HDR Rendering Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

This document guides you through the steps required to process some of the High Dynamic Range (HDR) rendering effects on the PlayStation®Vita hardware. In addition to the R8G8B8A8 format, the 64-bit F16F16F16F16 and 32-bit pseudo HDR formats are now available.

This document gives an overview of HDR rendering and the various effects visible, presents a walkthrough of the steps involved for rendering those effects, and gives an example of the code implementation.

Figure 1 HDR Rendering and Effects

(a) Indoor Scene

(b) With Shadow

Conventions

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

Hyperlinks

Hyperlinks are used to help you to navigate around the document. To allow you 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.

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:

```
m_targetBufferData[idx]);      // pointer to the surface data.
```

Errata

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

1 Introduction

This chapter provides an introduction to HDR rendering and the related effects that can be achieved in scenes where HDR rendering is used.

HDR Theory

It is easy to distinguish a real scene from one that is reproduced on a display screen, mainly because of the display screen's low dynamic range. This range is usually based on 8 bits of color precision representing color and light intensities. This is much lower than real-world scenes where the levels of lighting vary significantly; it is also lower than the range that can be perceived by the human visual system. Using a low range results in limited scenes that are rendered showing only standard average lighting, omitting certain high-intensity effects.

The purpose of the HDR rendering algorithm is to render scenes displaying a wide range of luminance values, thus creating lighting effects that add an extra level of realism to the scene. This requires rendering the scene with higher precision; the scene must be represented by more than 8 bits per color component to display very high brightness. One way to achieve this is by rendering to a floating-point (FP) render target that has a 16-bit or 32-bit precision instead of the usual 8-bit render target (A8R8G8B8). Usually, a 16-bit FP render target (F16F16F16F16) provides excellent results because textures store higher color values, resulting in precise luminance calculation. However, for reducing memory footprint and bandwidth requirements an A2R10G10B10 format is sometimes preferred.

Common HDR Effects

This sample implements some common HDR effects that are perceptually visible in bright areas and that display the wide dynamic range of the scene, such as:

- Bloom/Glare effect – this occurs when there is a high-intensity light source, where the light appears to be spreading out into the surrounding area.
- Star/Streak effect – a bright light appears to be spreading in a star form.
- Tone mapping – a technique to compress the color values to the display screen's low color range.

These effects are rendered by applying post-process steps performed on FP render targets that constantly store a wide range of color values and preserve these HDR characteristics of the scene. The next chapter gives an overview of the procedure implemented in this tutorial along with algorithm details for applying these effects.

2 HDR Rendering Algorithm

This chapter gives an overview of the procedure implemented in this tutorial and algorithm details for applying various effects.

Overview

Unlike Low Dynamic Range (LDR) scenes, which have color components limited to the range [0, 1], the HDR scene handles a higher range, thus correctly displaying bright and dark regions of the scene. This capability can be efficiently utilized to apply certain visual effects to high-luminance areas.

- The high-luminance areas (high-intensity light sources or specular highlights) can be extracted (Figure 3b).
- These areas can then be blurred to give a bloom effect (Figure 3c).
- Additional processing can create a star-like effect (Figure 3d).
- The two effects can be merged into the original source image in the back buffer, which is then tone-mapped before outputting the final image (Figure 3e).

Original Scene

The sample renders a cube map for the background, along with a model. The model is rendered to reflect the surrounding cube map environment, which together with the contribution of external lighting sources creates a HDR output. The reflectivity of the model mesh and the intensity of the external lights are adjustable.

