Post-processing Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

About This Document

Purpose

This document describes the implementation and performance characteristics of a number of common post-processing techniques as implemented on PlayStation®Vita.

Audience

This document is intended for engineers who are interested in how post-processing techniques perform on the PlayStation®Vita SGX graphics chip. A basic knowledge of real-time rendering is assumed.

Organization of This Document

This document provides the following major sections:

- Chapter 1, Color Correction, describes how to implement efficient per-pixel color correction using programmable blending.
- Chapter 2, Downsampling, describes how to create reduced resolution color and depth buffers.
- Chapter 3, Bloom, describes the implementation and performanm ce characteristics of Kawase Bloom.
- Chapter 4, Motion Blur and Depth of Field, describes how to implement combined motion blur and depth of field.
- Chapter 5, Recommendations, provides suggestions about how you can achieve the best performance for post-processing on PlayStation®Vita GPU.

Typographic Conventions

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

Hyperlinks

Hyperlinks (underlined and in blue) are available to help you to navigate around the document. To return to where you clicked a hyperlink, select **View > Toolbars > More Tools** from the Adobe® Reader® main menu, and then enable the **Previous View** and **Next View** buttons.

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;
```

Related Documentation and Other Resources

Errata

You can find any updates or amendments to this guide in the release notes that accompany the SDK release packages.

Source Code

Source code for all implemented post-processing techniques is available at:

```
SDK/target/samples/sample_code/graphics/tutorial_postprocessing
```

References

[1] E. Reinhart, M. Stark, P. Shirley, J. Ferwada, "Photographic Tone Reproduction for Digital Images" in *ACM Transactions on Graphics*, July 2002.

[2] J. Demers, "Depth of Field: A Survey of Techniques", in *GPU Gems*, 2004.

[3] M. Kawase, "Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L", 2003.

1 Color Correction

Overview of Technique

Color correction (also known as color grading) is a technique for enhancing or adapting the color of images. This technique has become very popular in motion pictures where it is typically performed digitally by specialist color graders. Color correction can be implemented in many different ways, each giving the grader varying levels of control of the final image. This sample uses one of the simpler approaches that allows control over the brightness, hue, contrast, and saturation of the image.

Like most post-processing techniques, color correction can be implemented by drawing a full-screen quadrilateral that samples each pixel of the color surface containing the main scene. For each pixel that is sampled, some calculations are applied to the color value retrieved from the main scene color surface and the result is stored in a different color surface.

Brightness, Hue, Contrast, and Saturation

Brightness is a measure of the amount of light reflected or radiated from a source.

Hue is a measure of the dominant fundamental component of a particular color.

Contrast is the extent to which the color and light differ from one part of an image to another.

Saturation is a measure of the perceived intensity of a given color; it describes how intense (saturated) or monochromatic (unsaturated) a particular color is.

The transformations for brightness, hue, contrast, and saturation are computed on the CPU and propagated to the color correction fragment shader as a single 4x4 matrix through the libGxm graphics API as shown in Listing 1.

Listing 1 Setting the Value of the Color Correction Matrix Uniform Parameter

