses a separate render target.

```
// Create a render target that describes the tiling setup we want to use
g_mainRenderTarget = createRenderTarget(DISPLAY_WIDTH, DISPLAY_HEIGHT,
     msaaMode);

// Done
    return err;
}
```

After `initGxm()` completes, libgxm is ready to be used for general rendering.

termGxm() Function

The `termGxm()` function un-initializes libgxm, bringing the system back to a state as though libgxm had never been initialized in the first place.

It performs steps similar to `initGxm()`, in reverse order: it destroys the render target, then display buffers, display queue, shader patcher, rendering context, and finally un-initializes libgxm itself.

```
int termGxm(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Destroy render target
    destroyRenderTarget(g_mainRenderTarget);

    // Destroy depth buffer
    heapFree(g_mainStencilBufferData);
    heapFree(g_mainDepthBufferData);

    // Wait for display processing to finish before deallocating buffers
    err = sceGxmDisplayQueueFinish();
    SCE_DBG_ASSERT(err == SCE_OK);

    // Free the display buffers and sync objects
    for (uint32_t i = 0; i < DISPLAY_BUFFER_COUNT; ++i) {
        // Clear the buffer and deallocate it
        heapFree(g_displayBufferData[i]);

        // Destroy sync object
        err = sceGxmSyncObjectDestroy(g_displayBufferSync[i]);
        SCE_DBG_ASSERT(err == SCE_OK);
    }

    // Destroy the shader patcher
    err = sceGxmShaderPatcherDestroy(g_shaderPatcher);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Destroy the rendering context
    err = sceGxmDestroyContext(g_context);
    SCE_DBG_ASSERT(err == SCE_OK);
    heapFree(g_contextFragmentUsseMem);
    heapFree(g_contextFragmentMem);
    heapFree(g_contextVertexMem);
```

```
    heapFree(g_contextVdmMem);
    free(g_contextHostMem);

    // Destroy heaps
    heapTerminate();
    sceKernelFreeMemBlock(g_lpddrUid);
    sceKernelFreeMemBlock(g_cdramUid);

    // Terminate libgxm
    err = sceGxmTerminate();
    SCE_DBG_ASSERT(err == SCE_OK);

    // Done
    return err;
}
```

displayCallback Function

The function `displayCallback()` is called every time a display buffer is ready to be flipped. Flipping Display Buffers describes a variation of this function, but here it performs a bit more work.

The sample makes use of debug font: this function is a good place to invoke the final font rendering because rendering of everything else is complete. This is done by calling `sceDbgFontFlush()`, and passing in information about the frame buffer in which the font will be displayed.

```
static void displayCallback(const void *callbackData)
{
    int err = SCE_OK;
    UNUSED(err);

    // Cast the parameters back
    const DisplayData *displayData = (const DisplayData *)callbackData;

    // Render debug text now GPU rendering has finished
    renderDbgFont();


    // Flush debug text
    SceDbgFontFrameBufInfo info;
    memset(&info, 0, sizeof(info));
    info.framebuf_addr = (SceUChar8 *)displayData->address;
    info.framebuf_pitch = displayData->strideInPixels;
    info.framebuf_pixelformat = DISPLAY_DBGFONT_FORMAT;
    info.framebuf_width = displayData->width;
    info.framebuf_height = displayData->height;
    err = sceDbgFontFlush(&info);
    SCE_DBG_ASSERT(err == SCE_OK);
```

The sample also waits for an extra VSync if necessary, as is the case if the application chooses to run at a fixed frame rate of 30fps instead of 60fps. This is done by calling `sceDisplayWaitSetFrameBufMulti()`.

```
    // Check this buffer has been displayed for the necessary number of VSYNCs
    // (Avoids queuing a flip before the second VSYNC has happened)
    if (displayData->flipMode == FLIP_MODE_VSYNC_2) {
        err = sceDisplayWaitSetFrameBufMulti(2);
    }

    // Swap to the new buffer
    SceDisplayFrameBuf framebuf;
    memset(&framebuf, 0x00, sizeof(SceDisplayFrameBuf));
    framebuf.size = sizeof(SceDisplayFrameBuf);
    framebuf.base = displayData->address;
```

```
    framebuf.pitch = displayData->strideInPixels;
    framebuf.pixelformat = DISPLAY_PIXEL_FORMAT;
    framebuf.width = displayData->width;
    framebuf.height = displayData->height;
    err = sceDisplaySetFrameBuf(&framebuf,
        (displayData->flipMode == FLIP_MODE_HSYNC)
        ? SCE_DISPLAY_UPDATETIMING_NEXTHSYNC
        : SCE_DISPLAY_UPDATETIMING_NEXTVSYNC);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Block this callback until the swap has occurred and the old buffer
    // is no longer displayed
    if (displayData->flipMode != FLIP_MODE_HSYNC) {
        err = sceDisplayWaitSetFrameBuf();
        SCE_DBG_ASSERT(err == SCE_OK);
        }
    }
```

cycleDisplayBuffers Function

The function `cycleDisplayBuffers()` must be called after a frame is fully rendered and is ready for display. Such functionality is sometimes called flipping buffers.

This function incorporates functionality outlined in the second part of Flipping Display Buffers.

The function finishes by updating `g_displayFrontBufferIndex` and `g_displayBackBufferIndex` variables, so that applications cycle through every display buffer.

```
int cycleDisplayBuffers(FlipMode flipMode, uint32_t width, uint32_t
    height, uint32_t strideInPixels)
{
    int err = SCE_OK;
    UNUSED(err);

    // Queue the display swap for this frame
    DisplayData displayData;
    displayData.address = g_displayBufferData[g_displayBackBufferIndex];
    displayData.width = width;
    displayData.height = height;
    displayData.strideInPixels = strideInPixels;
    displayData.flipMode = flipMode;
    err = sceGxmDisplayQueueAddEntry(
        g_displayBufferSync[g_displayFrontBufferIndex], // Front buffer is
        g_displayBufferSync[g_displayBackBufferIndex],  // OLD buffer
        &displayData);                                  // Back buffer is
                                                        // NEW buffer
    SCE_DBG_ASSERT(err == SCE_OK);

    // Update buffer indices
    g_displayFrontBufferIndex = g_displayBackBufferIndex;
    g_displayBackBufferIndex = (g_displayBackBufferIndex + 1) %
        DISPLAY_BUFFER_COUNT;

    // Done
    return err;
}
```

patcherBufferAlloc/patcherVertexUsseAlloc/patcherFragmentUsseAlloc and patcherFree Functions