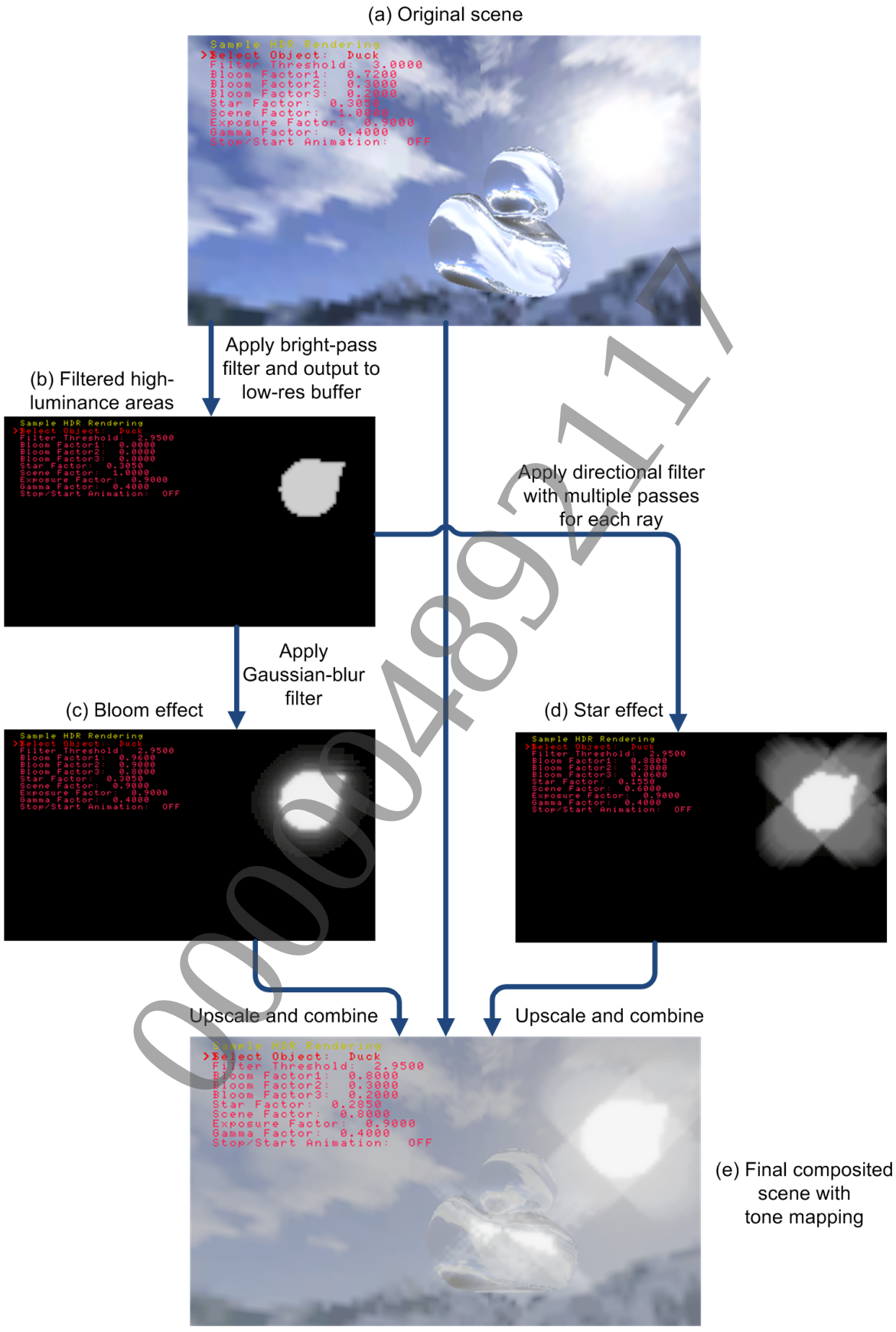
Figure 2 Rendered Original Scene with Environment Map and Lit-up Model

(a) (b)

(a) Rendered cube map for background

(b) Model rendered with environment map and external lighting

Figure 3 Rendered Cube Map and Model with Environment Map and External Light Contribution with Encoding



Downscaled Buffer and High-Luminance Filter

Rendering scenes with HDR effects requires multiple texture fetches; therefore it is important to reduce the number of fetches made during processing of the intermediate render targets. One way to speed up rendering is to process a downscaled version of the original rendered scene.