```
void* uniformBuffer = NULL;
sceGxmReserveFragmentDefaultUniformBuffer(context, &uniformBuffer);
sceGxmSetUniformDataF(uniformBuffer, g_colorMatParam, 0, 16, colorMatrix);
```

PlayStation®Vita Implementation Details

This section discusses important platform-specific implementation details that help accelerate color correction on the PlayStation®Vita platform. The source code for this can be found at:

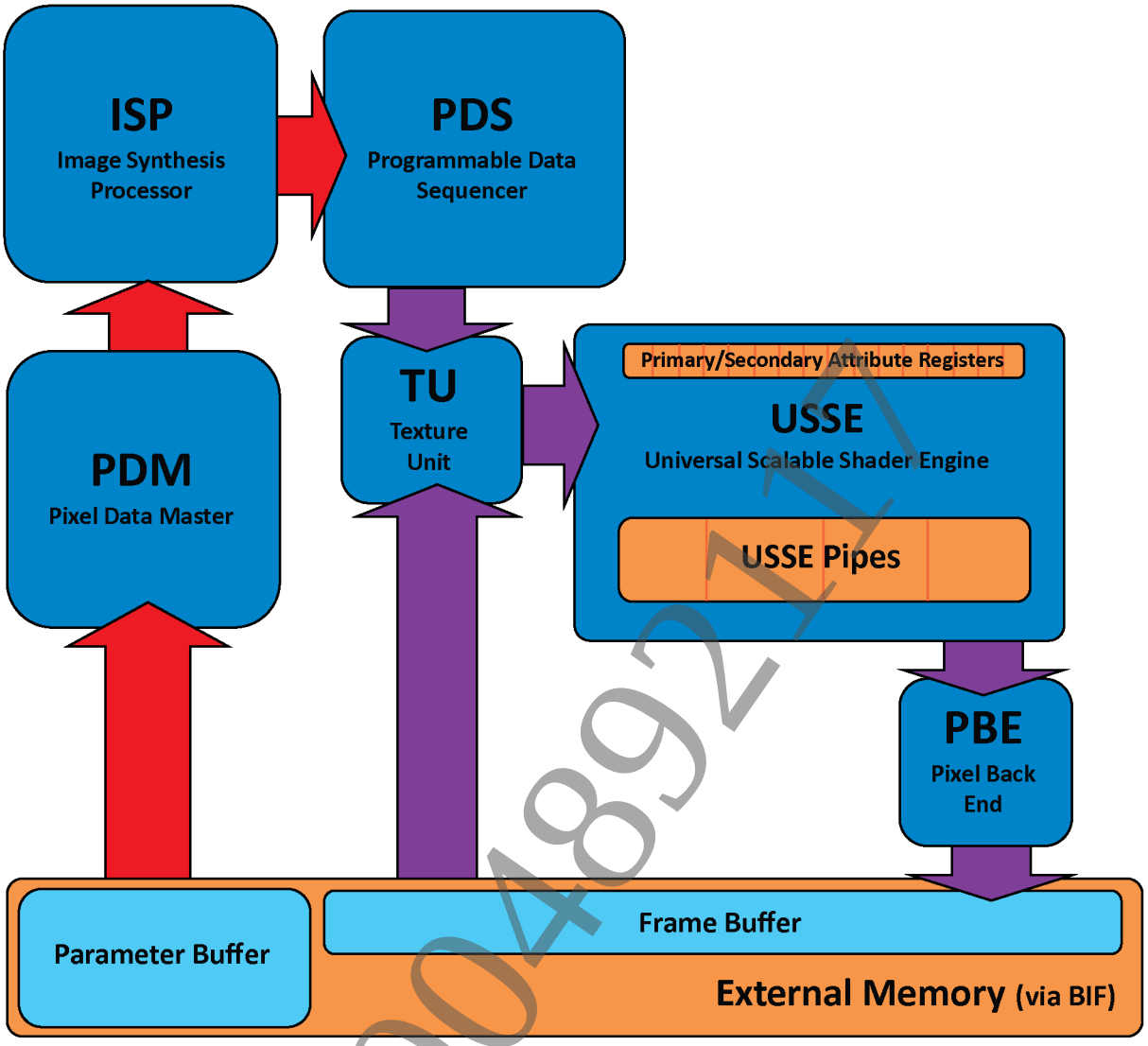
```
SDK/target/samples/source_code/graphics/tutorial_postprocessing/
color_correction.c
```

Keeping Data On-Chip

Figure 1 shows the data flow for fragment processing for an implementation of color correction that uses an approach commonly used on most traditional stream GPUs. The purple arrows show the portion of the data flow that is of particular interest to this discussion. The main scene is first rendered and the Pixel Back End (PBE) unit writes the buffer's tiles to memory as they are completed. A second scene is then started that draws a single screen-sized quadrilateral, whose shader simply reads the previous render target as a texture.

The USSE is responsible for the execution of your fragment programs. After it has calculated the color for a particular pixel, it writes this value to one of a bank of output registers within the USSE itself. When the entire tile is done, the PBE unit performs any format conversion and packing on the values in the output registers before writing these values to your render target, which is located in either graphics memory or main memory.

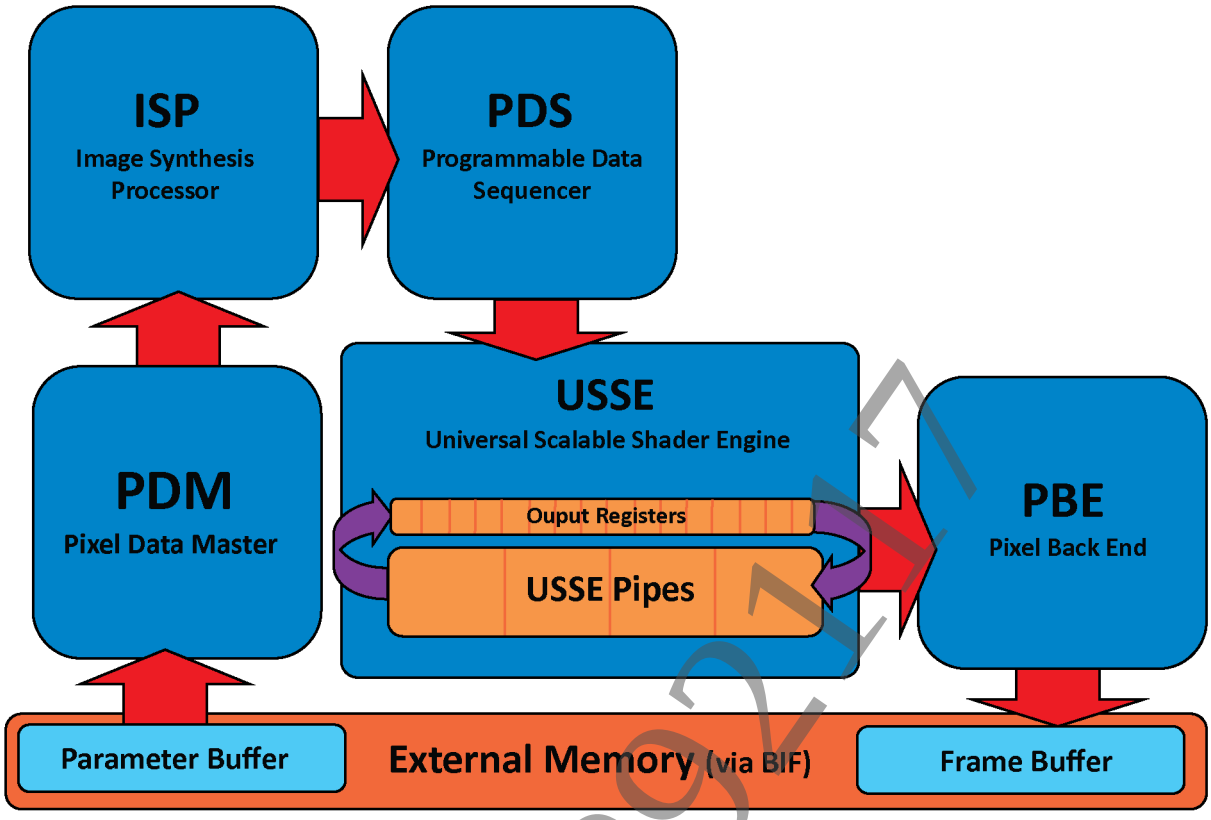
Figure 1 Data Flow Through the SGX543MP4+ Using Multiple Scenes



Minimizing memory bandwidth is a dominant philosophy in the design of graphics processors for portable architectures. Thus the SGX543MP4+ graphics chip supports reading the last written color of the target pixel for a given fragment program. This allows color surface data to be kept entirely on-chip, saving valuable memory bandwidth. This feature is extreme useful for techniques such as alpha blending, but can also be very useful in certain types of post-processing.

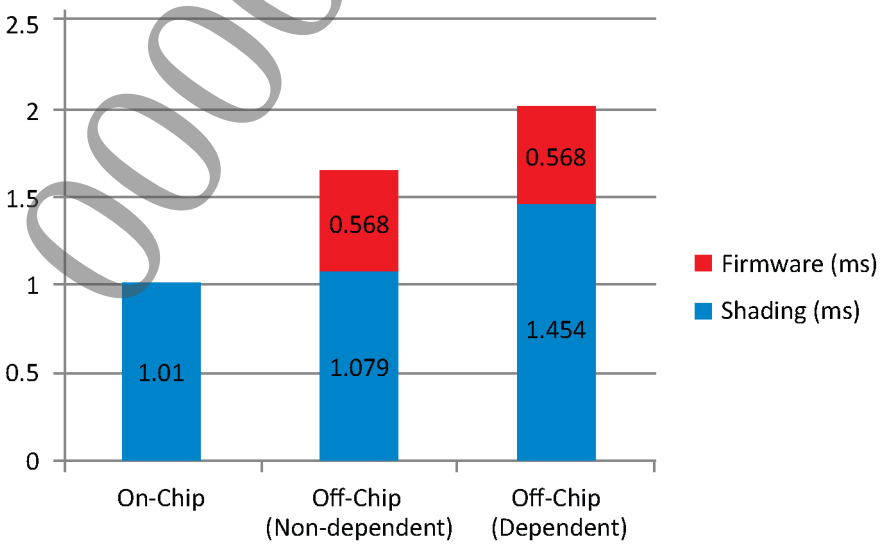
Figure 2 shows the data flow for fragment processing through SGX543MP4+ when using this hardware feature. The important departure from the standard data flow diagram occurs inside the USSE. In the multi-scene approach shown in Figure 1, color values are read from a texture in memory by the PDS unit (assuming non-dependent reads) and forwarded to the USSE's primary attribute registers prior to the fragment program being executed in the USSE. However, by using programmable blending it is possible to access the bank of output registers; specifically it is possible to read the output register containing the color of the particular fragment that is about to be written to. The purple arrows in Figure 2 show the reuse of the on-chip color value from the output register. As you can see by comparing the two figures, this feature eliminates the need to take the much longer path shown in Figure 1 (which goes via the texture unit and memory), thus saving valuable memory bandwidth and time.

Figure 2 Data Flow Through the SGX543MP4+ When Using Programmable Blending



To use this hardware feature, you must use the Cg language extensions provided by the shader compiler as described in the "Programmable Blending" section of the *Shader Compiler User's Guide*. Figure 3 shows the performance for a matrix-based color correction shader using data from both on and off the graphics chip. As you can see, spilling the color buffer back to memory and reading it through non-dependent texture fetches is not too costly, although the additional cost introduced by the firmware when a scene is changed makes doing this undesirable. This overhead introduced by the scene change is shown as the red part of the bars in Figure 3.

Figure 3 Performance of Color Correction Reading Fragment Color from Both on and off Chip

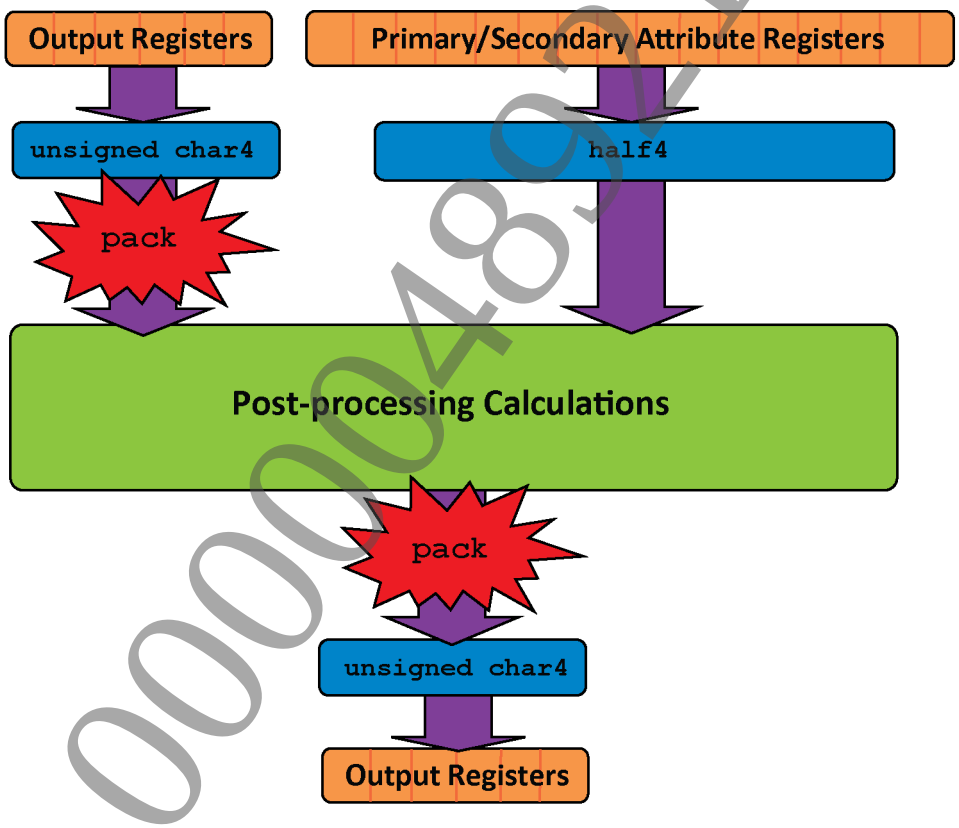


Color Buffer Interaction with Floating-Point Data Types

The vast majority of fragment shaders will produce a final color by some calculations based on a mixture of interpolated values, data sampled from textures, and uniform parameters propagated to the program through the GXM rendering library. Conveniently this data is often stored in a floating-point format (or, in the case of data sampled from a texture, can usually be converted to a floating-point format by the texture unit immediately after filtering). Refer to the appendix of the *GPU User's Guide* to ensure that your particular formats conversion is supported. Sometimes, however, data must be converted from one format to another within the USSE. To facilitate this, there are a number of packing and format conversion instructions that the USSE supports to coerce data into the correct format for calculations. These conversion instructions are generated by the shader compiler wherever format conversion must take place for interoperability between different data involved in a specific calculation.

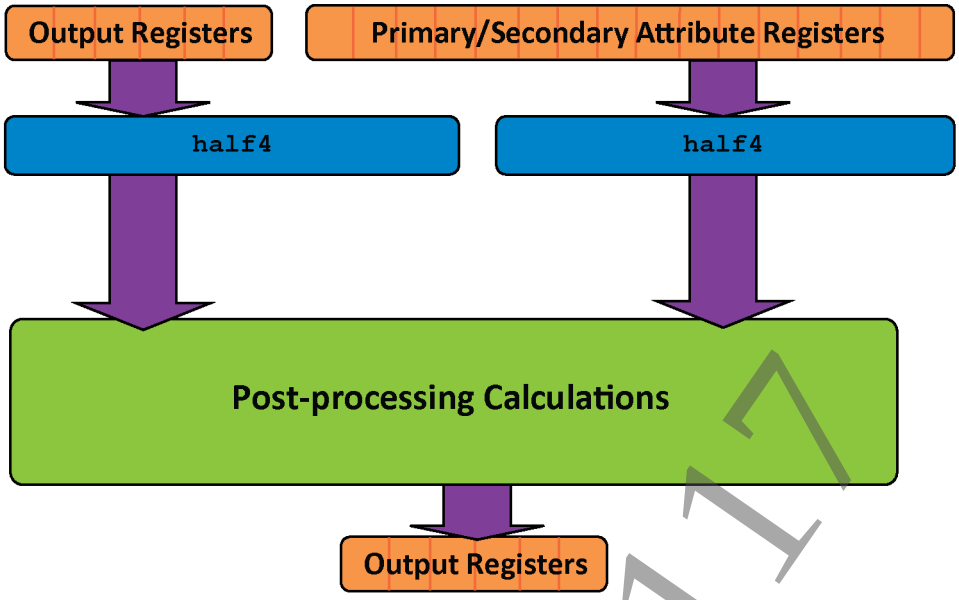
The previous section advised you to keep data on-chip as much as possible to reduce memory bandwidth requirements and boost rendering speed. The SGX543MP4+ GPU supports color surfaces in a variety of formats (refer to the *GPU User's Guide* for details); some of these formats (such as common 32-bit per pixel integer formats) may require format conversion to occur before the pixel data can interact with uniform parameters or interpolated data stored in floating-point format. This is shown in Figure 4.

Figure 4 Data Flow for On-Chip unsigned char4 Color Format When Interacting with half4 Uniform Parameters



Conveniently the SGX543MP4+ also supports 64-bit per pixel formats, such as `half4`. As shown in Figure 5, this means that interaction between values read from the output registers and higher-precision floating-point uniform parameters can occur without the compiler needing to generate extraneous format conversion instructions.

Figure 5 Data Flow for On-chip half4 Color Format When Interacting with half4 Uniform Parameters



For the church scene in the Post-processing Tutorial Sample shown in Figure 6, there were actually significant performance gains by using 64-bit per pixel output register formats (measurements are provided in Figure 7). This is because the 64-bit version does not require additional packing instructions to convert to the results of the shading calculations from `half4` into `char4`. The 64-bit output register format can be used without penalty in order to avoid packing instructions in subsequent post-processing passes. By selecting an appropriate color surface format, a conversion to a lower-precision format in off-chip memory is carried out by the PBE unit when the tile is completed. This means that using the higher-precision output register format does not cause additional memory or bandwidth requirements. For details about supported format conversions, refer to Appendix A of the *GPU User's Guide*.

Figure 6 Church Scene in Post-processing Tutorial Sample

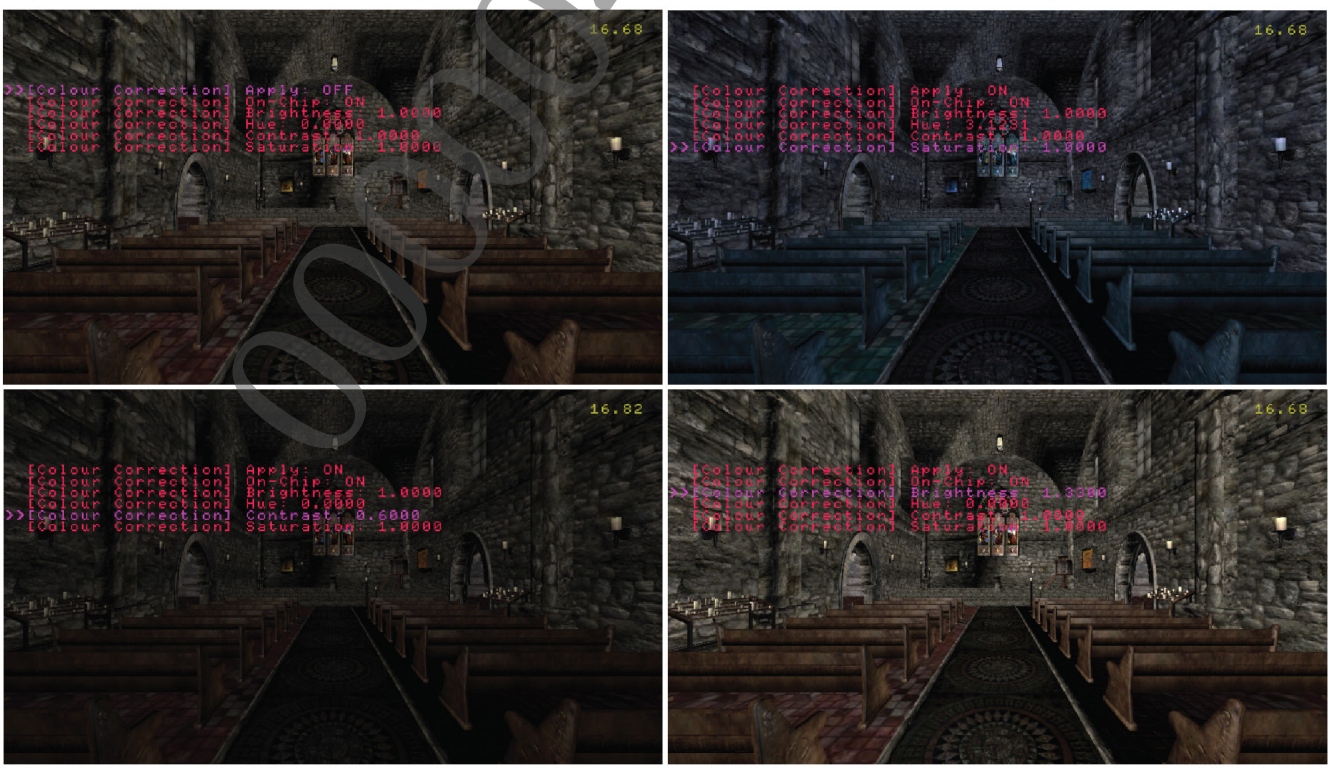
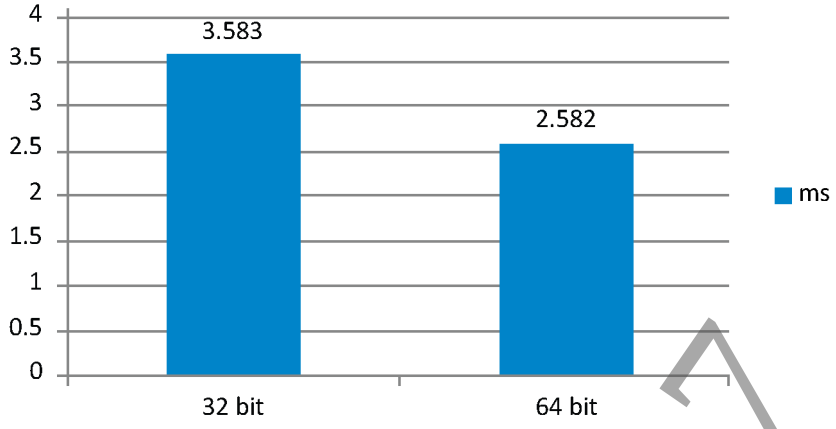


Figure 7 Amount of USSE Time Spent Fragment Shading for the Main Scene with Both 32-Bit and 64-Bit Output Register Widths



Matrix Transposition with Uniform Values

Using a matrix to transform each pixel is the most efficient way of applying the series of linear equations involved in color correction. The matrix transformation can be applied to the color as a row or column operation with each generating a slightly different disassembly. To transform a vector by a matrix, Cg provides the intrinsic `mul()` function. The selection of a row or column operation is made by the order of the arguments passed to this function. The disassembly for column and row major transformations respectively is shown below in Listing 2 and Listing 3.

Listing 2 Disassembly for Column-Major Matrix Transformation

```
Estimated cost: 5 cycles, parallel mode
Register count: 2 PAs, 0 temps, 10 SAs *
Texture reads:  0 non-dependent, dependent: 0 unconditional, 0 conditional
Primary program:
 0:     dot.f16      pa0.x, sa0.xyzw, o0.xyzw
 1:     dot.f16      pa0.-y, sa2.xyzw, o0.xyzw
 2:     dot.f16      pa0.--z, sa4.xyzw, o0.xyzw
 3:     dot.f16      pa0.---w, sa6.xyzw, o0.xyzw
 4:     mov.f32      o0.xy, pa0.xy
```

Listing 3 Disassembly for Row-Major Matrix Transformation