This set of four helper functions are there to allocate and free memory specifically destined to contain vertex shader data, fragment shader data, or both combined. Internally, they simply call `heapAlloc` with appropriate parameters for the function. Below is the code for vertex USSE allocation, but very similar code can be found in the other two flavors. The function `patcherFree()` frees memory for all cases.

```
void *patcherVertexUsseAlloc(void *userData, uint32_t size, uint32_t
*usseOffset)
{
    UNUSED(userData);
    Return heapAlloc(HEAP_TYPE_VERTEX_USSE, size, SCE_GXM_USSE_ALIGNMENT,
        usseOffset);
}

void patcherFree(void *userData, void *mem)
{
    UNUSED(userData);
    heapFree(mem);
}
```

patcherHostAlloc / patcherHostFree Functions

The last two functions, `patcherHostAlloc()` and `patcherHostFree()`, are callback functions used by the shader patcher for allocating and freeing main memory. They can be used to allocate and free memory from a user-defined heap, but here they simply defer memory management to the standard heap by respectively calling `malloc()` and `free()`.

```
static void *patcherHostAlloc(void *userData, void *mem)
{
    UNUSED(userData);
    return malloc(size);
}

void patcherHostFree(voi *userData, void *mem)
{
    UNUSED(userData);
    free(mem);
}
```

heap.cpp Source File

The file `heap.cpp` contains heap-management functionality shared with most graphics samples. This file offers the following functions:

```
void heapInitialize();
void heapTerminate();
void heapExtend(int32_t type, void *base, uint32_t size, uint32_t offset);
void *heapAlloc(int32_t type, uint32_t size, uint32_t alignment, uint32_t
    *offset);
void heapFree(void *addr);
```

Because heap management is a fairly known process, and because its inner-workings are not directly relevant to this document, code snippets from this source file are not provided here. Refer to the source-file content for more information.

main.cpp Source File

The file `main.cpp` contains sample-specific functionality typically not shared with other samples.

renderDbgFont Function

The function `renderDbgFont()` prints simple information on the screen, such as sample name:

```
int renderDbgFont(void)
{
    // Add some strings
    sceDbgFontPrint(30,  30, 0x8000e000, (const SceChar8 *)"libgxm sample:
        shadow mapping");
    return SCE_OK;
}
```

Note that this information does not find its way to the draw buffer until after a flush is done using `sceDbgFontFlush()`. In this sample, the flush is performed by the `displayCallback()` function found in `common.cpp`, so the sample itself doesn't have to take care of it.

createClearData / destroyClearData Functions

The functions `createClearData()` and `destroyClearData()` create and delete clear-related data, including the vertex and fragment shaders used to clear the screen and the geometry and indices used in the process. The sample includes six clear-related steps, described here.

Step one registers both the vertex and fragment shaders with the shader patcher. This is a necessary step before either shader can be instantiated for actual rendering use. As described in Building Shaders, shaders are compiled as part of the VSI project and are made available to the application via external variables. The two clear shaders are defined as the two variables `_binary_clear_v_gxp_start` and `_binary_clear_f_gxp_start`, which can be found at the top of the file.

```
extern const SceGxmProgram _binary_clear_v_gxp_start;
extern const SceGxmProgram _binary_clear_f_gxp_start;

// Data structure for clear geometry
typedef struct ClearVertex {
    float x;
    float y;
} ClearVertex;

// Clear geometry data
SceGxmShaderPatcherId g_clearVertexProgramId;
SceGxmShaderPatcherId g_clearFragmentProgramId;
SceGxmVertexProgram *g_clearVertexProgram = NULL;
SceGxmFragmentProgram *g_clearFragmentProgram = NULL;
SceUID g_clearVerticesUid;
SceUID g_clearIndicesUid;
ClearVertex *g_clearVertices = NULL;
uint16_t *g_clearIndices = NULL;

static void createClearData(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Register programs with the shader patcher
    err = sceGxmShaderPatcherRegisterProgram(g_shaderPatcher,
        &_binary_clear_v_gxp_start,&g_clearVertexProgramId);
    SCE_DBG_ASSERT(err == SCE_OK);
    err = sceGxmShaderPatcherRegisterProgram(g_shaderPatcher,
        &_binary_clear_f_gxp_start,&g_clearFragmentProgramId);
    SCE_DBG_ASSERT(err == SCE_OK);
```

Step two queries the vertex shader's input variable by name, `aPosition`.

```
// Get vertex attributes by name to bind vertex format
const SceGxmProgram *clearVertexProgram =

sceGxmShaderPatcherGetProgramFromId(g_clearVertexProgramId);
const SceGxmProgramParameter *paramPositionAttribute =

sceGxmProgramFindParameterByName(clearVertexProgram, "aPosition");
SCE_DBG_ASSERT(paramPositionAttribute &&
    (sceGxmProgramParameterGetCategory(paramPositionAttribute) ==
       SCE_GXM_PARAMETER_CATEGORY_ATTRIBUTE));
```

This allows step three to build a single vertex stream containing a single vertex attribute (position), and to link that attribute with the shader's input variable.

```
// Create clear vertex format
SceGxmVertexAttribute clearVertexAttributes[1];
SceGxmVertexStream clearVertexStreams[1];
clearVertexAttributes[0].streamIndex = 0;
clearVertexAttributes[0].offset = 0;
clearVertexAttributes[0].format = SCE_GXM_ATTRIBUTE_FORMAT_F32;
clearVertexAttributes[0].componentCount = 2;
clearVertexAttributes[0].regIndex =
    sceGxmProgramParameterGetResourceIndex(paramPositionAttribute);
clearVertexStreams[0].stride = sizeof(ClearVertex);
clearVertexStreams[0].indexSource = SCE_GXM_INDEX_SOURCE_INDEX_16BIT;
```

Step four instantiates the clear vertex and fragment shaders. To do so, a vertex stream and vertex attributes description must be provided, to allow the shader to fetch attributes. Step three built the needed information, and so `clearVertexAttributes` and `clearVertexStreams` variables are used.

```
// Create clear programs
err = sceGxmShaderPatcherCreateVertexProgram(
    g_shaderPatcher,
    g_clearVertexProgramId,
    clearVertexAttributes,
    1,
    clearVertexStreams,
    1,
    &g_clearVertexProgram);
SCE_DBG_ASSERT(err == SCE_OK);
err = sceGxmShaderPatcherCreateFragmentProgram(
    g_shaderPatcher,
    g_clearFragmentProgramId,
    SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4,
    MSAA_MODE,
    NULL,
sceGxmShaderPatcherGetProgramFromId(g_clearVertexProgramId),
   &g_clearFragmentProgram);
SCE_DBG_ASSERT(err == SCE_OK);
```