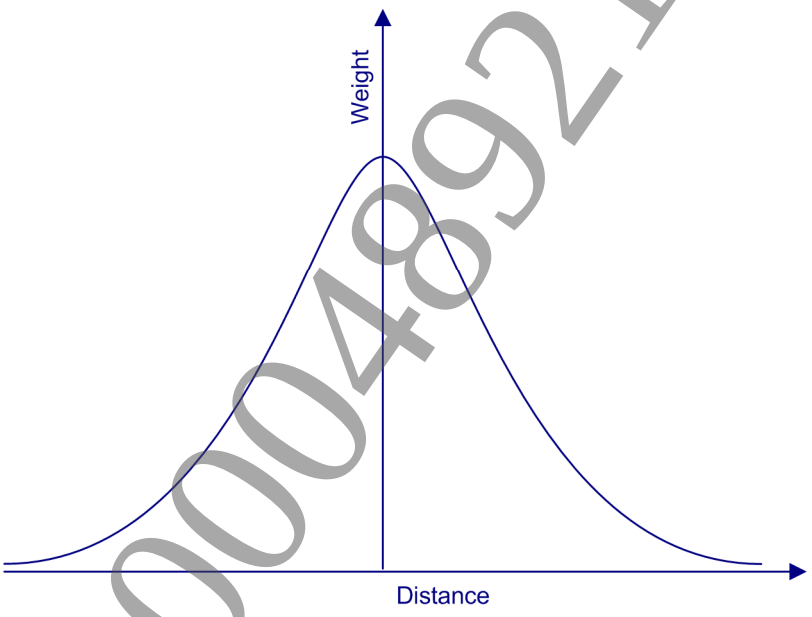
This step is performed when applying a high-luminance filter to determine high-intensity areas in the scene that are likely to show HDR characteristics. Any areas having a higher luminance than a user-set threshold are output to the low-resolution buffer using bilinear filtering.

Bloom Effect

To achieve a bloom, a wide filter is applied that falls off from the center of a bright source, thus spreading out intensity to pixels around the source. The bloom pass used in this sample is a two-pass separable horizontal and vertical Gaussian blur of the high-luminance filtered scene. The filter calculates the weight of the circular blur based on the distance from the center pixel irrespective of direction, which is represented by the Gaussian filter shown in Figure 4.

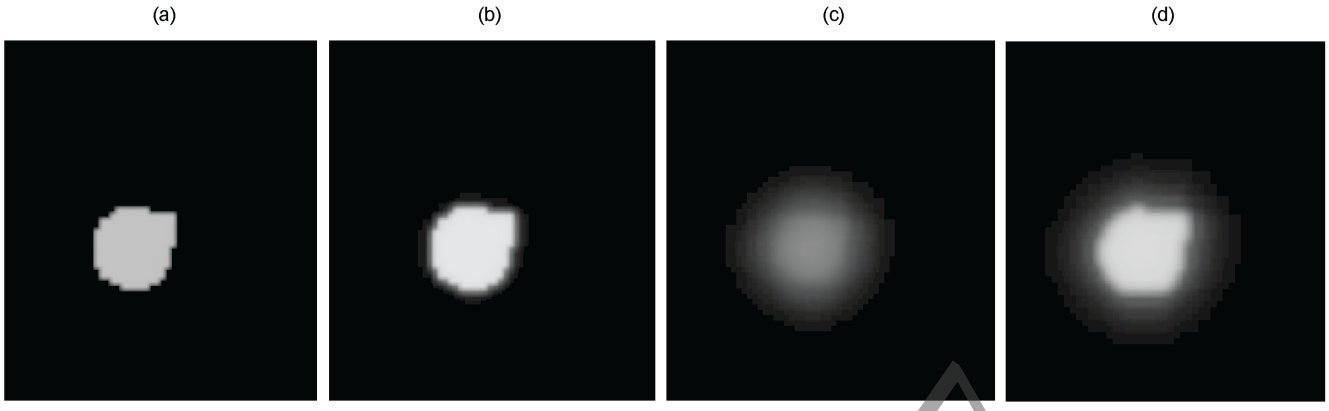
Figure 4

$$bloomWeights[n] = e^{(-n*n)/(distance*distance)}$$



- The texture is first blurred along the horizontal direction, and then the horizontally blurred texture is blurred along the vertical direction to give a smooth blur effect.
- The blur size is varied by changing the filter radius and using the new set of Gaussian weights along each direction (Figure 5c). Compositing the Gaussian filters of varying radii produces a larger and sharper bloom (Figure 5d).
- For the final scene rendering, the bloom textures are scaled to the size of the back buffer using bilinear filtering, and then added directly to the output of the scene with bloom weights applied.