```
Estimated cost: 7 cycles, parallel mode
Register count: 2 PAs, 2 temps, 10 SAs *
Texture reads:  0 non-dependent, dependent: 0 unconditional, 0 conditional
Primary program:
 0:     mul.f16          pa0.xyzw, o0.zzzz, sa4.xyzw
 1:     mov.f16          r0.xyzw, sa6.xyzw
 2:     mad.f16          pa0.xyzw, r0.xyzw, o0.wwww, pa0.xyzw
 3:     mov.f16          r0.xyzw, sa2.xyzw
 4:     mad.f16          pa0.xyzw, r0.xyzw, o0.yyyy, pa0.xyzw
 5:     mov.f16          r0.xyzw, sa0.xyzw
 6:     mad.f16          o0.xyzw, r0.xyzw, o0.xxxx, pa0.xyzw
```

Ignoring the `mov` operations (which will be discussed in detail later) the row-major version of the matrix transformation generates a single `mul` instruction, followed by three `mad` instructions. However, because the alpha component of our color channel is of no interest, treating the color as a row-vector allows an optimization that is unavailable with the column-major version. By always assuming that the `w` component (alpha channel) of the row vector is exactly 1, the last row of the matrix can simply be added into the calculation. Because vector addition is a commutative operation, the first `mul` instruction in

Listing 3 can be replaced with a `mad` and reduce the overall instruction count by one. This is shown in Listing 4

Listing 4 Disassembly for Row-Major Matrix Transformation Assuming w Is Exactly 1

```
Estimated cost: 6 cycles, parallel mode
Register count: 2 PAs, 2 temps, 10 SAs *
Texture reads:  0 non-dependent, dependent: 0 unconditional, 0 conditional
Primary program:
 0:     mov.f16         pa0.xyzw, sa4.xyzw
 1:     mad.f16         pa0.xyzw, pa0.xyzw, o0.zzzz, sa6.xyzw
 2:     mov.f16         r0.xyzw, sa2.xyzw
 3:     mad.f16         pa0.xyzw, r0.xyzw, o0.yyyy, pa0.xyzw
 4:     mov.f16         r0.xyzw, sa0.xyzw
 5:     mad.f16         o0.xyzw, r0.xyzw, o0.xxxx, pa0.xyzw
```

Uniform parameters are uploaded from memory to an area of memory partitioned from the USSE Unified Store by the Programmable Data Sequencer (PDS) unit; this usually occurs before your vertex or fragment shader is executed on the USSE. The area in the USSE Unified Store from which uniform parameters are read from your vertex and fragment programs is known as secondary attribute registers. Some USSE instructions on the SGX543MP4+ have restrictions imposed on them by the limited number of bits used to represent source/target registers used by the operands in their op codes. This means that certain operands cannot directly access secondary attribute registers (among other things) in the USSE Unified Store and that the PlayStation®Vita Shader Compiler will generate instructions to relocate these values into primary attribute registers or temporary registers where necessary. This can be seen in the psp2shaderperf output shown in Listing 4 on lines 0, 2, and 4 of the primary program.

This movement of data can add unwanted overhead to your vertex and fragment programs; this overhead can in some cases dominate the performance of your draw call. In many post-processing passes, performance is limited by the fragment program running on the USSE. This means that relocating our uniform parameters into primary attribute registers is desirable because it will reduce the number of instructions in the fragment program. In order to achieve this, we propagate the uniform values via the interpolation output from the associated vertex program, through the parameter buffer, and into the fragment program as varying parameters. The psp2shaderperf output for this version of the color correction fragment program is shown in Listing 5.

Listing 5 Disassembly for Matrix Transformation with Uniform Parameters in Primary Attribute Registers

```
Estimated cost: 3 cycles, parallel mode
Register count: 8 PAs, 0 temps, 2 SAs *
Texture reads:  0 non-dependent, dependent: 0 unconditional, 0 conditional
Primary program:
 0:     mad.f16         pa4.xyzw, pa4.xyzw, o0.zzzz, pa6.xyzw
 1:     mad.f16         pa2.xyzw, pa2.xyzw, o0.yyyy, pa4.xyzw
 2:     mad.f16         o0.xyzw, pa0.xyzw, o0.xxxx, pa2.xyzw
```

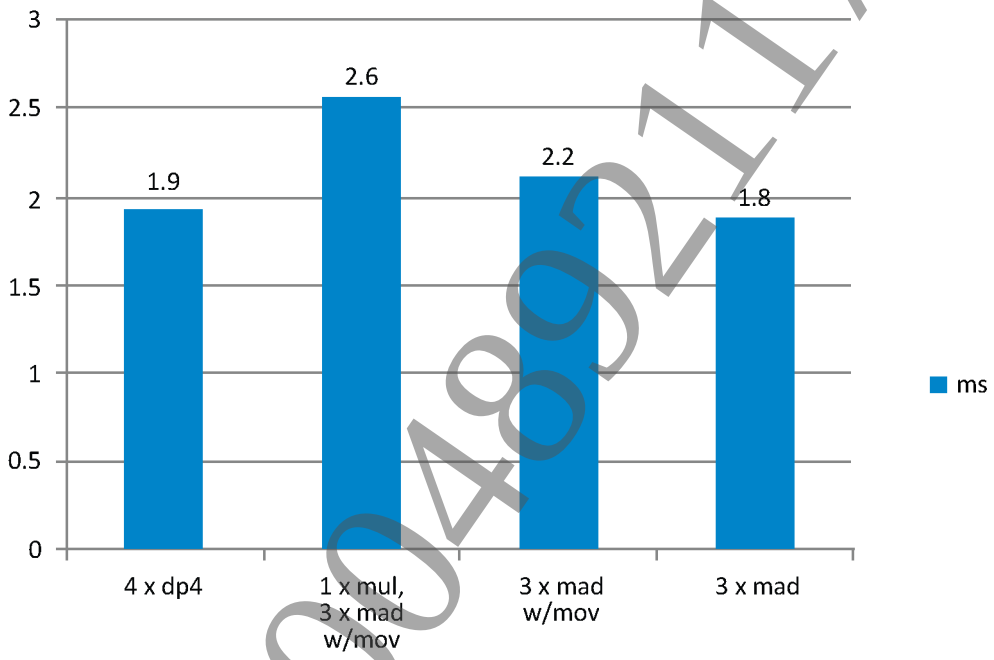
For draw calls involved in post-processing (where the vertex count is typically low and parameter buffer usage is therefore minimal), this technique works well and requires minimal space in your parameter buffer and only a little extra work from the PDS unit. However, in a more general context this technique should be used with great care because it will lead to increased usage of the parameter buffer. Note that if we are concerned about applying our transformation to alpha values as well, then we could simply use uniform parameters in the usual way because the desired operand in the `dp4` instruction is able to access the secondary attribute registers directly. The Cg code that generates the most optimal disassembly is shown in Listing 6.

Listing 6 Cg Source Code for the Most Optimal Color Correction Implementation on PlayStation®Vita

```
__nativecolor __regformat half4 main ( __regformat half4 color : FRAGCOLOR,
                                       half4 row1 : TEXCOORD0,
                                       half4 row2 : TEXCOORD1,
                                       half4 row3 : TEXCOORD2,
                                       half4 row4 : TEXCOORD3) {

    half4x4 mat = half4x4(row1, row2, row3, row4);
    return mul(half4(color.xyz, 1), mat);
}
```

The performance of each implementation discussed above is shown in Figure 8. All measurements were taken with a linear color surface and used programmable blending as discussed in the preceding section.

Figure 8 Performance of Each Implementation with Programmable Blending and Linear Color Surface



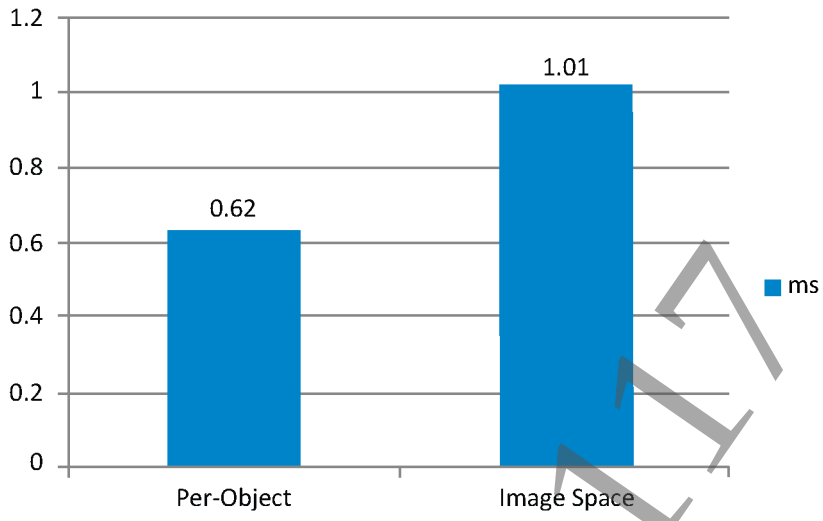
Although registers for primary/secondary attributes and temporary data are all part of the USSE's Unified Store, there is no way to force uniform parameters to be allocated from the bank of primary attribute registers. The purpose of this limitation is to avoid unwanted duplication of data in the unified store. Secondary attribute registers are shared between every instance of the same shader; conversely, primary attribute registers and temporary registers are unique to each individual instance of the program. Allocating uniform parameters from primary attribute registers would increase pressure on the USSE's Unified Store and could result in fewer shading threads in flight at any one time on the GPU. This could drastically impact performance.

A Note on Opaque-Only Scenes

Of course, one of the main motivations behind post-processing is to decouple shading cost from scene complexity. This is done by ensuring that the data transformations required for your post-processing effects are carried out only once for each pixel. However due to the tile-based deferred rendering (TBDR) architecture of the SGX543MP4+ graphics chip in PlayStation®Vita, this benefit is gained implicitly to some extent. If your scene contains only opaque objects, the additional calculations can be placed in the fragment shaders of your objects and the deferred architecture will automatically ensure that these calculations are carried out only once per pixel. This approach is not as effective if you have alpha blended objects in your scene or require access to information in neighboring pixels.

Figure 9 shows the performance of an opaque-only scene with the color correction calculations being performed in the fragment shader of the scene's objects and also with the same calculations being performed in image space.

Figure 9 Cost of Fragment Shading Is Kept to a Minimum due to TDBR Architecture



Impact of Buffer Sizes, Formats, and Surface Types

Figure 10 shows the performance off-chip color correction for different locations and layouts of the memory used for rendering surfaces. Rendering to surfaces in main memory does not generally carry a large penalty. Figure 11 shows how the color correction scales with different color surface sizes. Due to the simplicity of the color correction, shader changes in performance are proportional to the number of fragments shaded.

Figure 10 Cost of Fragment Shading for Off-Chip Color Correction with Dependent Reads to Surfaces with Different Memory Layouts and Located in Different Memory Pools

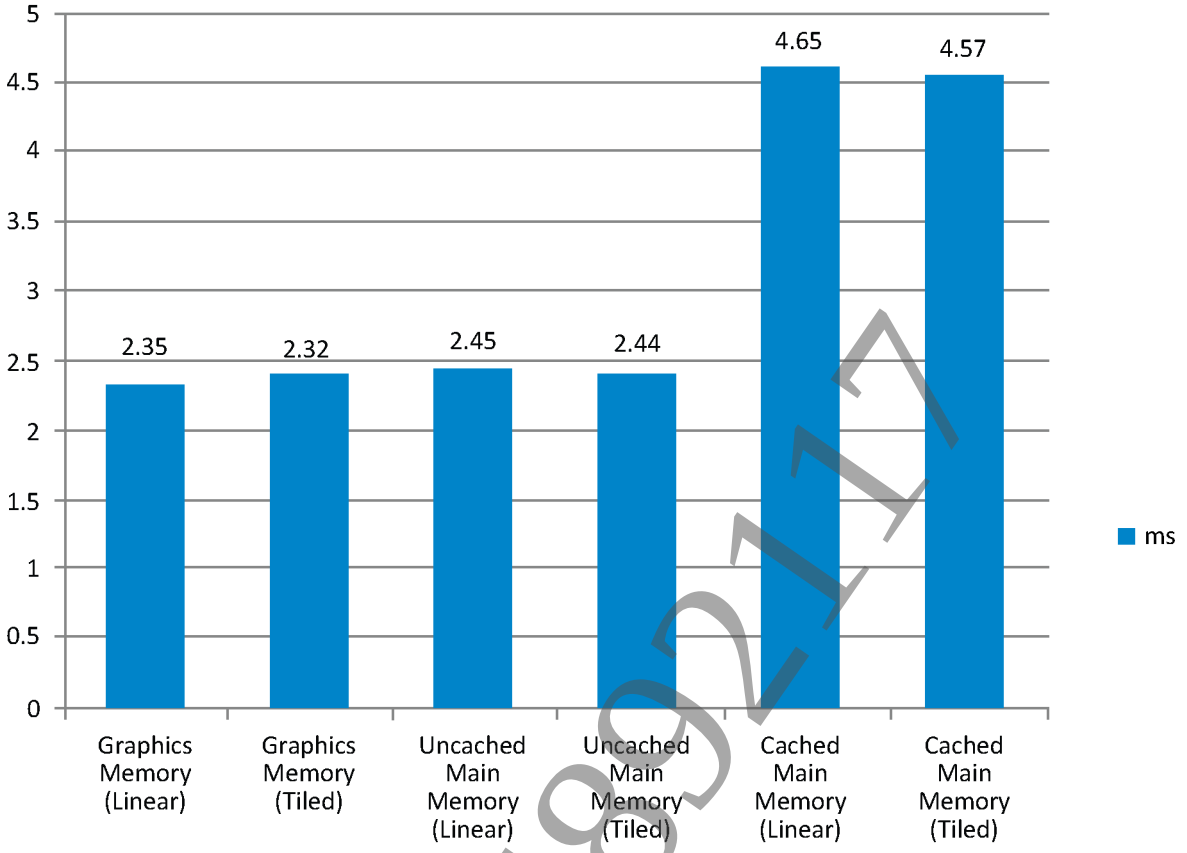
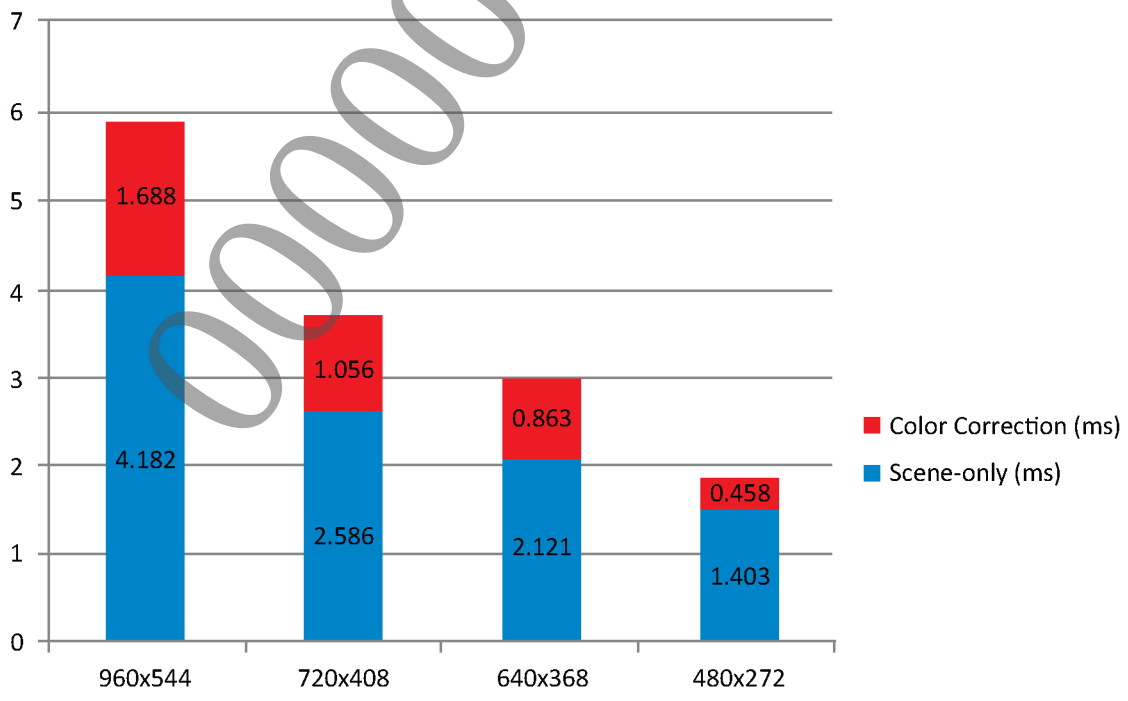


Figure 11 Cost of Fragment Shading for On-Chip Color Correction with Changing Back-Buffer Sizes

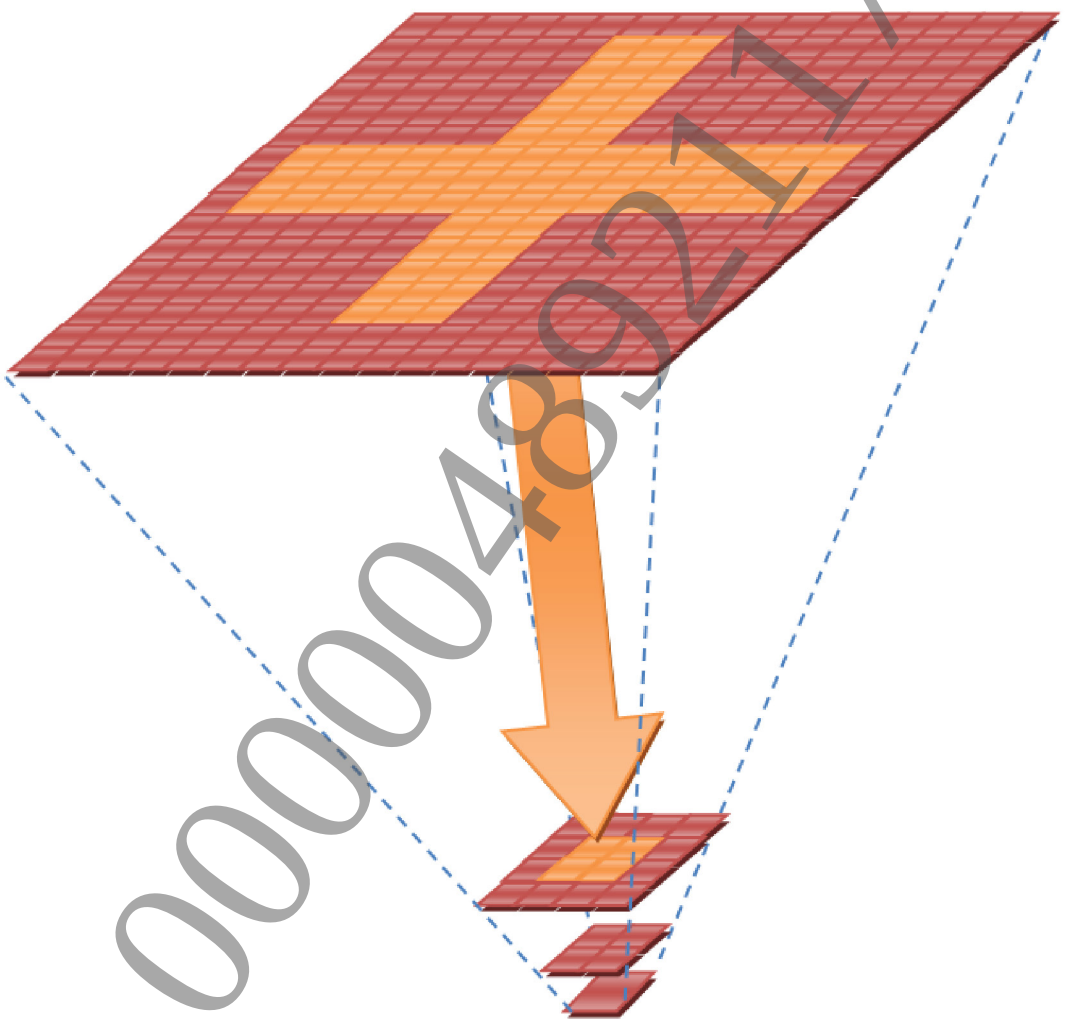


2 Downsampling

Overview of Technique

Generally speaking, *downsampling* is the act of reducing the sample rate of a signal. In the context of rendering, this term usually refers to computing a lower-resolution representation of a higher-resolution rendering surface, such as a color or depth buffer. This is desirable in a wide variety of contexts such as calculating average luminance (the result of which might later be used in a bright-pass filter), generating a mip-map chain for a GPU created texture resource, or performing more expensive post-processing techniques at a lower resolution. Figure 12 shows the downsampling of a higher-resolution color surface to a 1x1 texture.

Figure 12 Downsampling a Color Surface



The chapter discusses performing a single downsample to 25% of the original resolution. For a discussion about progressive downsampling for calculating a scene's average luminance, see Chapter 3, Bloom).

PlayStation®Vita Implementation Details

This section discusses important platform-specific implementation details that help accelerate downsampling on the PlayStation®Vita platform. The source code for this can be found at:

```
SDK/target/samples/source_code/graphics/tutorial_postprocessing/downsample.c
```

Downsampling to a Quarter Size

Downsampling Color

Because the cost of image post-processing tends to be proportional to the number of pixels that make up the image to be processed, it is often desirable to perform some of the more expensive techniques on a smaller surface that has been constructed by downsampling the signal present in the higher resolution surface. To perform downsampling, a quadrilateral that covers the entire screen is rendered to a render target that is 25% the size of the source surface. Each pixel in the output render target is computed by sampling the full resolution version with bilinear filtering; this is shown in Figure 13.

Figure 13 The Church Scene Downsampled to 25% Resolution

The performance of downsampling color surfaces in different memory pools, with different memory layouts, is shown in Figure 14. In these tests, the destination buffer was resident in graphics memory; avoid reading buffers from cached main memory. Figure 15 shows the performance for downsampling a color surface resident in graphics memory to other color surfaces resident in different memory pools, with different memory layouts.

Figure 14 Downsampling a 720x408 Color Surface to a Color Surface in Graphics Memory

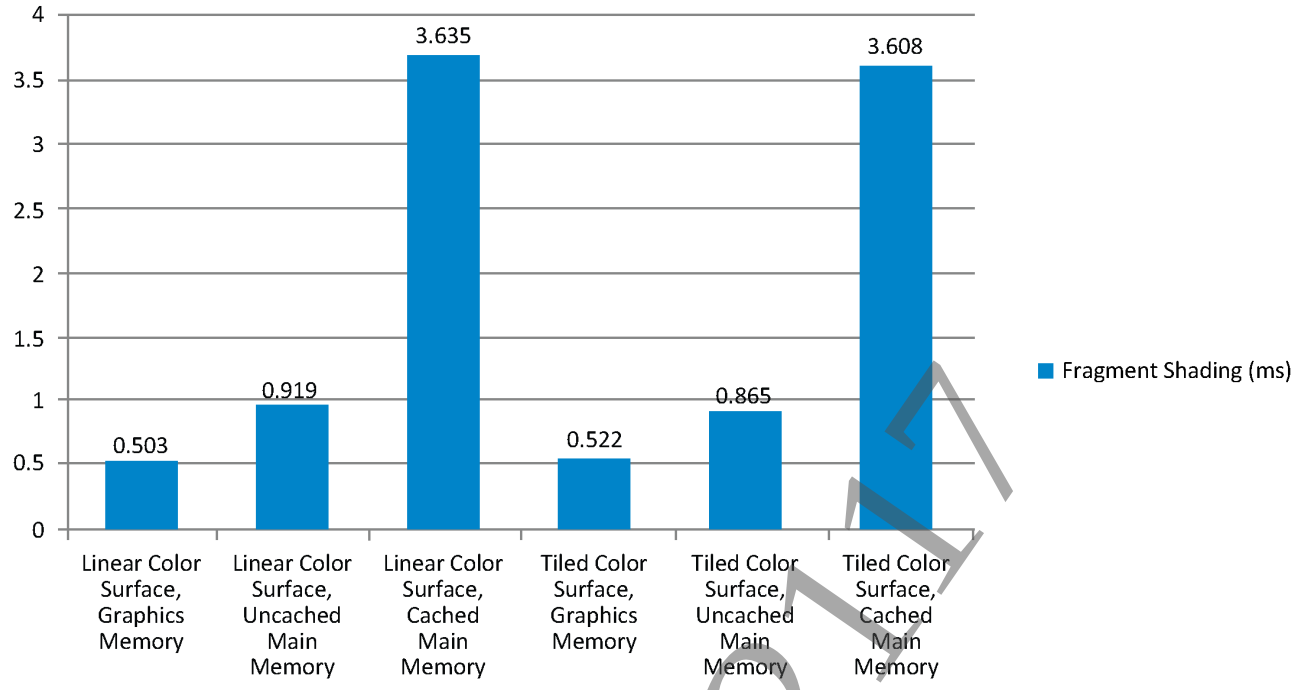
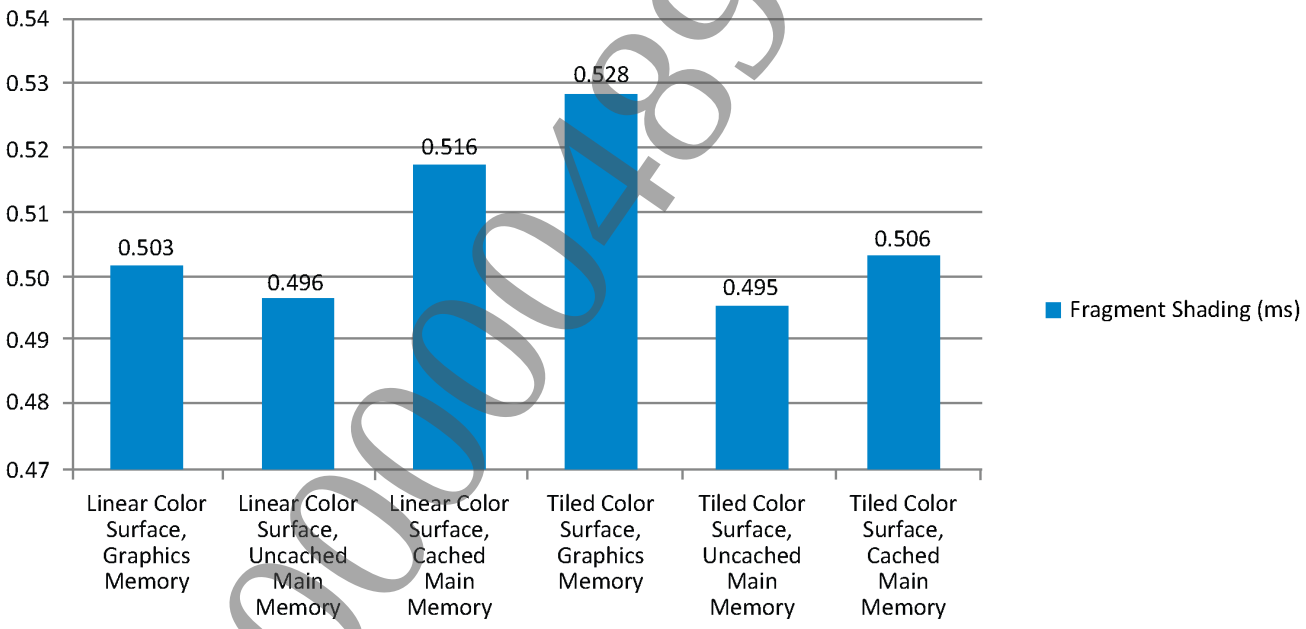


Figure 15 Downsampling a 720x408 Color Surface in Graphics Memory to Other Color Surfaces



Downsampling Depth

Color values are usually calculated by taking the average of the sampled pixels (for example, a bilinear filter can be applied to the nearest four pixels with equal weights). In contrast, the downsampling of depth surfaces requires calculating the minimum or maximum (depending on the direction in which the depth buffers' values increase) of a number of point-filtered samples.

Non-dependent texture reads should be preferred when attempting a depth downsample. Listing 7 shows how you can create a reasonably lean shader of only two instructions in length, which you can then use to downsample the depth buffer. You can create this shader by:

- Carefully controlling the packing of data in the primary attribute registers using the template-like syntax for the `tex2D` intrinsic function (avoiding the generation of extraneous `or` instructions to permute the data)
- Explicitly specifying the output register format for the shader

Listing 7 Cg Code for Selecting the Correct Depth Value During Downsampling