Step five allocates memory for both the vertex stream and vertex index array, using `graphicsAlloc()` helper function from `common.cpp`. Enough space for three vertices is allocated to clear the screen by drawing a triangle big enough to encompass the whole screen. The memory is requested with `SCE_GXM_MEMORY_ATTRIB_READ` only, because the SGX will only ever read that data, without ever writing it. `SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE` is used to guarantee that the data written will be fully committed to memory, and therefore made available correctly to the SGX.

```
    // Allocate vertices and indices
    g_clearVertices = (ClearVertex *)graphicsAlloc(
        SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
        3*sizeof(ClearVertex),
        4,
        SCE_GXM_MEMORY_ATTRIB_READ,
    &g_clearVerticesUid);
    g_clearIndices = (uint16_t *)graphicsAlloc(
        SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
        3*sizeof(uint16_t),
        2,
        SCE_GXM_MEMORY_ATTRIB_READ,
        &g_clearIndicesUid);
```

Step six populates of the vertex attribute array and vertex index array. Keep in mind that the vertex coordinates used are in clip space; for the triangle to cover the whole screen, coordinates {-1, 1}, {3, 1}, and {-1, -3} are used.

```
    // Write vertex data
    g_clearVertices[0].x = -1.0f;
    g_clearVertices[0].y =  1.0f;
    g_clearVertices[1].x =  3.0f;
    g_clearVertices[1].y =  1.0f;
    g_clearVertices[2].x = -1.0f;
    g_clearVertices[2].y = -3.0f;

    // Write index data
    g_clearIndices[0] = 0;
    g_clearIndices[1] = 1;
    g_clearIndices[2] = 2;
}

static void destroyClearData(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Release the shaders
    err = sceGxmShaderPatcherReleaseFragmentProgram(g_shaderPatcher,
        g_clearFragmentProgram);
    SCE_DBG_ASSERT(err == SCE_OK);
    err = sceGxmShaderPatcherReleaseVertexProgram(g_shaderPatcher,
        g_clearVertexProgram);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Free the memory used for vertices and indices
    graphicsFree(g_clearIndicesUid);
    graphicsFree(g_clearVerticesUid);

    // Unregister programs since we don't need them any more
    err = sceGxmShaderPatcherUnregisterProgram(g_shaderPatcher,
        g_clearFragmentProgramId);
    SCE_DBG_ASSERT(err == SCE_OK);
    err = sceGxmShaderPatcherUnregisterProgram(g_shaderPatcher,
        g_clearVertexProgramId);
    SCE_DBG_ASSERT(err == SCE_OK);
}
```

The `destroyClearData()` function deletes the two instanced clear shaders, frees up vertex attributes array and vertex index array using the helper function `graphicsFree()`, and unregisters the two clear shaders.

createCubeData / destroyCubeData Functions

The two functions `createCubeData()` and `destroyCubeData()` are constructed in essentially the same way as `createClearData()` and `destroyClearData()`, though they also handle shadow-map generation shaders. This makes sense because these shader instances need to know about cube vertex attributes and streams as well.

Note that in this sample, the floor is actually a flattened cube. Otherwise, there is nothing special about it.

```
// Cube geometry data
SceGxmShaderPatcherId g_cubeVertexProgramId;
SceGxmShaderPatcherId g_cubeFragmentProgramId;
SceGxmShaderPatcherId g_shadowVertexProgramId;
SceGxmShaderPatcherId g_shadowFragmentProgramId;
SceGxmVertexProgram *g_cubeVertexProgram = NULL;
SceGxmFragmentProgram *g_cubeFragmentProgram = NULL;
SceGxmVertexProgram *g_shadowVertexProgram = NULL;
SceGxmFragmentProgram *g_shadowFragmentProgram = NULL;
SceUID g_cubeVerticesUid;
SceUID g_cubeIndicesUid;
BasicVertex *g_cubeVertices = NULL;
uint16_t *g_cubeIndices = NULL;
const SceGxmProgramParameter *g_cubeWorldParam = NULL;
const SceGxmProgramParameter *g_cubeViewProjParam = NULL;
const SceGxmProgramParameter *g_cubeShadowTexViewProjParam = NULL;
const SceGxmProgramParameter *g_cubeWorldLightPosParam = NULL;
const SceGxmProgramParameter *g_cubeDiffuseColorParam = NULL;
const SceGxmProgramParameter *g_shadowWorldViewProjParam = NULL;

static void createCubeData(void)
{
    int err = SCE_OK;
    UNUSED(err);
```