Figure 5



(a) No filter
(b) Filter with small radius
(c) Filter with large radius
(d) Final image achieved by merging the different-sized filters

Star Effect

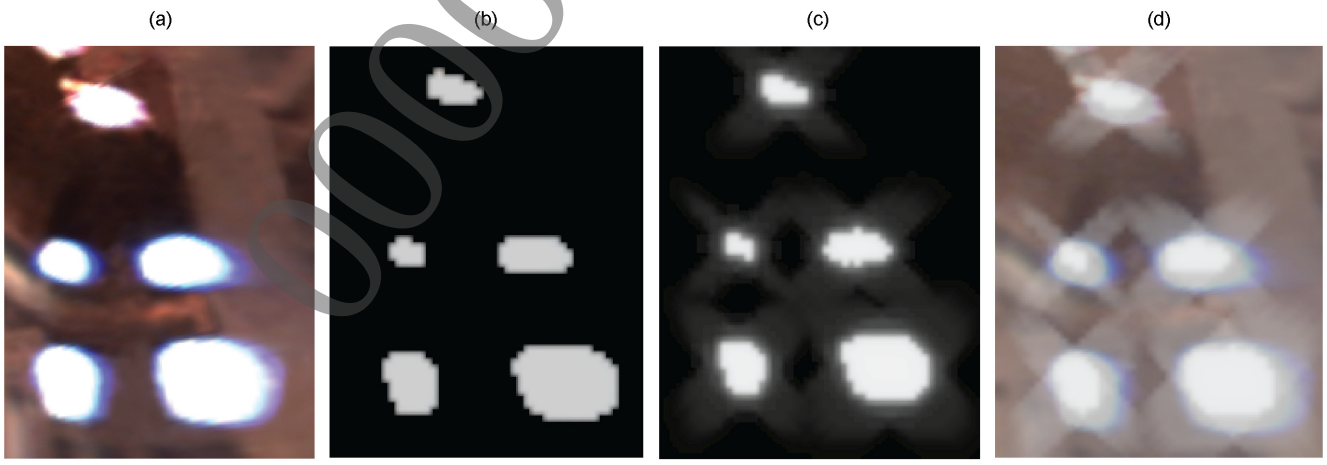
The star pattern created in this sample has four rays. Each ray requires three passes. Each pass uses different weights for the four samples along a ray direction. The weights are given by the following function:

weightStar = pow(n, distance)

This function attributes less weight to samples that are farther from the pixel being rendered. This pixel combination allows the creation of longer, gradually fading rays along the specific direction with each pass.

Each directional ray is rendered to a separate render target, and then all the targets are merged to create the star-like effect. A weighted color value of the star contribution is added to the fragment color when rendering the final composited scene (Figure 6).

Figure 6



(a) Original scene
(b) Filtered scene
(c) Star effect applied
(d) Final composited scene

Tone Mapping

The final step is to prepare the scene to be displayed on-screen while correctly reflecting HDR colors and maintaining the strong contrast ratio. In this step, an operator maps each pixel's luminance value to that of the screen. In this sample, a simple but efficient exponential function is applied that sets the exposure and then gamma-corrects the final scene that can be controlled by user-provided parameters. The function is given by:

$$outColor = 1.0 - exp(-inputColor * exposure)$$

$$finalColor = pow(outColor, 1.0/gamma)$$

A8R8G8B8 Format Render Target Usage

If using a fixed-point format such as A8R8G8B8 for the color surface, to correctly handle the HDR information of the scene you need to perform some encoding of the original scene's color information. One way to achieve this is by using the alpha channel to store a multiplier representing the overall brightness of the scene. Encoding is done by finding the largest color value among the RGB-channels and 1.0, and then dividing the RGB values by that value:

$$\alpha = \frac{1.0}{\max(1, r, g, b)}$$

$$rgb^{encoded} = rgb^{original} * \alpha$$

The cube map together with the lighted up model is rendered, and encoded values are stored in an integer texture to be used for all post-processing. When applying the effects, all the color components are decoded, and then the processing is performed on the decoded data. This decoding is done by dividing the incoming color value by this alpha:

$$rgb^{decoded} = \frac{rgb^{encoded}}{\alpha}$$

After the processing is done, the values are again encoded and stored to the integer render target to be used for later processing. Because multiple passes are done when performing bloom and star effects, decoding and encoding each time affects the performance; however, performing 8-bit integer computation is slightly faster than performing 16-bit FP computation. The decoding is also done before the final compositing and tone mapping to calculate the final pixel color.