```
float2 depth0, depth1;
depth0.x = tex2D<float>(depthBuffer, texCoord0).r;
depth0.y = tex2D<float>(depthBuffer, texCoord1).r;
depth1.x = tex2D<float>(depthBuffer, texCoord2).r;
depth1.y = tex2D<float>(depthBuffer, texCoord3).r;
float2 result = min(depth0, depth1);
return min(result.x, result.y);
```

Performance figures for downsampling depth surfaces are similar to those for color surfaces; refer to Figure 14 and Figure 15 above.

Downsampling Depth and Color in a Single Pass

If your program requires the downsampling of both depth and color surfaces, you can optimize it by using fragment programs to write depth values to perform both the color and depth sampling in a single pass. Listing 8 shows the Cg code to achieve this. There is overhead involved in switching scenes to perform downsampling of color and depth buffers in separate passes. This situation could change with future improvements to the USSE microkernel.

Listing 8 Downsampling Color and Depth Buffers in a Single Pass

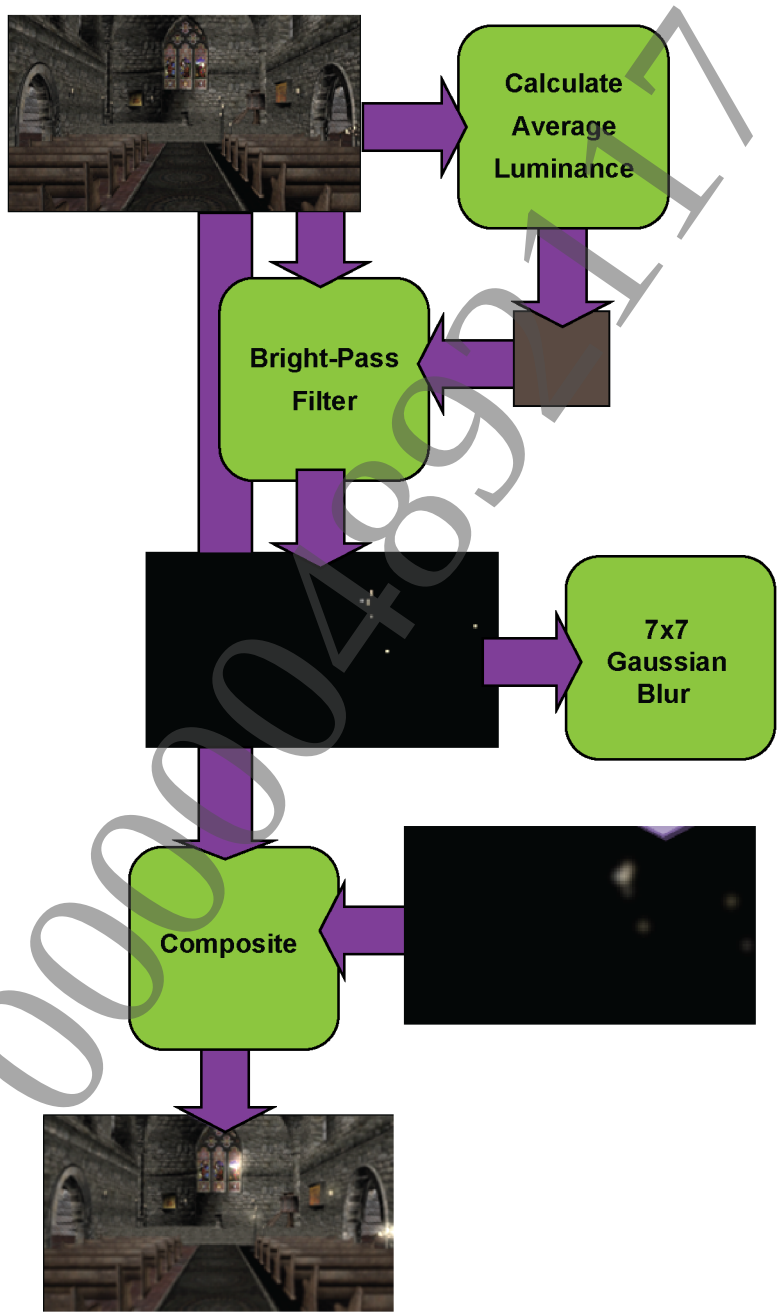
```
__regformat __nativecolor half4 main (float2 texCoord0 : TEXCOORD0,
                                      float2 texCoord1 : TEXCOORD0,
                                      float2 texCoord2 : TEXCOORD0,
                                      float2 texCoord3 : TEXCOORD0,

                                      uniform sampler2D color : TEXUINT0,
                                      uniform sampler2D depth : TEXUINT1,

                                      out float depth : DEPTH0) : COLOR0 {

    half4 color  = tex2D(color, texCoord0);
    float2 depth0, depth1;
    depth0.x = tex2D<float>(depth, texCoord0).r;
    depth0.y = tex2D<float>(depth, texCoord1).r;
    depth1.x = tex2D<float>(depth, texCoord2).r;
    depth1.y = tex2D<float>(depth, texCoord3).r;
    float2 result = min(depth0, depth1);
    depth = min(result.x, result.y);
    return color;
}
```

3 Bloom

Overview of Technique

Bloom is a very common technique used in many post-processing frameworks. Its goal is to emulate the phenomena of light diffraction through the non-perfect transmission medium of a camera's lens. Figure 16 shows the steps involved in computing bloom as a post-processing effect and shows the data flow between these steps.

Figure 16 Data Flow Diagram for Bloom Post-processing Effect

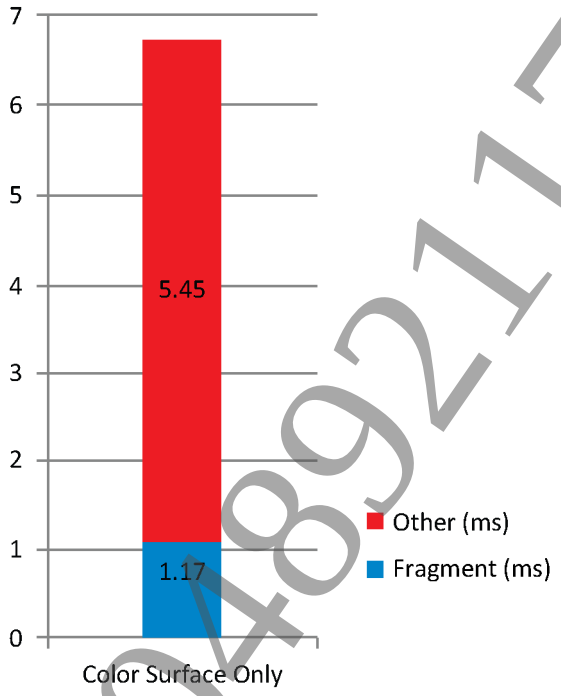


Calculating Average Luminance

Multi-scene Approach

The most common approach found on traditional stream GPUs is to use a number of render targets, with each target progressively smaller than the previous. Unfortunately this technique does not suit the SGX543MP4+ due to its use of multiple scenes. Figure 17 shows the timings for downsampling of a 720x408 color buffer (with no MSAA) to a 2x2 color surface carried out via eight scenes. The red part of the bar shows the amount of time that is attributable to the GPU being busy with other processing activities, including the microkernel running on the USSE (mostly during scene changes).

Figure 17 Performance of a Downsample to a 2x2 Texture Using Multiple Scenes



Sparse Sampling and Small Surfaces

In many game applications, the distribution of intensities in color data allows a reasonable approximation of the luminance in an image to be generated without considering every pixel. Although there is a risk that the high-frequency fluctuations in luminance can be missed by using a sparser sampling pattern, there are potentially large performances gains to be made by doing this. In the post-processing sample, downsampling occurs from the original full resolution rendering to a buffer 25% of the original size, and from there a further level of downsampling occurs to a 64x64 buffer.

The choice of 64x64 is not arbitrary and is selected in order to make the best use of the available processing elements in the GPU. The USSE microkernel will create USSE tasks for each tile in a color surface. Each task represents a single tile that is 32x32 samples in size. Each task is dispatched to a single USSE core where the tile is broken down into its constituent quadrants and from there dispatched to the each of the four pipes within that core (this is shown in Figure 18).

This scheduling strategy has implications when performing post-processing on small surfaces. Rendering to surfaces that cover less than a four tiles will compromise the efficiency of the SGX543MP4+ GPU significantly. Post-processing surfaces that are only a single tile or smaller result in only a single USSE core being used to perform the fragment shading work, with color surfaces that are 16x16 samples or smaller resulting in only a single pipe of a single core being utilized. Figure 19 shows a Razor trace demonstrating the use of parallelism when post-processing a color surface 64x64 samples in size.

Figure 18 Work for a Single 32x32 Sample Tile Parallelized over Four USSE Pipes in a USSE Core

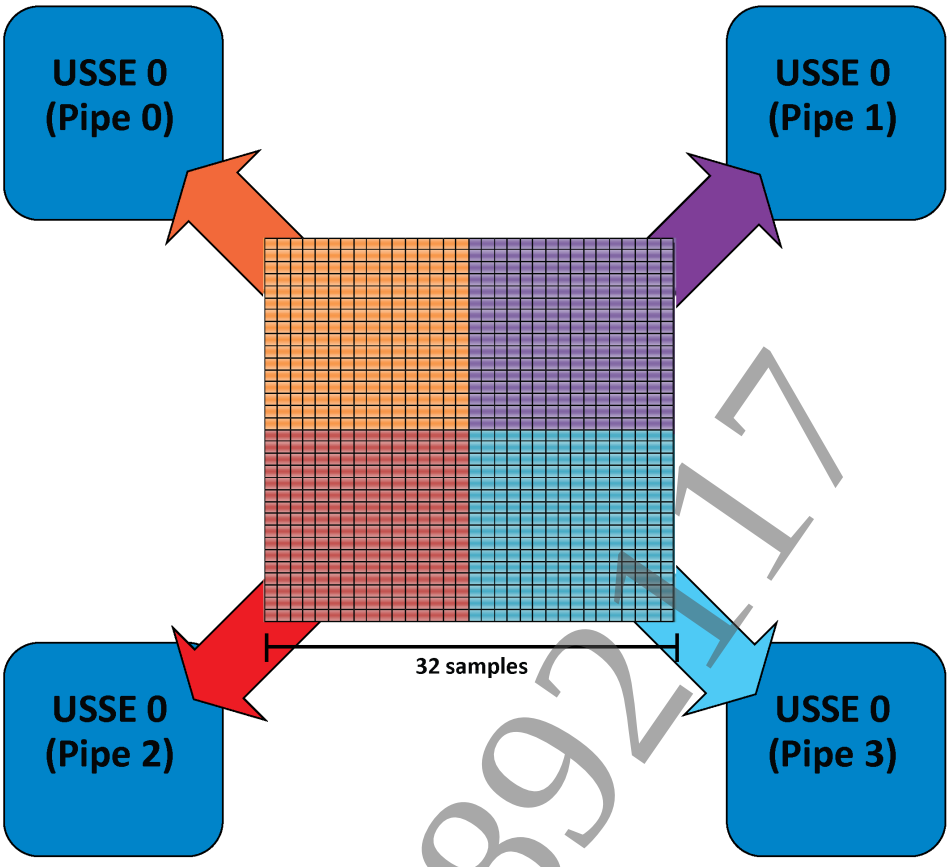
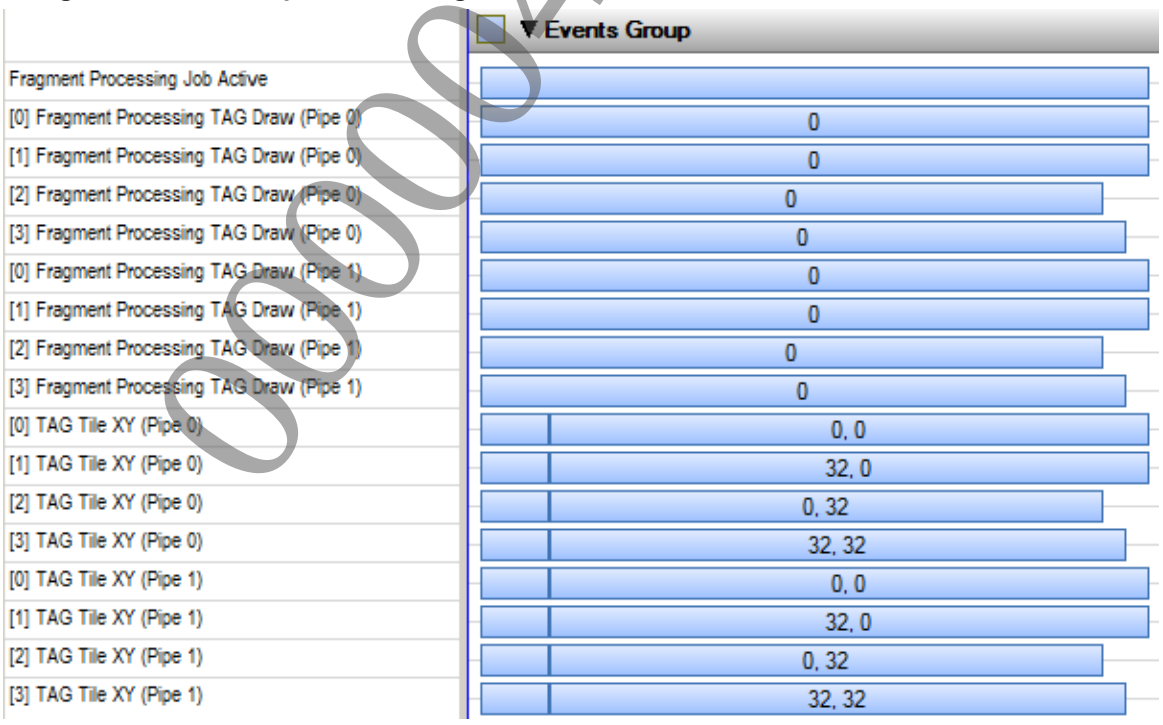


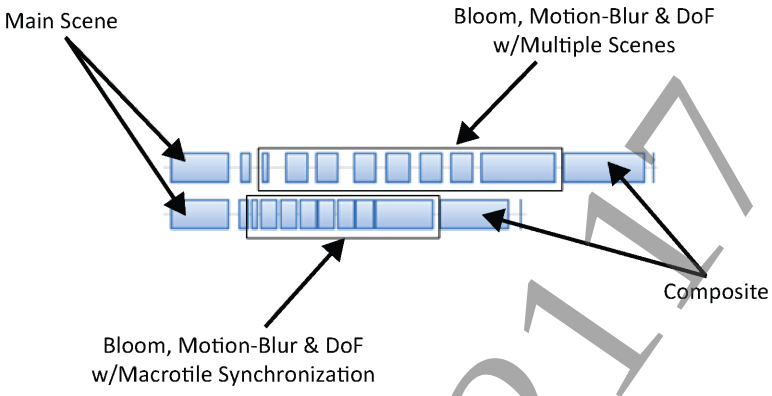
Figure 19 Razor Capture Showing Four Tiles Distributed Across the Four USSE Cores



Mitigating Scene Change Overhead

As shown in Figure 17, changing the scene many times can quickly become an expensive proposition. As of SDK 1.0 libGxm supports a new technique called *Macrotile Synchronization*. Macrotile Synchronization allows the color surface of the current scene to be *safely* aliased as a texture (and hence sampled) during the rendering of the same scene. While this affords the application the benefit of avoiding a costly change of scene there are a number of requirements that the application must adhere to in order to take advantage of this feature.

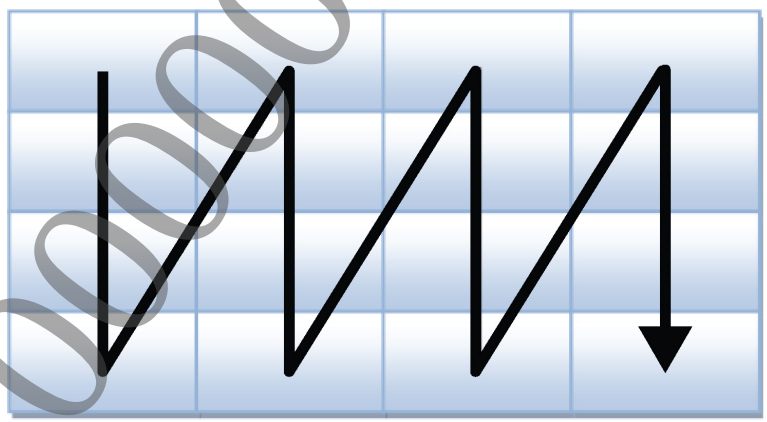
Figure 20 Performance is Improved Significantly Through the use of Macrotile Synchronization



What is Macrotile Synchronization?

In order to understand macrotile synchronization it is first helpful to understand the concept of the macrotile. The SGX543MP4+ subdivides the target color surface into tiles of a fixed size (32x32 samples) these tiles are then arranged into a higher-level grid pattern, each cell of which is known as a *macrotile*. Please refer to the *GPU User's Guide* for more details. Macrotiles are shaded in a fixed order, beginning with the top-left most macrotile and proceeding in columns from the left to the right as shown in Figure 21.

Figure 21 The Rendering Order of Macrotiles is Fixed in Hardware



Each core is assigned a list of tiles to shade. Since the cores are working independently of one another shading their respective tiles there is no guarantee that a core will not progress more quickly than its neighbors and begin shading tiles belonging to the next macro tile before shading on the previous macrotile is completed and the results written back to memory by the PBE.

Macrotile synchronization is simply a means of ensuring that the shading of a macrotile is completed and the results have been written back to memory before any of the cores may progress to tiles which constitute part of the next macro tile in sequence.

Setting Up Macro Tile Synchronization

To enable macrotile synchronization a flag must be set on the render target parameters when it is created. Along with the synchronization flag, the number of tiles in both the X and Y dimensions must also be specified, this is demonstrated in Figure 22.

Listing 9 Enabling Macrotile Synchronization on a Render Target