This is where the sample includes the shadow map generation shaders.

```
// Register programs with the patcher
err = sceGxmShaderPatcherRegisterProgram(g_shaderPatcher,
    &_binary_cube_v_gxp_start, &g_cubeVertexProgramId);
SCE_DBG_ASSERT(err == SCE_OK);
err = sceGxmShaderPatcherRegisterProgram(g_shaderPatcher,
    &_binary_cube_f_gxp_start, &g_cubeFragmentProgramId);
SCE_DBG_ASSERT(err == SCE_OK);
err = sceGxmShaderPatcherRegisterProgram(g_shaderPatcher,
    &_binary_shadow_v_gxp_start, &g_shadowVertexProgramId);
SCE_DBG_ASSERT(err == SCE_OK);
err = sceGxmShaderPatcherRegisterProgram(g_shaderPatcher,
    &_binary_shadow_f_gxp_start,

&g_shadowFragmentProgramId);
SCE_DBG_ASSERT(err == SCE_OK);

// Find vertex uniforms by name and cache parameter info
// Note: name lookup is a slow load-time operation
const SceGxmProgram *cubeVertexProgram =
    sceGxmShaderPatcherGetProgramFromId(g_cubeVertexProgramId);
SCE_DBG_ASSERT(cubeVertexProgram);
g_cubeWorldParam = sceGxmProgramFindParameterByName(
    cubeVertexProgram, "world");
SCE_DBG_ASSERT(g_cubeWorldParam && (sceGxmProgramParameterGetCategory(
g_cubeWorldParam) == SCE_GXM_PARAMETER_CATEGORY_UNIFORM));
```

```
g_cubeViewProjParam = sceGxmProgramFindParameterByName(
    cubeVertexProgram, "viewProj");
SCE_DBG_ASSERT(g_cubeViewProjParam && (sceGxmProgramParameterGetCategory(
g_cubeViewProjParam) == SCE_GXM_PARAMETER_CATEGORY_UNIFORM));
g_cubeShadowTexViewProjParam = sceGxmProgramFindParameterByName(
    cubeVertexProgram, "shadowTexViewProj");
SCE_DBG_ASSERT(g_cubeShadowTexViewProjParam &&
    (sceGxmProgramParameterGetCategory(g_cubeShadowTexViewProjParam) ==
       SCE_GXM_PARAMETER_CATEGORY_UNIFORM));

const SceGxmProgram *shadowVertexProgram =
    sceGxmShaderPatcherGetProgramFromId(g_shadowVertexProgramId);
SCE_DBG_ASSERT(shadowVertexProgram);
g_shadowWorldViewProjParam = sceGxmProgramFindParameterByName(
    shadowVertexProgram,"worldViewProj");
SCE_DBG_ASSERT(g_shadowWorldViewProjParam &&
    (sceGxmProgramParameterGetCategory(g_shadowWorldViewProjParam) ==
       SCE_GXM_PARAMETER_CATEGORY_UNIFORM));

// Find fragment uniforms by name and cache parameter info
// Note: name lookup is a slow load-time operation
const SceGxmProgram *cubeFragmentProgram =
    sceGxmShaderPatcherGetProgramFromId(g_cubeFragmentProgramId);
SCE_DBG_ASSERT(cubeFragmentProgram);
g_cubeWorldLightPosParam = sceGxmProgramFindParameterByName(
    cubeFragmentProgram, "worldLightPos");
SCE_DBG_ASSERT(g_cubeWorldLightPosParam &&
(sceGxmProgramParameterGetCategory(g_cubeWorldLightPosParam) ==
    SCE_GXM_PARAMETER_CATEGORY_UNIFORM));
g_cubeDiffuseColorParam = sceGxmProgramFindParameterByName(
    cubeFragmentProgram, "diffuseColor");
SCE_DBG_ASSERT(g_cubeDiffuseColorParam &&
    (sceGxmProgramParameterGetCategory(g_cubeDiffuseColorParam) ==
    SCE_GXM_PARAMETER_CATEGORY_UNIFORM));
// Find attributes by name to create vertex format bindings
const SceGxmProgramParameter *paramPositionAttribute =
    sceGxmProgramFindParameterByName(cubeVertexProgram, "aPosition");
SCE_DBG_ASSERT(paramPositionAttribute &&
    (sceGxmProgramParameterGetCategory(paramPositionAttribute) ==
    SCE_GXM_PARAMETER_CATEGORY_ATTRIBUTE));
const SceGxmProgramParameter *paramNormalAttribute =
    sceGxmProgramFindParameterByName(cubeVertexProgram, "aNormal");
SCE_DBG_ASSERT(paramNormalAttribute && (sceGxmProgramParameterGetCategory(
    paramNormalAttribute) == SCE_GXM_PARAMETER_CATEGORY_ATTRIBUTE));

// Create shaded triangle vertex format
SceGxmVertexAttribute basicVertexAttributes[3];
SceGxmVertexStream basicVertexStreams[1];
basicVertexAttributes[0].streamIndex = 0;
basicVertexAttributes[0].offset = 0;
basicVertexAttributes[0].format = SCE_GXM_ATTRIBUTE_FORMAT_F32;
basicVertexAttributes[0].componentCount = 3;
basicVertexAttributes[0].regIndex =

sceGxmProgramParameterGetResourceIndex(paramPositionAttribute);
basicVertexAttributes[1].streamIndex = 0;
basicVertexAttributes[1].offset = 12;
basicVertexAttributes[1].format = SCE_GXM_ATTRIBUTE_FORMAT_F32;
basicVertexAttributes[1].componentCount = 3;
basicVertexAttributes[1].regIndex =
```

```
sceGxmProgramParameterGetResourceIndex(paramNormalAttribute);
basicVertexStreams[0].stride = sizeof(BasicVertex);
basicVertexStreams[0].indexSource = SCE_GXM_INDEX_SOURCE_INDEX_16BIT;

// Create cube vertex program
err = sceGxmShaderPatcherCreateVertexProgram(
    g_shaderPatcher,
    g_cubeVertexProgramId,
    basicVertexAttributes,
    2,
    basicVertexStreams,
    1,
    &g_cubeVertexProgram);
SCE_DBG_ASSERT(err == SCE_OK);

// Create cube fragment program

SCE_DBG_ASSERT(err == SCE_OK);
err = sceGxmShaderPatcherCreateFragmentProgram(
    g_shaderPatcher,
    g_cubeFragmentProgramId,
    SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4,
    MSAA_MODE,
    NULL,

sceGxmShaderPatcherGetProgramFromId(g_cubeVertexProgramId),
    &g_cubeFragmentProgram);
SCE_DBG_ASSERT(err == SCE_OK);

// Create shadow vertex program
err = sceGxmShaderPatcherCreateVertexProgram(
    g_shaderPatcher,
    g_shadowVertexProgramId,
    basicVertexAttributes,
    1,
    basicVertexStreams,
    1,
    &g_shadowVertexProgram);
SCE_DBG_ASSERT(err == SCE_OK);

// Create shadow fragment program
// Note: we mask off the write to COLOR to allow libgxm to disable the fragment
// program
SceGxmBlendInfo maskInfo;
maskInfo.colorFunc = SCE_GXM_BLEND_FUNC_NONE;
maskInfo.alphaFunc = SCE_GXM_BLEND_FUNC_NONE;
maskInfo.colorSrc = SCE_GCM_BLEND_FACTOR_ZERO;
maskInfo.colorDst = SCE_GCM_BLEND_FACTOR_ZERO;
maskInfo.alphaSrc = SCE_GCM_BLEND_FACTOR_ZERO;
maskInfo.alphaDst = SCE_GCM_BLEND_FACTOR_ZERO;
maskInfo.colorMask = SCE_GXM_COLOR_MASK_NONE;
err = sceGxmShaderPatcherCreateFragmentProgram(
    g_shaderPatcher,
    g_shadowFragmentProgramId,
    SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4,
    SCE_GXM_MULTISAMPLE_NONE,
    &maskInfo,

sceGxmShaderPatcherGetProgramFromId(g_shadowVertexProgramId),
    &g_shadowFragmentProgram);
```

```
SCE_DBG_ASSERT(err == SCE_OK);

// Allocate memory for vertex and index data
g_cubeVertices = (BasicVertex *)graphicsAlloc(
    SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
    24*sizeof(BasicVertex),
    4,
    SCE_GXM_MEMORY_ATTRIB_READ,
    &g_cubeVerticesUid);
g_cubeIndices = (uint16_t *)graphicsAlloc(
    SCE_KERNEL_MEMBLOCK_TYPE_USER_RW_UNCACHE,
    36*sizeof(uint16_t),
    2,
    SCE_GXM_MEMORY_ATTRIB_READ,
    &g_cubeIndicesUid);
```