3 Implementation

This chapter describes the implementation of HDR rendering and the effects shown in Figure 3. The sample is located in the following folder:

```
SDK/target/samples/sample_code/graphics/tutorial_hdr_rendering
```

The sample uses the `sample_utilities_cpp` library, which has some basic utility functions for initializing some low-level libraries and setting up resources as required by the sample.

Initializing Render Targets

The first step in implementing HDR rendering is to initialize the render targets and corresponding texture and color surface for storing in the intermediate offscreen processing results. To do this, use the following functions (`tutorial_hdr_rendering.cpp`):

```
// Initialize a render target as well as the associated texture
// and color surface used for rendering the scene.
int ret = m_renderTargets[idx].initialize(
          getGraphicsContext(),
          width,                              // width of the surface.
          height,                             // height of the surface.
          SCE_GXM_COLOR_SURFACE_LINEAR,       // memory layout type.
          colorFormat,                        // color format for the surface.
          outputRegisterSize);                // output register size.
SCE_DBG_ASSERT(ret == SCE_OK);
```

The `idx` is the index corresponding to the offscreen target that is set for all of the effects, such as downsampling, bloom, and star streaks.

Rendering HDR Effects

After the render targets are set, rendering to these targets is started. First, the scene with the environment map on which HDR processing is to be performed is rendered on its offscreen target. The setup is done as follows:

```
pGraphicsContext->beginScene(
&m_renderTargets[CUBEMAP_TARGET],// render target to render environment with
                                 // cube map
&m_depthStencilSurface);         // depth/stencil surface

// set state params
...

// set params for cube map shader, and render cube map for the environment
...

// set shader params for rendering the duck, and draw it
...

pGraphicsContext->endScene();
```

This is then followed by a sequence of HDR effects applied to the output from the previous scene. This is done by calling `renderHDREffects()`, which in turn calls the following functions in the indicated sequence:

```
drawRenderTargetByIndex(REDUCTION_TARGET); // downscales the render target to
                                           // quarter-resolution to perform
                                           // bloom and star effect operations

drawRenderTargetByIndex(FILTER_TARGET);    // perform operations to filter out
                                          // high luminance areas

drawBloom(BLOOM_TARGET1);  // apply a 3x3 Gaussian blur filter
drawBloom(BLOOM_TARGET4);  // apply an 11x11 Gaussian blur filter

drawStar(STAR_TARGET1);    // apply filter to render bottom left star ray
drawStar(STAR_TARGET2);    // apply filter to render bottom right star ray
drawStar(STAR_TARGET3);    // apply filter to render top left star ray
drawStar(STAR_TARGET0);    // apply filter to render top right star ray
```

Note: In this sample, these various intermediate effects are rendered in separate scenes, which is not well suited for PlayStation®Vita and will incur a penalty for each new scene. A more favorable implementation of a post-processing chain uses the macrotile synchronization feature and is available at:

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

The remainder of this chapter illustrates the shaders that are used to implement the HDR effects discussed above.

Encoding to A8R8G8B8 Format

Encode the original scene with the cube map and lit-up model onto an A8R8G8B8 format render target as follows (`env_map_f.cg`):

```
// Find the maximum component of the RGB values.
half s = max(max(fragmentColor.r, fragmentColor.g), fragmentColor.b);

// Clamp the maximum value between 1 and 255 to avoid dividing by 0 for scale.
s = clamp(s, 1.0, INFINITY);

// Calculate the scale value.
half scale = 1.0/s;

// Encode the RGB color values by scaling with the calculated scale value and
// store it in alpha channel for decoding later on.
return half4(fragmentColor.rgb * scale, scale);
```

This step is not necessary if using high-precision textures; instead the scene data can be directly stored to FP targets and used for processing without encoding/decoding.