```
SceGxmRenderTargetParams params;
        /* set other flags here. */
params.flags = SCE_GXM_RENDER_TARGET_MACROTILE_SYNC
        | (tilesInX << SCE_GXM_RENDER_TARGET_MACROTILE_COUNT_X_SHIFT)
        | (tilesInY << SCE_GXM_RENDER_TARGET_MACROTILE_COUNT_Y_SHIFT);
```

Consideration must be given to the number of macrotiles required in both the X and Y dimensions. The maximum is four in each dimension (allowing for up to sixteen macrotiles in total), but, depending on the requirements of the application , less can be used if desired. The hardware supports two macrotile configurations, 2x2 and 4x4 - intermediate configurations are actually implemented in firmware by skipping empty macrotiles. This means that if more than two macrotiles are requested in either dimension then the 4x4 configuration will be selected by libGxm. If four or less macrotiles are required it is better to choose a macrotile layout that would result in libGxm selecting the 2x2 configuration as this will both save some memory from your application budget and also minimize the number of synchronization points that are required for the scene.

In order to constrain rendering to a single macrotile, viewports can be used. Figure 22 demonstrates how to set the viewport for a particular macrotile, that matches the default viewport set by libGxm, for a scene.

Listing 10 Setting the Viewport to Render to a Macrotile

```
sceGxmSetViewport(context,
        0.5f * (maxX + minX),
        0.5f * (maxX - minX),
        0.5f * (maxY + minY),
        -0.5f * (maxY - minY), /* negative y scale */
        0.5f,
        0.5f);
```

The main benefit of macrotile synchronization is the ability to sample previously rendered macrotiles of the color surface of the current scene , as a texture. If your color surface is linear then using linear strided textures can eliminate the need to calculate UV coordinates in the vertex shader, allowing standard normalized UV coordinates to be used. In addition, specifying the CLAMP address mode is advised to ensure that cache lines from neighboring macrotiles are not read during texture filtering. Figure 22 demonstrates how to create a strided texture and set the addressing mode to CLAMP.

Listing 11 Using Linear, Strided Textures can Simplify Vertex Shaders

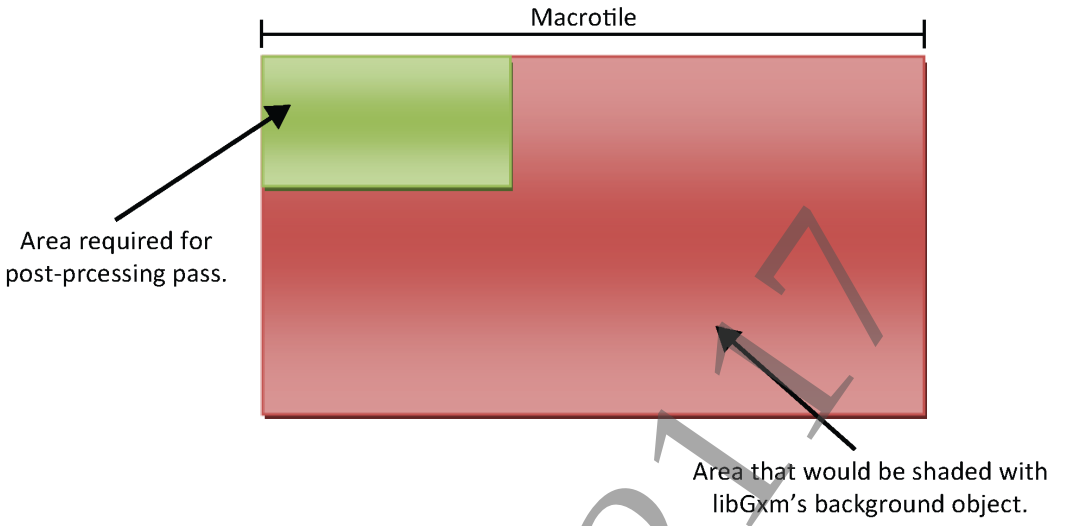
```
sceGxmTextureInitLinearStrided(&texture,
    ptrToSurface,
    format,
    widthOfMacrotile,
    heightOfMacrotile,
    strideOfFullBuffer);
sceGxmTextureSetUAddrMode(&texture, SCE_GXM_TEXTURE_ADDR_CLAMP);
sceGxmTextureSetVAddrMode(&texture, SCE_GXM_TEXTURE_ADDR_CLAMP);
```

Performance Tips when Using Macro Tile Synchronization

During post-processing it is common to have buffers with varying dimensions. When this occurs you may find that the full-screen quadrilateral, rendered for your individual post-processing, passes covers less real-estate than is afforded to you by a full macrotile, see Figure 22. For example, the post-processing

sample performs the steps of separable Gaussian blur passes at a lower resolution than that of a single macrotile. In these cases a small performance gain has been observed if the entire color surface is cleared with a very cheap shader (one that simply returns a constant color) before rendering post-processing passes into the surface.

Figure 22 When Fragment Coverage is low; Clear the Color Surface with a Cheap Shader



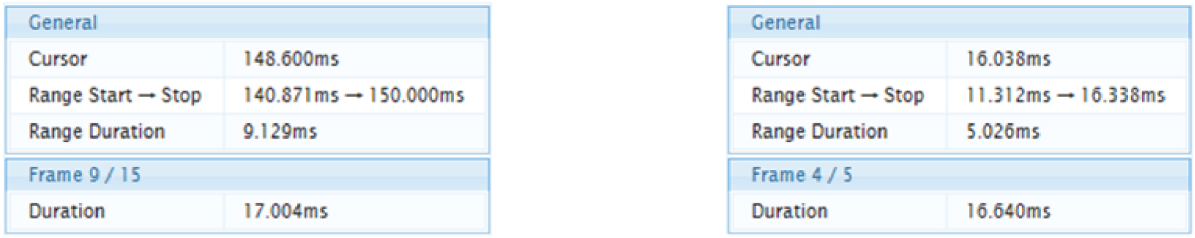
The reason for this is that the background object (maintained by libGxm) will actually be shaded for those fragments that are left untouched by the application draw calls. The shader used for this background object actually performs a non-dependent texture fetch for each fragment that it will shade. It is for this reason that we are able to improve performance slightly by performing a clear of the color surface with an even more minimal fragment shader. This is shown in Figure 23.

Figure 23 Custom Clear Shader Provided a Small Performance Gain

Performance Gains in the Post-Processing Sample

The significant gains in performance demonstrated in Figure 22 are almost entirely attributable to the elimination of the firmware overhead by keeping post-processing passes confined to a single scene. While offering significant performance improvements for post-processing, macrotile synchronization does introduce a small firmware overhead between macrotiles which currently stands at approximately 10% of the cost of a full scene change. For this reason it is not recommended that applications enable macrotile synchronization for scenes which do not require it.

Figure 24 Significant Performance Gains are made through Reducing Scene Changes



Post-processing Chain w/ Multiple Scenes

Post-processing Chain w/ Macrotile Synchronization

Restrictions When Using Macro Tile Synchronization

Finally, there are a number of restrictions to be aware of when using macrotile synchronization, currently they are as follows:

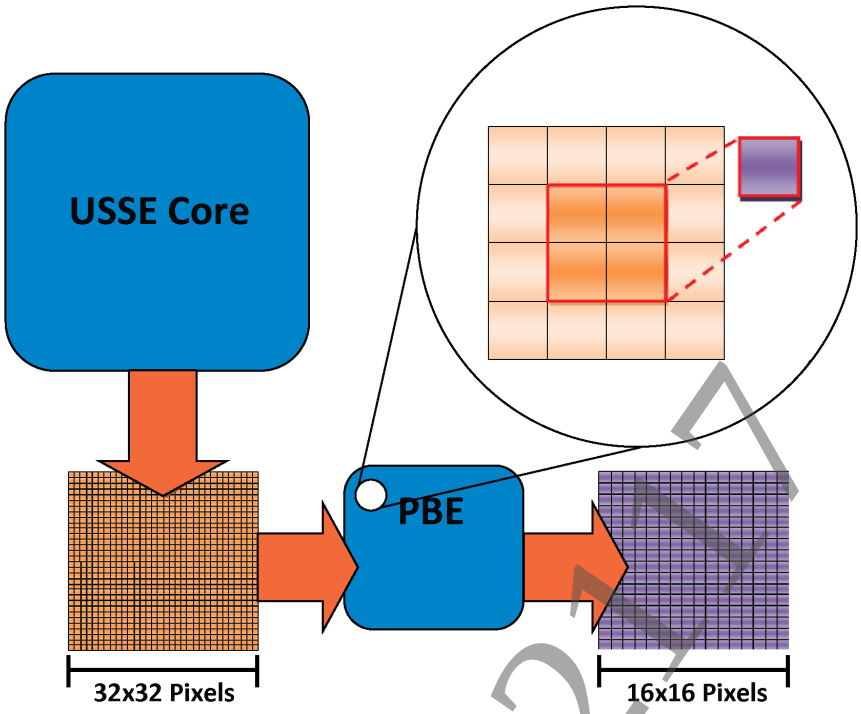
- A custom valid region cannot be used when rendering a scene using a render target that uses macrotile synchronization.
- MSAA cannot be used when using macrotile synchronization.
- The width and height of each macrotile must be a multiple of 128 pixels and cannot be larger than 1024 pixels.
- Color surface data for each macrotile must lay in separate 64-byte system level cache (SLC) lines.
- All primitives that lie in macrotiles 1 to N-1 must be drawn before any primitives that lie in macrotile N are drawn. This ensures that split scenes, due either to ring buffer pressure or partial rendering, do not change the visual results.
- Reading depth or stencil values from previous macrotiles should not be done as there is currently no synchronization to ensure that such values are not read before the depth or stencil writes have completed.

Some of these restrictions may be lifted in future releases of the SDK.

Taking Advantage of the PBE to Perform Additional Downscaling

As of SDK 0.990, libGxm supports both the use of multisample anti-aliased (MSAA) surfaces with no downscaling and the use of non-MSAA surfaces with downscaling. This means that a fragment shader can run on each pixel of a color surface and when the USSE core has finished shading each tile, the PBE can then be used to perform an extra level of downscaling. Although this may have a small run-time cost, using the PBE in this way is much faster than performing an extra level of downsampling by rendering another scene. The PBE downscaling process is shown in Figure 25.

Figure 25 Using the PBE to Perform Downscaling



The C code shown in Listing 12 shows how to create a color surface that will be downscaled by the PBE. For details on mixing programmable blending with multisample anti-aliasing, refer to the *Shader Compiler User's Guide*.

Listing 12 The PBE Can Be Used to Downscale a Color Surface, Even Without MSAA