Whereas clear data consisted of a three-element position, cube data also has a three-element normal per vertex.

```
// Write vertices
BasicVertex *vertexData = g_cubeVertices;
for (uint32_t face = 0; face < 6; ++face) {
    float sign = ((face & 0x1) ? 1.0f : -1.0f);
    uint32_t axis = face >> 1;

    float ox = (axis == 0) ? sign : 0.0f;
    float oy = (axis == 1) ? sign : 0.0f;
    float oz = (axis == 2) ? sign : 0.0f;

    float ux = (axis != 0) ? 1.0f : 0.0f;
    float uy = (axis == 0) ? 1.0f : 0.0f;
    float uz = 0.0f;

    float vx = uy*oz - uz*oy;
    float vy = uz*ox - ux*oz;
    float vz = ux*oy - uy*ox;

    vertexData->x = ox - ux - vx;
    vertexData->y = oy - uy - vy;
    vertexData->z = oz - uz - vz;
    vertexData->nx = ox;
    vertexData->ny = oy;
    vertexData->nz = oz;
    ++vertexData;

    vertexData->x = ox + ux - vx;
    vertexData->y = oy + uy - vy;
    vertexData->z = oz + uz - vz;
    vertexData->nx = ox;
    vertexData->ny = oy;
    vertexData->nz = oz;
    ++vertexData;

    vertexData->x = ox + ux + vx;
    vertexData->y = oy + uy + vy;
    vertexData->z = oz + uz + vz;
    vertexData->nx = ox;
    vertexData->ny = oy;
    vertexData->nz = oz;
    ++vertexData;
```

```
            vertexData->x = ox - ux + vx;
            vertexData->y = oy - uy + vy;
            vertexData->z = oz - uz + vz;
            vertexData->nx = ox;
            vertexData->ny = oy;
            vertexData->nz = oz;
            ++vertexData;
        }

        // Write indices
        uint16_t *indexData = g_cubeIndices;
        for (uint32_t face = 0; face < 6; ++face) {
            uint32_t offset = 4*face;

            *indexData++ = offset + 0;
            *indexData++ = offset + 1;
            *indexData++ = offset + 2;

            *indexData++ = offset + 2;
            *indexData++ = offset + 3;
            *indexData++ = offset + 0;
        }
    }

    static void destroyCubeData(void)
    {
        int err = SCE_OK;
        UNUSED(err);

        // Release the shaderss
        err = sceGxmShaderPatcherReleaseFragmentProgram(g_shaderPatcher,
            g_shadowFragmentProgram);
        SCE_DBG_ASSERT(err == SCE_OK);
        err = sceGxmShaderPatcherReleaseVertexProgram(g_shaderPatcher,
            g_shadowVertexProgram);
        SCE_DBG_ASSERT(err == SCE_OK);
        err = sceGxmShaderPatcherReleaseFragmentProgram(g_shaderPatcher,
            g_cubeFragmentProgram);
        SCE_DBG_ASSERT(err == SCE_OK);
        err = sceGxmShaderPatcherReleaseVertexProgram(g_shaderPatcher,
            g_cubeVertexProgram);
        SCE_DBG_ASSERT(err == SCE_OK);

        // Free the memory used for vertices and indices
        graphicsFree(g_cubeIndicesUid);
        graphicsFree(g_cubeVerticesUid);

        // Unregister programs since we don't need them any more
        err = sceGxmShaderPatcherUnregisterProgram(g_shaderPatcher,
            g_shadowFragmentProgramId);
        SCE_DBG_ASSERT(err == SCE_OK);
        err = sceGxmShaderPatcherUnregisterProgram(g_shaderPatcher,
            g_shadowVertexProgramId);
        SCE_DBG_ASSERT(err == SCE_OK);
        err = sceGxmShaderPatcherUnregisterProgram(g_shaderPatcher,
            g_cubeFragmentProgramId);
        SCE_DBG_ASSERT(err == SCE_OK);
        err = sceGxmShaderPatcherUnregisterProgram(g_shaderPatcher,
            g_cubeVertexProgramId);
        SCE_DBG_ASSERT(err == SCE_OK);
    }
```

createShadowMapBuffer / destroyShadowMapBuffer Functions

As its name implies, the function `createShadowMapBuffer()` creates the shadow map buffer, used as both a draw buffer and a texture.

First, a buffer is allocated using the helper function `graphicsAlloc()`. For alignment, it uses the higher of `SCE_GXM_TEXTURE_ALIGNMENT` and `SCE_GXM_DEPTHSTENCIL_SURFACE_ALIGNMENT`, and both `READ` and `WRITE` attributes are requested because the SGX performs both operations during the course of rendering.

```
// Offscreen surface data and render target
SceUID g_shadowMapBufferUid;
Void *g_shadowMapBufferData;
SceGxmTexture g_shadowMapTexture;
SceGxmDepthStencilSurface g_shadowMapDepthSurface;
SceGxmRenderTarget *g_shadowMapRenderTarget;

static void createShadowMapBuffer(void)
{
    int err = SCE_OK;
    UNUSED(err);

    // Allocate memory
    const uint32_t alignedWidth = ALIGN(SHADOW_MAP_WIDTH,
        SCE_GXM_TILE_SIZEX);
    const uint32_t alignedHeight = ALIGN(SHADOW_MAP_HEIGHT,
        SCE_GXM_TILE_SIZEY);
    g_shadowMapBufferData = heapAlloc(
          HEAP_TYPE_CDRAM_RW,
          alignedWidth*alignedHeight*4,
          MAX(SCE_GXM_TEXTURE_ALIGNMENT,
          SCE_GXM_DEPTHSTENCIL_SURFACE_ALIGNMENT));
```

After the buffer is allocated, the function sets it up as a depth surface, using `sceGxmDepthStencilSurfaceInit()` with a `TILED` configuration for best rendering performance. Because the depth surface is used as a texture after rendering, the SGX needs to know to commit the depth buffer to memory. Otherwise, depth buffers are kept in SGX's internal memory by default. To set up a depth surface to be used as a texture, call `sceGxmDepthStencilSurfaceSetForceStoreMode()` with the second parameter `SCE_GXM_DEPTH_STENCIL_FORCE_STORE_ENABLED`.

```
    // Set up the surface
    err = sceGxmDepthStencilSurfaceInit(
        &g_shadowMapDepthSurface,
        SHADOW_MAP_DEPTH_FORMAT,
        SCE_GXM_DEPTH_STENCIL_SURFACE_TILED,
        alignedWidth,
        g_shadowMapBufferData,
        NULL);
    SCE_DBG_ASSERT(err == SCE_OK);

    // Force the surface to always write data to memory (since we want to
    // texture from it)
    sceGxmDepthStencilSurfaceSetForceStoreMode(
        &g_shadowMapDepthSurface,
        SCE_GXM_DEPTH_STENCIL_FORCE_STORE_ENABLED);
```

In addition, a texture structure must be declared using `sceGxmTextureInitTiled()` so as to match the same `TILED` memory setup that was used when creating the equivalent rendering surface. The same buffer is used, with the same dimension as the surface.

```
// Set up the texture
err = sceGxmTextureInitTiled(
    &g_shadowMapTexture,
    g_shadowMapBufferData,
    SHADOW_MAP_TEXTURE_FORMAT,
    SHADOW_MAP_WIDTH,
    SHADOW_MAP_HEIGHT,
    1);
SCE_DBG_ASSERT(err == SCE_OK);

// Set linear filter for PCF
sceGxmTextureSetMagFilter(&g_shadowMapTexture,
    SCE_GXM_TEXTURE_FILTER_LINEAR);
sceGxmTextureSetMinFilter(&g_shadowMapTexture,
    SCE_GXM_TEXTURE_FILTER_LINEAR);
```