High-Luminance Filter

Extract high-luminance areas and render them to color surface for further processing (`filter_f.cg`):

```
half4 color = tex2D(envTex, texCoord);
// Calculate luminance.
half luminance = color.r + color.g + color.b;

half4 fragmentColor;

// Compare with threshold and output higher luminance.
if (luminance > gThreshold)
```

```
{
        fragmentColor = half4(color.xyz, 1.0);
}
else
{
        fragmentColor = half4(half3(0.0), 1.0);
}
```

Bloom Effect

A sequence of blur filters is applied to the high-luminance areas. Texture coordinates are calculated in the vertex shader and used in the fragment program for texture fetch and selecting samples. Because a separable filter is used, we first use a horizontal filter and then apply a vertical blur to the resultant target. The vertex shader for both filters is set as follows (`blur_filter*_v.cg`):

```
uniform float2 gOffset;
// gOffset = float2(1.0f/gWidth, 0.0f) for horizontal blur
// gOffset = float2(0.0f, 1.0f/gHeight) for vertical blur

uniform float gWeight[5];  // Array stores the Gaussian blur filter coefficients.

oTexCoord0 = aTexCoord;

float fw;
fw = gWeight[2] / (gWeight[1] + gWeight[2]);
oTexCoord1.xy = aTexCoord + (1.0f + fw) * gOffset;
oTexCoord1.zw = aTexCoord - (1.0f + fw) * gOffset;
fw = gWeight[4] / (gWeight[3] + gWeight[4]);
oTexCoord2.xy = aTexCoord + (3.0f + fw) * gOffset;
oTexCoord2.zw = aTexCoord - (3.0f + fw) * gOffset;
```

In the fragment shader (`blur_filter*_f.cg`), these coordinates are used to fetch texture samples and then added together as follows:

```
uniform half gWeight[3];
half4 fragmentColor;
half4 fragmentColor = gWeight[0] * tex2D(gTexture, texCoord0);

fragmentColor += gWeight[1] * (tex2D(gTexture, texCoord1.xy) +
                               tex2D(gTexture, texCoord1.zw));
fragmentColor += gWeight[2] * (tex2D(gTexture, texCoord2.xy) +
                               tex2D(gTexture, texCoord2.zw));
```

Star Effect

To apply this effect, a specific blur filter is used that acts in the direction of the required star ray. Four samples are picked up in that direction in the vertex shader (`star_fx_v.cg`), as shown below. The ray length is progressively increased by performing multiple passes and using the pass count to determine offset to pick the sample.

```
// These parameters are set by the user for each of the star rays.
uniform float3 gOffset;

const float2 offset = pow(3.0, gOffset.z) * gOffset.xy;

oTexCoord0 = aTexCoord;
oTexCoord1 = aTexCoord + offset;
oTexCoord2 = aTexCoord + 2 * offset;
oTexCoord3 = aTexCoord + 3 * offset;
```

In the fragment shader, each selected sample's color value is weighted by a factor (computed based on the sample number and pass count). All the weighted color values are then added together to create the final ray in the specified direction. The shader code (star_fx_f.cg) for this is as follows:

```
uniform half4 gWeight;
fragmentColor =  gWeight.x * tex2D(gTexture, texCoord0);
fragmentColor += gWeight.y * tex2D(gTexture, texCoord1);
fragmentColor += gWeight.z * tex2D(gTexture, texCoord2);
fragmentColor += gWeight.w * tex2D(gTexture, texCoord3);
fragmentColor.a = 1.0;
```

Final Compositing

The final composition pass performs a texture fetch on all of the processed render targets and adds them up with user-controlled weights. The fragment value is then tone-mapped by changing the exposure of the fragment value and then applying gamma correction in the fragment shader (`finalscene_f.cg`).

```
// Decode the base texture to get HDR color values.
half4 original;
original = tex2D(gTexture, texCoord0);

original.rgb *= (1.0 / original.a);

// Final compositing.
half4 fragmentColor = original;

// adding 5x5 blur filter component
fragmentColor += gWeightBloom.x * tex2D(gBloomTexture1, texCoord0);

// adding 11x11 blur filter component
fragmentColor += gWeightBloom.z * tex2D(gBloomTexture4, texCoord0);

// adding star effect component
fragmentColor += gWeightStar.x * starColor;

// Tone mapping.
fragmentColor.xyz = half3(1.0) - exp(-fragmentColor.xyz *gWeightStar.z);
fragmentColor.xyz = pow(fragmentColor.xyz, gWeightStar.w);
```