```
sceGxmColorSurfaceInit(&buffer->colorSurface,
                       surFmt,
                       type,
                       SCE_GXM_COLOR_SURFACE_SCALE_MSAA_DOWNSCALE,
                       outRegSize,
                       width,
                       height,
                       width,
                       surfaceMemory);
```

Using the CPU to Accelerate Post-processing

The CPU is used to process color surfaces less than 64x64 in order to avoid GPU underutilization. In the post-processing sample code, the PBE is first used to perform an additional level of downscaling from the 64x64 color surface to a surface that is 32x32 pixels in size. The CPU then processes this buffer one frame later in order to calculate the final average luminance value to use for bright-pass filtering. Be careful when using this technique; in order to process buffers on both CPU and GPU safely, the color surfaces must be multi-buffered. Doing this ensures that the GPU and CPU are not attempting to read and write from the same buffer at the same time.

Reinhart (et al) proposes Equation 1 as a useful approximation to a scene's average luminance[1] (see Related Documentation and Other Resources), where n is the total number of pixels in the color surface, and δ is a small value required to avoid a singularity in case there are black pixels in the image being processed.

Equation 1 Calculating a Close Approximation of a Scene's Average Luminance

$$luminance_{avg} = \exp\left(\frac{1}{n} \times \sum_{x,y} \log\left(\delta + luminance\,(x,y)\right)\right)$$

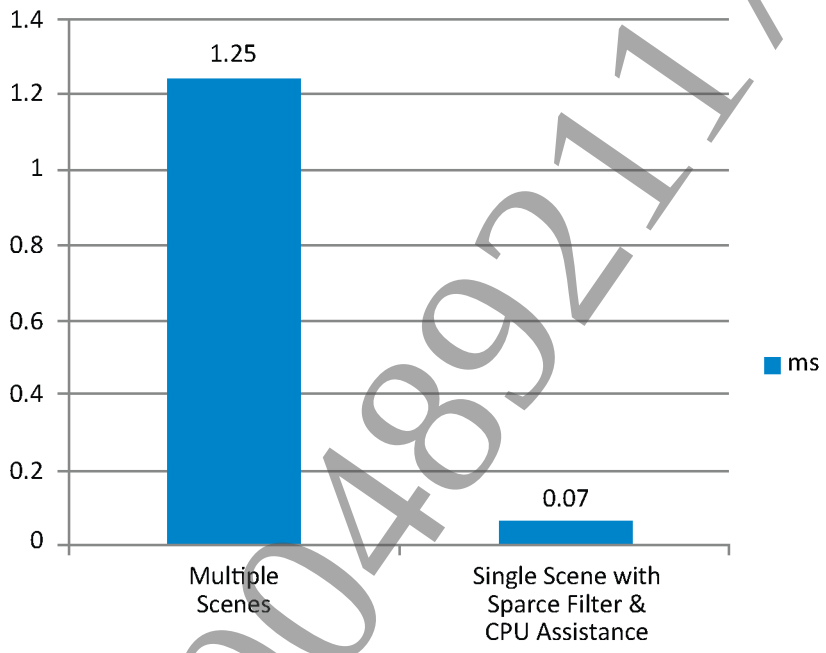
Source code that performs conversions between the SGX543MP4+ 10-bit floating-point and IEEE754 32-bit floating-point formats can be found in:

```
SDK/target/samples/source_code/graphics/tutorial_postprocessing/fp10.c
```

A Comparison of Technique Runtime Performance

Figure 26 shows the runtime performance of the two methods of calculating average luminance.

Figure 26 Performance of Different Approaches to Calculating Average Luminance



Bright-Pass Filtering

To isolate the "bright" portions of the image, a filter is applied to the original source image that will mask out areas of the image that fall below a particular brightness threshold (this value is primarily determined by the scene's average luminance as described in the preceding section). This technique is known as bright-pass filtering and was originally proposed by Kawase[3] (see Related Documentation and Other Resources) as part of his general solution for "Bloom". Source code for an implementation of a simple bright-pass filter is provided in Listing 13.

Listing 13 Cg Code for a Bright-Pass Filter

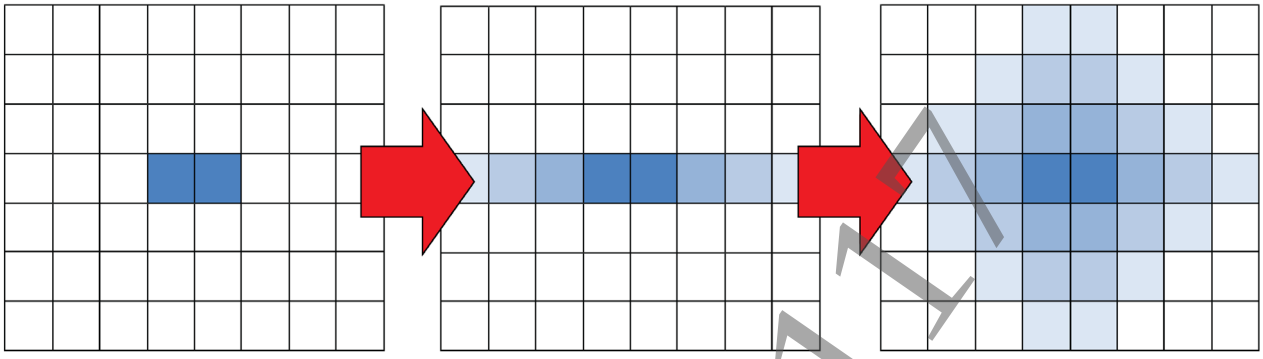
```
/** Apply bright-pass filter.
 */
half4 main (half2 texCoord : TEXCOORD0,
            uniform half avgLum,
            uniform sampler2D colorBuffer : TEXUINT0) : COLOR0 {

    half4 color  = tex2D(colorBuffer, texCoord);
    half lum = color.x + color.y + color.z;
    return ((lum*lum > avgLum) * color);
}
```

Gaussian Blur

A Separable Gaussian Blur

A Gaussian blur is separable and thus is typically decomposed into two distinct passes. The first pass will apply the blur horizontally only and write the results to a surface; the second then performs the same operation vertically, using the results of the first pass as an input. Figure 27 shows these steps.

Figure 27 A Gaussian Blur Performed via Two Passes



Performing Texture Fetches as Non-dependent Texture Reads

To avoid undesirable dependent texture reads, all mathematics pertaining to the calculation of UV coordinates for each texture fetch for the Gaussian blur should be performed in the vertex program and interpolated to the fragment program. Note that packing two sets of UV coordinate sets together into a single `float4` output for interpolation is undesirable; this is because performing 2D texture reads with anything other than the *x* and *y* components of a vector will force the texture read to occur on the dependent path. Listing 14 shows the fragment program used to perform a Gaussian blur with non-dependant texture reads.

Listing 14 The Fragment Program for Performing a Gaussian Blur

```
/* Gaussian blur weighting. */
static half blurWeights[] = {

        0.00443304,
        0.05400557,
        0.24203622,
        0.39905026,
        0.24203622,
        0.05400557,
        0.00443304
};

half4 main(half2 texCoord0 : TEXCOORD0,
           half2 texCoord1 : TEXCOORD1,
           half2 texCoord2 : TEXCOORD2,
           half2 texCoord3 : TEXCOORD3,
           half2 texCoord4 : TEXCOORD4,
           half2 texCoord5 : TEXCOORD5,
           half2 texCoord6 : TEXCOORD6,

           uniform sampler2D colorBuffer : TEXUINT0) : COLOR0 {

        half4 sample = (half4)0.0;
        sample += blurWeights[0] * (half4)tex2D(colorBuffer, texCoord0);
        sample += blurWeights[1] * (half4)tex2D(colorBuffer, texCoord1);
```

```
            sample += blurWeights[2] * (half4)tex2D(colorBuffer, texCoord2);
            sample += blurWeights[3] * (half4)tex2D(colorBuffer, texCoord3);
            sample += blurWeights[4] * (half4)tex2D(colorBuffer, texCoord4);
            sample += blurWeights[5] * (half4)tex2D(colorBuffer, texCoord5);
            sample += blurWeights[6] * (half4)tex2D(colorBuffer, texCoord6);
            return sample;
}
```

Figure 28 A Frame with “Bloom” Applied

4 Motion Blur and Depth of Field

Motion Blur

Overview of Technique

Often the first step when trying to compute motion blur for a scene is to calculate a velocity for each pixel. This must be per-pixel because the velocity will vary depending on what the focal point of the scene is. Consider the example of a camera accelerating in a circular motion around a particular object, but always remaining focused on it. The viewer would expect the object to remain more or less in focus, with the peripheral areas of the scene becoming more blurry as the camera's speed increased.

Figure 29 A Scene Rendered with Motion Blur

Reconstructing View Space Positions from Depth Buffer

Calculating the velocity of a pixel is achieved through a technique called reprojection. To calculate a velocity in screen space, each pixel is back-projected into view space and then reprojected using the last frame's projection matrix. This velocity can then be multiplied by a scaling factor to achieve the desired level of motion blur for each pixel. Reconstructing a view space z position from a depth buffer value can be achieved through the Cg code shown in Listing 15.

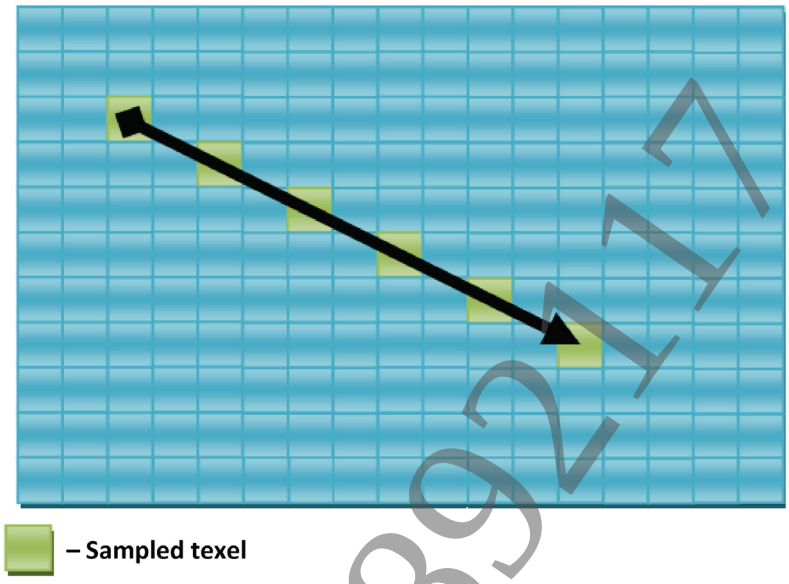
Listing 15 Cg Code to Reconstruct a View Space z Value from a Value Read from the Depth Buffer