And finally, the last step creates the render target structure, used to initiate the rendering to the shadow map surface.

```
// Create a render target
g_shadowMapRenderTarget = createRenderTarget (SHADOW_MAP_WIDTH,
    SHADOW_MAP_HEIGHT,
    SCE_GXM_MULTISAMPLE_NONE);
}
```

The opposite function, `destroyShadowMapBuffer()`, destroys the render target structure and frees the buffer allocated for both the surface and texture. Note that there is no need to specifically destroy the surface nor the texture, as they don't involve any extra internal resource beyond the structures themselves.

```
static void destroyShadowMapBuffer(void)
{
    // Destroy render target
    destroyRenderTarget(g_shadowMapRenderTarget);

    // Free the memory
    heapFree(g_shadowMapBufferData);
}
```

update Function

The `update()` function performs everything necessary to update the scene. It starts by reading the controller.

```
// Update data
Bool g_quit = false;
Float g_rotationAngle = 0.0f;
Matrix4 g_cubeWorldMatrices[CUBE_COUNT];
Matrix4 g_mainViewProjMatrix;
Matrix4 g_shadowViewProjMatrix;
Matrix4 g_shadowTexViewProjMatrix;
Point3 g_worldLightPos;

// Control data
SceCtrlData g_ctrlData;

static void update(void)
{
    // Update control buffer
    sceCtrlReadBufferPositive(0, &g_ctrlData, 1);
```

Then it updates a single rotation value, which is used by all cubes in the scene (minus the ground's own cube).

```
// Advance rotation
g_rotationAngle += 0.25f*SCE_MATH_TWOPI/60.0f;
if (g_rotationAngle > SCE_MATH_TWOPI) g_rotationAngle -= SCE_MATH_TWOPI;
```

It proceeds with the generation of a local-to-world transformation matrix for each cube.

```
// First cube is the floor
g_cubeWorldMatrices[0] = Matrix4::translation(Vector3(0.0f, -0.5f, 0.0f))
    * Matrix4::scale(Vector3(16.0f, 0.5f, 16.0f));

// The rest are the object
for (uint32_t i = 1; i < CUBE_COUNT; ++i) {
float localAngle = (float)(i & 0x3)*SCE_MATH_PI/2.0f;
g_cubeWorldMatrices[i] = Matrix4::rotation(localAngle + g_rotationAngle,
Vector3::yAxis()) * Matrix4::translation(Vector3(2.0f, (float)
    (2*i - 1), 0.0f));
}

Point3 lookAtPos(0.0f, 8.0f, 0.0f);

g_worldLightPos = lookAtPos + 10.0f*Vector3(1.0f, 2.0f, 1.0f);
```

The light position and look-at position, although fixed, are set every time as {10, 28, 10} and {0, 8, 0} respectively. The function goes on to generate the following three matrices:

- `g_shadowViewProjMatrix`shadow: map's world-to-clip space transformation matrix

```
g_shadowViewProjMatrix
    = Matrix4::perspective(
    SCE_MATH_PI/4.0f,
     (float)SHADOW_MAP_WIDTH/(float)SHADOW_MAP_HEIGHT,
    1.0f,
    50.0f)
* Matrix4::lookAt(
    g_worldLightPos,
    lookAtPos,
    Vector3::yAxis());
```

- `g_mainViewProjMatrix`: the world-to-clip space transformation matrix

```
g_mainViewProjMatrix
    = Matrix4::perspective(
    SCE_MATH_PI/4.0f,
     (float)DISPLAY_WIDTH/(float)DISPLAY_HEIGHT,
    0.1f,
    100.0f)
* Matrix4::lookAt(
      lookAtPos + Vector3(0.0f, 20.0f, 20.0f),
      lookAtPos,
      Vector3::yAxis());
```

- `g_shadowTexViewProjMatrix`: a world-to-shadow-map space transformation matrix

```
g_shadowTexViewProjMatrix
    = Matrix4(
    Matrix3::scale(Vector3(0.5f, -0.5f, 0.5f)),
    Vector3(0.5f, 0.5f, 0.5f))
* g_shadowViewProjMatrix;
}
```

The last of the three matrices is the same as the first one, except that it includes a final conversion from clip space {-1, -1, -1} – {1, 1, 1} to shadow map space {0, 0, 0} – {1, 1, 1}. The latter space allows the `cube_f.cg` shader to use transformed values as texture coordinate to fetch the shadow map texture.

renderShadowMap Function

This function encapsulates the rendering of the shadow map. It relies on the `update()` function and all initialization functions including `initGxm()` and `createCubeData()` to have been called previously. Rendering to the shadow map is simple because all cubes use the same vertex and fragment shader and the same render state.

The only render state set up by this function is the cull mode. In the rendering itself, as described in Beginning and Ending a Scene, rendering function calls must be done between calls to `sceGxmBeginScene()` and `sceGxmEndScene()`.

The sample initiates rendering by providing `sceGxmBeginScene()` with a render target and the surfaces to draw into. To render only to a depth buffer, it uses a pointer to `g_shadowMapDepthSurface` that was previously created in `createShadowMapBuffer()`. `NULL` is passed in as color surface pointer.

No scene flag is used, because this scene's vertex processing does not depend on any previous scene's fragment processing, and no following scene's vertex processing will depend on this scene's fragment processing.

```
static void renderShadowMap(void)
{
    // Cull front faces
    sceGxmSetCullMode(g_context, SCE_GXM_CULL_CW);

    // Set up a scene, offscreen render target, no sync required
    sceGxmBeginScene(
        g_context,
        0,
        g_shadowMapRenderTarget,
        NULL,
        NULL,
        NULL,
        NULL,
    &g_shadowMapDepthSurface);
```

Once the scene is initiated, the function sets up the shadow map generation's vertex and fragment shaders, and sets the cube's vertex stream. The stream is the same for all cubes, and so it only needs to be set once, outside of the object-traversal loop.

```
// Set up for cubes
sceGxmSetVertexProgram(g_context, g_shadowVertexProgram);
sceGxmSetFragmentProgram(g_context, g_shadowFragmentProgram);
sceGxmSetVertexStream(g_context, 0, g_cubeVertices);
```

A loop follows, in which each cube (including the floor) is drawn one after another. For each cube, the sample builds a shadow map's local-to-clip space transformation matrix, and copies it to the vertex shader's `worldProjView` float4x4 uniform parameter using the function `sceGxmSetUniformDataF()`. `g_shadowWorldViewProjParam` is initialized beforehand in `createCubeData()`.

```
// Render them
for (uint32_t i = 0; i < CUBE_COUNT; ++i) {
// Compute worldViewProj
Matrix4 worldViewProj = g_shadowViewProjMatrix * g_cubeWorldMatrices[i];

// Set the vertex program constants
void *vertexDefaultBuffer;
sceGxmReserveVertexDefaultUniformBuffer(g_context, &vertexDefaultBuffer);
sceGxmSetUniformDataF(vertexDefaultBuffer, g_shadowWorldViewProjParam,
    0, 16, (float *)&worldViewProj);
```

Next, the draw function `sceGxmDraw()` is called. SGX only supports indexed geometry drawing, so the cube index array is passed in and a count of 36 indices is specified.

```
// Draw the cube
sceGxmDraw(g_context, SCE_GXM_PRIMITIVE_TRIANGLES,
    SCE_GXM_INDEX_FORMAT_U16, g_cubeIndices, 36);
}
```

The `renderShadowMap()` function is then wrapped up with a call to `sceGxmEndScene()` to conclude rendering.

```
// Stop rendering to the main render target
sceGxmEndScene(g_context, NULL, NULL);
}
```

renderMain Function

The function `renderMain()` is only slightly more complex than `renderShadowMap()`.

```
static void renderMain(void)
{
    // Cull back faces
    sceGxmSetCullMode(g_context, SCE_GXM_CULL_CCW);

    // Set up a scene, main render target, synchronised with the back
    // buffer sync
    sceGxmBeginScene(
        g_context,
        0,
        g_mainRenderTarget,
        NULL,
        NULL,
        g_displayBufferSync[g_displayBackBufferIndex],
        &g_displaySurface[g_displayBackBufferIndex],
        &g_mainDepthSurface);
```

The first extra step it performs is clearing the color and depth buffer, right after beginning the scene. This involves setting up the clear vertex and fragment shader and draw using the clear geometry previously initialized in the function `createClearData()`. Because the clear shaders have no uniform parameter, there is no need to set one for this step.

```
// Set clear shaders
sceGxmSetVertexProgram(g_context, g_clearVertexProgram);
sceGxmSetFragmentProgram(g_context, g_clearFragmentProgram);

// Draw geometry
sceGxmSetVertexStream(g_context, 0, g_clearVertices);
sceGxmDraw(g_context, SCE_GXM_PRIMITIVE_TRIANGLES,
    SCE_GXM_INDEX_FORMAT_U16, g_clearIndices, 3);
```

The inner-loop includes processing to set the cube's diffuse color to be white, red, green, or blue based on its index. The sample then sets three vertex shader uniform parameters: the cube local-to-world space transformation matrix; the world-to-clip space transformation matrix; and world-to-shadow-map space transformation matrix.

```
// Set up for cubes
sceGxmSetVertexProgram(g_context, g_cubeVertexProgram);
sceGxmSetFragmentProgram(g_context, g_cubeFragmentProgram);
sceGxmSetVertexStream(g_context, 0, g_cubeVertices);
sceGxmSetFragmentTexture(g_context, 0, &g_shadowMapTexture);
```

```
// Render them
for (uint32_t i = 0; i < CUBE_COUNT; ++i) {
    // Compute a color
    float diffuseColor[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
    if (i == 0) {
    diffuseColor[0] = diffuseColor[1] = diffuseColor[2] = 0.5f;
        } else {
            switch (i & 0x3) {
            case 0:  diffuseColor[0] = diffuseColor[1] = diffuseColor[2] =
                1.0f; break;
            case 1:  diffuseColor[0] = 1.0f; break;
            case 2:  diffuseColor[1] = 1.0f; break;
            case 3:  diffuseColor[2] = 1.0f; break;
            }
        }

        // Set the vertex program constants
        void *vertexDefaultBuffer;
        sceGxmReserveVertexDefaultUniformBuffer(g_context,
            &vertexDefaultBuffer);
        sceGxmSetUniformDataF(vertexDefaultBuffer, g_cubeWorldParam,
            0, 16, (float *)&g_cubeWorldMatrices[i]);
        sceGxmSetUniformDataF(vertexDefaultBuffer, g_cubeViewProjParam,
            0, 16, (float *)&g_mainViewProjMatrix);
        sceGxmSetUniformDataF(vertexDefaultBuffer,
            g_cubeShadowTexViewProjParam, 0, 16,
            (float *)&g_shadowTexViewProjMatrix);
```

Two fragment shader uniform parameters are also set within the inner-loop: the light's position in world space, and the cube's diffuse color, which was just calculated.

It finally calls `sceGxmPadHeartbeat()` so a Razor performance analysis capture of the application can be performed.

```
// Set fragment program constants
void *fragmentDefaultBuffer;
sceGxmReserveFragmentDefaultUniformBuffer(
    g_context,&fragmentDefaultBuffer);
sceGxmSetUniformDataF(fragmentDefaultBuffer, g_cubeWorldLightPosParam,
    0, 3, (float *)&g_worldLightPos);
sceGxmSetUniformDataF(fragmentDefaultBuffer, g_cubeDiffuseColorParam,
    0, 3, diffuseColor);

// Draw the cube
sceGxmDraw(g_context, SCE_GXM_PRIMITIVE_TRIANGLES,
    SCE_GXM_INDEX_FORMAT_U16, g_cubeIndices, 36);
}

// Stop rendering to the main render target
sceGxmEndScene(g_context, NULL, NULL);

// PA heartbeat to notify end of frame
sceGxmPadHeartbeat(
    &g_displaySurface[g_displayBackBufferIndex],
    g_displayBufferSync[g_displayBackBufferIndex]);
}
```

main Function

Like all libgxm samples, this sample can be captured using the Razor performance analysis tool. To use it, define either `ENABLE_RAZOR_HUD`, `ENABLE_RAZOR_CAPTURE`, or both. If you define `ENABLE_RAZOR_CAPTURE`, then the application will be set to automatically capture a frame at frame 100.

```
// Entry point
int main(void)
{
    int err = SCE_OK;
    UNUSED(err);

#ifdef ENABLE_RAZOR_HUD
    // Initialize the Razor HUD system.
    // This should be done before the call to sceGxmInitialize().
    err = sceSysmoduleLoadModule( SCE_SYSMODULE_RAZOR_HUD );
    SCE_DBG_ASSERT(err == SCE_OK);
#endif

#ifdef ENABLE_RAZOR_CAPTURE
    // Initialize the Razor capture system.
    // This should be done before the call to sceGxmInitialize().
    err = sceSysmoduleLoadModule( SCE_SYSMODULE_RAZOR_CAPTURE );
    SCE_DBG_ASSERT(err == SCE_OK);

    // Trigger a capture after 100 frames.
    sceRazorCaptureSetTrigger( 100, "host0:shadow_mapping.sgx" );
#endif
```

The `main()` function includes five initialization functions: `initDbgFont()`, `initGxm()`, `createClearData()`, `createCubeData()`, and `createShadowMapBuffer()`, all described previously.

```
// Initialize
initDbgFont();
initGxm();

// Create graphics data
createClearData();
createCubeData();
createShadowMapBuffer();

// Message for SDK sample auto test
printf("## api_libgxm/shadow_mapping: INIT SUCCEEDED ##\n");
```

Now that everything is initialized, it's time for the main loop. Within that loop, the sample updates the scene, renders the shadow map, renders the main scene, and then cycles the display buffers. At the end of each loop instance, a new frame will be displayed to the user.