```
/** Move from post-projection space to view-space.
    @param depth [in] Value from depth buffer.
    @return View space depth value.
 */
static float convertDepthToViewSpace(const float d) {

    return -((g_zNear*g_zFar)/(d*(g_zFar-g_zNear)-g_zFar));
}
```

Blurring Along the Velocity Vector

After calculating a velocity vector for each pixel, samples from the color buffer can then be taken at constant intervals along this vector and blended together in order to produce the color of the final pixel of a motion blurred fragment. This is shown in Figure 30. The sample interval will depend on the number of texture fetches that you plan to blend together to produce the final color of the fragment. Typically as the number of samples that are taken increases, so to does the quality of the final image. The impact that the number of samples has on performance is discussed later in this chapter.

Figure 30 Samples from the Color Buffer Are Taken Along the Velocity Vector for Each Pixel



Note: The technique described here will not produce motion-blur effects in scenes that have a static camera and moving objects.

Depth of Field

Depth of field occurs in real-world photography (and indeed in the human eye) because the pinhole of a camera (or pupil in the human eye) is not infinitely small. As a result, rays of light that do not converge exactly to a single point on the camera's film (or the eye's retina) will instead fall onto a larger area of the film (or retina). The size of this area is dependent on the distance of the object from the ideal plane of focus as well as the size of the lens in a camera or the human eye.

Calculating the Circle of Confusion

The circle of confusion is calculated by evaluating the equation shown in Equation 2[3] (see Related Documentation and Other Resources). However, simplifications can be made to accelerate runtime performance. Refer to the Cg source code in the post-processing sample for details.

Equation 2 Evaluating the Circle of Confusion

$$CoC = \left|\left(\frac{a \times (f \times (o - p))}{o \times (p - f)}\right)\right|$$

where:

$a = aperature,$

$f = focal\ distance,$

$o = object\ distance,$

$p = plane\ in\ focus$

When the circle of confusion is calculated, multiple samples are taken, each of which lies within the radius of this circle from the current fragment. The samples are then averaged to produce a final color for the fragment. Figure 31 shows the results.

Figure 31 A Scene Rendered with Depth of Field Effect

Limitations and Artifacts

The method presented here has some artifacts that are common to techniques that rely on reverse-mapping the z-buffer[2] (see Related Documentation and Other Resources). Typically these artifacts occur at places in the image where there are sharp depth discontinuities, as shown in Figure 32. Color information from the in-focus area behind the tapestry bleeds into the tapestry as blur is applied to simulate it being out of focus. Fortunately, the artifacts are difficult to notice and are not so unsightly as to render the technique unusable.

Figure 32 In Focus Parts of the Image Bleed into Out of Focus Areas

Additionally, the technique described relies on scaling points in a Poisson filter by the circle of confusion in order to apply more or less blur depending on the pixels' distance from the plane in focus. Due to the nature of this type of filter, this can mean that areas of the image that are far out of focus (and have a large circle of confusion) can become noisy; this is shown on the left side of Figure 33. You can mitigate this

problem to some extent by reducing the aperture size, although this will reduce the overall blurriness of out-of-focus portions of the scene; this is shown in the right side of Figure 33.

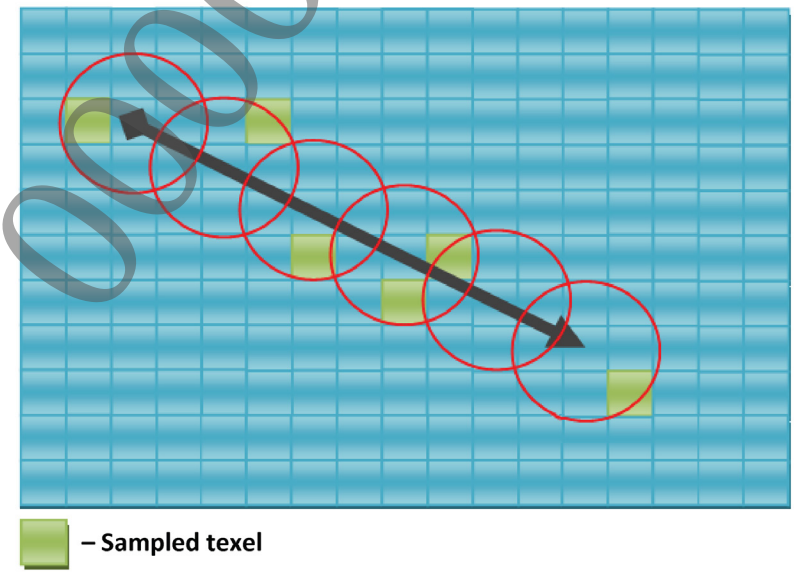
Figure 33 Areas That Are Far Out of Focus Can Appear Noisy and Exhibit Ghosting Artifacts

Applying Circle of Confusion During Motion Blur

Both techniques described in this chapter rely on blurring the underlying image to achieve the desired look. As a result they share similar setup steps and input data. You can take advantage of this by combining the two techniques into a single pass and applying the circle of confusion computed for a pixel's depth of field at each step along the velocity vector, as shown in Figure 34. Doing so has a number of performance advantages:

- The number of texture fetches required for both techniques is reduced to the number needed for only one of the techniques.
- The depth buffer needs to be accessed once per pixel, rather than accessing each pixel twice (once for each technique)
- USSE operations that are common to both techniques (such as blending colors) can be shared by both techniques.
- Only a single texture must be accessed during the composite phase.

Figure 34 The Circle of Confusion Is Applied at Each Step Along the Motion Vector



Resolution and Texture Fetch Considerations

Figure 35 and Figure 36 show the changes in performance, with increasing resolution and texture fetches, for the combined motion blur and depth of field implementation.

Note: Be careful when interpreting Figure 35. Each step along the horizontal axis does not represent a linear increase in the number of fragments being shaded.

Figure 35 Combined 8-Texture Fetch Motion Blur and DoF with Increasing Resolution

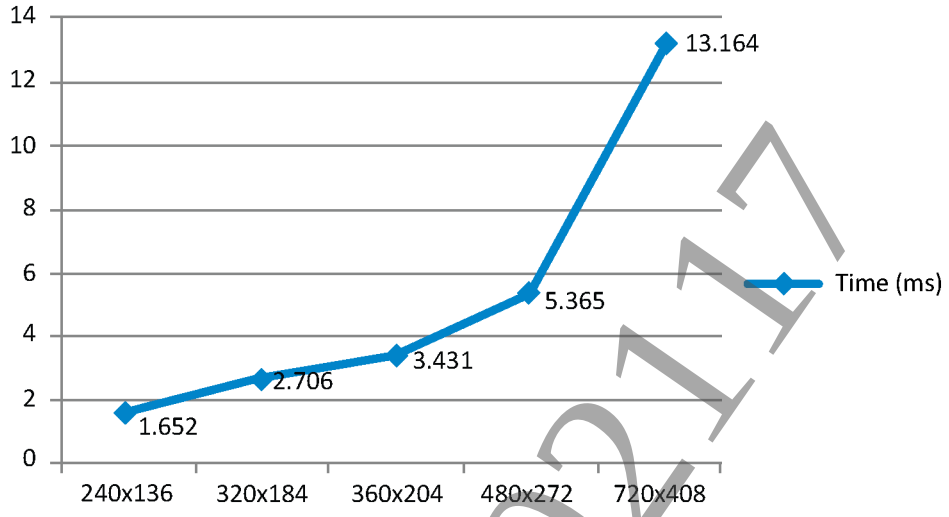
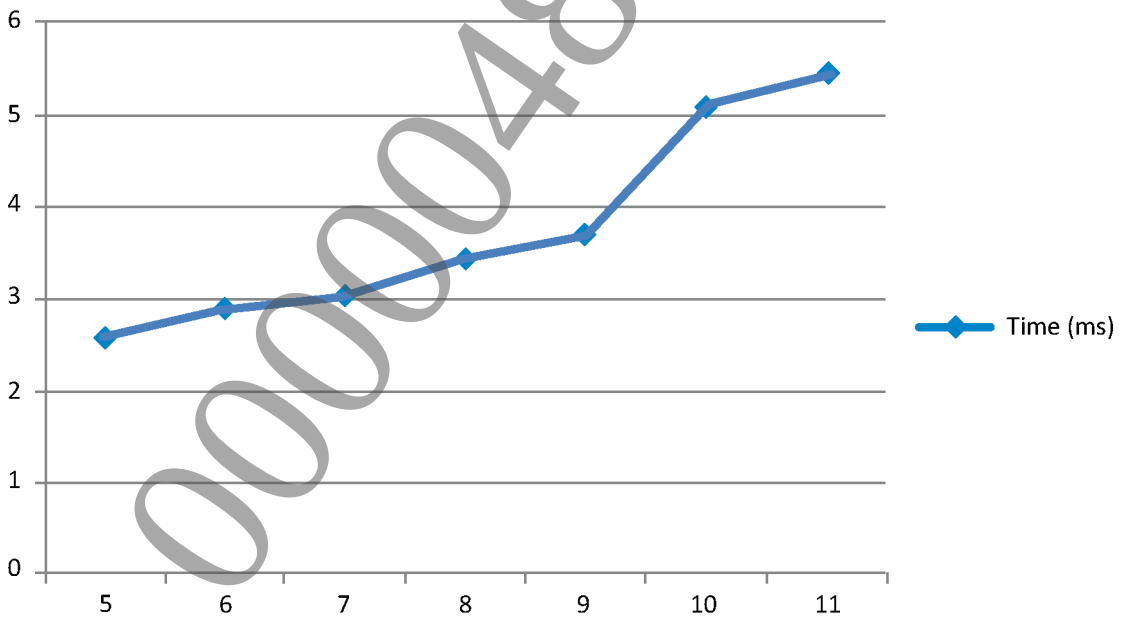


Figure 36 Combined Motion Blur and DoF (360x204) with Increasing Number of Texture Fetches*



* As of SDK 0.995, the steep drop in performance between 9 and 10 texture fetches no longer occurs due to improvements in the psp2cgc's register coloring algorithm. This figure is nevertheless preserved here to demonstrate a performance problem that will still occur in some circumstances where register pressure is high. It is advisable to monitor your shader's register utilization using psp2shaderperf.

Compositing

Because the motion blur and depth of field techniques are performed at 25% of the original resolution, you cannot simply display the results to the viewer without losing a significant amount of detail from the original rendering of the scene. Your program must complete a final compositing step, applying the effects that were computed to the original rendering of the scene. To do this, the alpha channel of the motion blur/depth of field color surface is used to store the total blur amount at each pixel. This value is used to linearly interpolate between the full resolution rendering of the scene, and the downsampled version that has been blurred to simulate the physical properties of the camera. The Cg in Listing 16 shows how the final image is composited using this value.

Listing 16 Cg code to Composite Motion Blur/Depth of Field with a Full-Resolution Rendering of the Scene

```
/** Fragment shader for final frame composite.
 */
half4 main(half2 texCoord0 : TEXCOORD0,

           /* samplers: */
           uniform sampler2D offscreenBuffer,
           uniform sampler2D motionBlurBuffer,
           uniform sampler2D bloomBuffer) : COLOR0 {

    const half4 color = tex2D<half4>(offscreenBuffer, texCoord0);
    const half4 blur  = tex2D<half4>(motionBlurBuffer, texCoord0);
    const half4 bloom = tex2D<half4>(bloomBuffer, texCoord0);
    const half4 mixedColor = lerp(color, blur, saturate(blur.w*blurScale));
    return mixedColor + bloom;
}
```

5 Recommendations

This section provides recommendations for optimizing the performance of post-processing techniques on PlayStation®Vita's GPU.

Minimize the Number of Scenes

Scene changes are currently expensive. Many post-processing effects will naturally fit well together (see Chapter 4, Motion Blur and Depth of Field), so try to ensure compatible techniques are implemented together where possible to reduce the number of scene changes. By making use of render to macrotile (see Chapter 3, Bloom), the cost associated with changing scenes can be greatly reduced.

Use Non-dependant Texture Fetches

Use non-dependent texture fetches where possible even if this means interpolating many texture coordinates. Because post-processing has very minimal geometry requirements, interpolating uniforms (rather than setting them directly on the pixel shader) can often yield a performance gain in fragment processing. Check your disassembly for unexpected `mov` instructions. Be careful when packing texture coordinates into registers; any accesses made to 2D textures with anything but *xy* components of the varying parameter will cause a dependent texture fetch. (Texture fetches carried out via tex2Dproj using the *xyw* components of the texture coordinate are also non-dependent.)

Verify the Disassembly of Your Shaders

Carefully monitor disassembly of your USSE programs, particular your fragment programs. The cost of extra instructions quickly adds up. Some things that are free on other architectures are not free on SGX (for example saturation). Keep a close eye on extraneous format conversions which will result in unwanted packing instructions. The psp2shaderperf tool will help you keep a track of your shader's performance.

Prefer Half-Precision Formats in Most Cases

In general, using the half-precision floating-point types will make shader perform significantly faster. This is the case for three reasons:

- It will reduce the number of registers required by your shader.
- It will (usually) reduce the number of format conversion instructions.
- 16-bit floating-point values can be used on all four SIMD lanes, while some 32-bit vector operations are limited to two lanes.

It is also advisable to pass uniforms to the fragment shader in half-precision types to avoid making the GPU do the conversion. Be careful in cases where the use of half-precision floating-point values would be detrimental to image quality; this is typically true with post-processing techniques such as motion blur or depth of field, where back-projecting depth buffer values is a key part of the technique.

Use the 64-bit Output Register Format

64-bit buffers seem to have negligible impact on performance and, if programmable blending is used, can help to greatly reduce packing instructions for typical post-processing operations. Consider using a 64-bit output register format if your post-processing shaders are performing many format conversions in order to interact with uniform parameters.

Take Advantage of the GPU's Architecture

Take advantage of TBDR by drawing a full-screen quadrilateral with programmable blending at the end of a scene to apply extra math to every pixel. This is especially true for scenes with alpha blending.

Use the PBE for Additional Downscaling

As of SDK 0.990, the PBE can perform MSAA downscaling even on buffers which do not have MSAA enabled. This can be exploited to get an extra level of downscaling without the cost of an extra scene. This can be particularly useful in situations where progressive downscaling is required (see Chapter 3, Bloom).

Avoid GPU Underutilization

Rendering to surfaces that are less than four tiles in size will impact the efficiency of the GPU. The CPU can be useful when processing smaller buffers, but be conscious of synchronization. Double or triple buffering is required to eliminate contention for resources.

Be Careful When Using Swizzling

The hardware can replace individual SIMD lanes with constants such as 0, 0.5, and 1 in specific circumstances. This can often be used to improve vectorized code. However, not all combinations of `xyzw` swizzles are supported by all instructions, so be careful that the compiler is not forced to waste an instruction to simply swizzle an operand.

Do Not Initialize Unused Channels

It is better to leave unused channels of your output buffer uninitialized rather than writing a known value to them.

Carefully Monitor Your Shader's Register Usage

Carefully evaluate the impact of increasing the number of primary and secondary attribute registers and temporary registers. When register pressure reaches a certain level, there appears to be a steep performance drop. This is currently being investigated.

Verify That the Downscaling of Buffers Provides a Performance Gain

In some cases, it may be quicker to simply access full resolution buffers from post-processing effects rather than to downsample them and thereby incur the penalty of a scene change. Verify that the gains made from reading from a downsampled version of the input buffer are significant enough to justify both an additional scene change and the work required for downscaling.

Prefer Buffers with a Linear Data Layout That Are Resident in Graphics Memory

Graphics memory is quicker than main memory for both reading and writing. If you are putting buffers into main memory, ensure that you use uncached main memory. Linear buffer layouts offer a small performance increase against their tiled counterparts.