Note that the loop goes on as long as `g_quit` is `false`. The `update()` function sets it to `true` when the Select button is pressed by the user.

```
// Loop until exit
while (!g_quit) {
    // Do update step
    update();

    // Render
    renderShadowMap();
    renderMain();

    // Queue a display swap and cycle our buffers
    cycleDisplayBuffers();
}
```

If and when the user presses the Select button, the application exits the main loop and the un-initialization sequence starts. `sceGxmFinish()` is called first, to make sure all pending graphics processing is completed before any resource is freed in following function calls.

```
// Wait until rendering is done
sceGxmFinish(g_context);

// Destroy graphics data
destroyShadowMapBuffer();
destroyCubeData();
destroyClearData();

// Terminate graphics
termGxm();
termDbgFont();

#ifdef ENABLE_RAZOR_CAPTURE
    // Terminate Razor capture.
    // This should be done after the call to sceGxmTerminate().
    sceSysmoduleUnloadModule( SCE_SYSMODULE_RAZOR_CAPTURE );
#endif

#ifdef ENABLE_RAZOR_HUD
    // Terminate Razor HUD.
    // This should be done after the call to sceGxmTerminate().
    sceSysmoduleUnloadModule( SCE_SYSMODULE_RAZOR_HUD );
#endif

    // Message for SDK sample auto test
    printf("## api_libgxm/shadow_mapping: FINISHED ##\n");
    return SCE_OK;
}
```

Appendix A: Summary of PSP™ and PlayStation®Vita Differences

The graphics chips in PSP™ and PlayStation®Vita are vastly different. Here is an overview of those various differences, from the point of view of a graphics programmer.

Feature	PSP™	PlayStation®Vita
Matrix system	OpenGL API-like	User-specified
Supported primitive types	Points, lines, triangles, rectangles, triangle strips, triangle fans	Points, lines, triangles, triangle strips, triangle fans
Curved surface primitives support	Bezier and spline surfaces	None
Supported vertex formats	Texture coordinate: 8-bit, 16-bit, 32-bit floating-point Color: 5:6:5, 5:5:5:1, 4:4:4:4, 8:8:8:8 Normal: 8-bit, 16-bit, 32-bit floating-point Vertex: 8-bit, 16-bit, 32-bit floating-point Bones weight: 8-bit, 16-bit, 32-bit floating-point Indices: 8-bit, 16-bit	Signed/unsigned 8-bit Signed/unsigned 16-bit Signed/unsigned 8-bit normalized Signed/unsigned 16-bit normalized 16-bit floating-point 32-bit floating-point (for all attributes)
Vertex data storage	Fixed order	Free-form
Supported index formats	8- and 16-bit	16- and 32-bit
Morphing	Hardware support, up to 8	Fully programmable
Skinning	Hardware support, up to 8 at a time	Fully programmable
Shading	Flat and Gouraud	Flat and Gouraud
Lighting	OpenGL API-like: Ambient, directional, omni, and spot lights, up to 4	Fully programmable
Color calculation	Single and Separate	Fully programmable
Clipping	Near plane only in hardware	Full clipping in hardware
High-level culling	Hardware support	Up to 16384 visibility tests
Texture mapping	Non-perspective correct	Perspective correct
Shade mapping	Limited, hardware support	Fully programmable
Texture scale / offset	Limited, hardware support	Fully programmable
Supported texture formats	5:6:5, 5:5:5:1, 4:4:4:4, 8:8:8:8, 4-bit index color, 8-bit index color, 16-bit index color, 32-bit index color, DXT1, DXT3, DXT5	U8, U4U4U4U4, U5U6U5, U8U8, U1U5U5U5, U8U8U8U8, U16, F16, F16F16, F32, PVRT2BPP, PVRT4BPP, PVRTII2BPP, PVRTII4BPP, YUV422, S8, U8U3U3U2, S5S5U6, S8S8, S16, S8S8S8S8, U2U10U10U10, U16U16, S16S16, F32M, X8U24, U8U24, U32, S32, SE5M9M9M9, F11F11F10, F16F16F16F16, U16U16U16U16, S16S16S16S16, F32F32, U32U32, YUV420P2, YUV420P3, P4, P8, U8U8U8, S8S8S8, U2F10F10F10, UBC1, UBC2, UBC3, UBC4, SBC4, UBC5, SBC5, X8S8S8U8
Texture size	From 1x1 to 512x512, 2D only	From 1x1 to 4096x4096, 1D, 2D and Cube map
Texture wrap modes	Wrap and clamp	Repeat, mirror, clamp, Mirror clamp, repeat with border, clamp with border
Texture functions	Limited, hardware support	Fully programmable
Color doubling and addition	Limited, hardware support	Fully programmable

Feature	PSP™	PlayStation®Vita
Fogging	Limited, hardware support	Fully programmable
Color surface formats	Only 1 U5U6U5, U5U5U5U1, U4U4U4U4, U8U8U8U8	Up to 4 U8U8U8U8, U5U6U5, U1U5U5U5, U4U4U4U4, U8, U8U8U8, U8U3U3U2, F16, F16F16, F32, S16, S16S16, U16, U16U16, U2U10U10U10, S8, S5S5U6, U8U8, S8S8, S8S8S8S8, F16F16F16F16, F32F32, F11F11F10, SE5M9M9M9, U2F10F10F10, U8S8S8U8
Depth buffer	16-bit	16-bit, 24-bit (32-bit with stencil), 32-bit floating-point
Stencil buffer	Up to 4 bits	8-bit (32-bit with depth)
Alpha blending equations	Programmable fixed	Fully programmable
Dithering	User-specified 4x4 matrix	Hardware support
Logical operations	Fixed	Fully programmable
Anti-aliasing	Per primitive only	Full-screen 2X and 4X MSAA
Buffer clearing	Hardware support	Done as regular rendering
Screen resolution	480x272	960x544
VRAM configuration	2 MB EDRAM	128 MiB VRAM (112 MiB available to developers)