Shadow Techniques Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

Purpose

This document provides a walkthrough of some shadow techniques implemented on the PlayStation®Vita hardware. It includes:

- An overview of the implementation of some common shadow map algorithms and the various artifacts affecting them
- Several filtering techniques that help alleviate those artifacts and improve the quality of shadowed scenes
- A discussion of Cascaded Shadow Maps as an efficient technique for perspective aliasing free shadows in large-scale scenes

Typographic Conventions

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

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
```

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.

Related Documentation and Other Resources

Sample Code

The code for both of the shadow-technique samples, `soft_shadows` and `cascaded_shadow_maps`, is located at:

```
SDK/target/samples/sample_code/graphics/tutorial_shadow_techniques
```

References

[1] L. Bavoil: “Advanced Soft Shadow Mapping Techniques”, http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_SoftShadowMapping.pdf, 2008.

[2] C. Everitt: “Projective Texture Mapping” (NVIDIA white paper), http://developer.nvidia.com, 2001.

[3] M. Bunnell, F. Pellacini: “Shadow Map Antialiasing” (pages 185-192), in *GPU Gems*, 2004.

[4] A. Lauritzen: “Summed-area variance shadow maps” (pages 157-182), in *GPU Gems 3*, 2007.

[5] R. Fernando: “Percentage-closer soft shadows”, in *ACM SIGGRAPH 200: Sketches and Applications*, 2005.

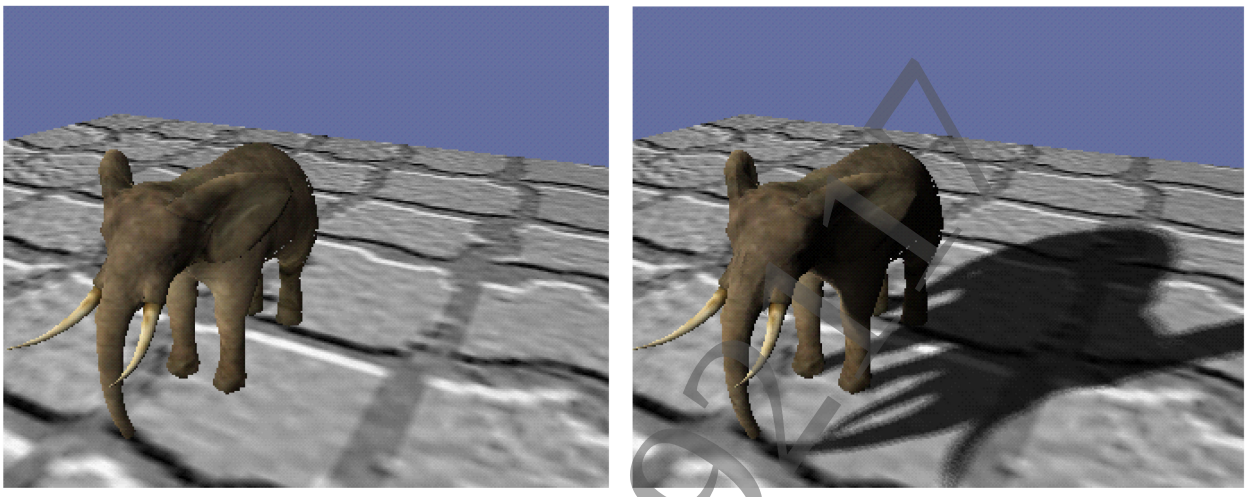
[6] R. Dimitrov: “Cascaded Shadow Maps”, http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf, 2007.

[7] M. Salvi: “Rendering filtered shadows with exponential shadow maps”, in *ShaderX 6.0 – Advanced Rendering Techniques*, 2008.

1 Introduction

Shadows are an essential component in the rendering of convincing 3D graphic scenes. Without them, scenes often feel unrealistic and flat, and the spatial location of objects in the scene can be ambiguous. Whether hard or soft, physically correct or perceptually satisfying, real-time shadows are essential for games development. This chapter provides an introduction to the shadow mapping techniques and filtering options available for improving the performance and quality of scenes.

Figure 1 Scene Without and With Shadows



(a) Without Shadow (b) With Shadow

Shadow Mapping Theory

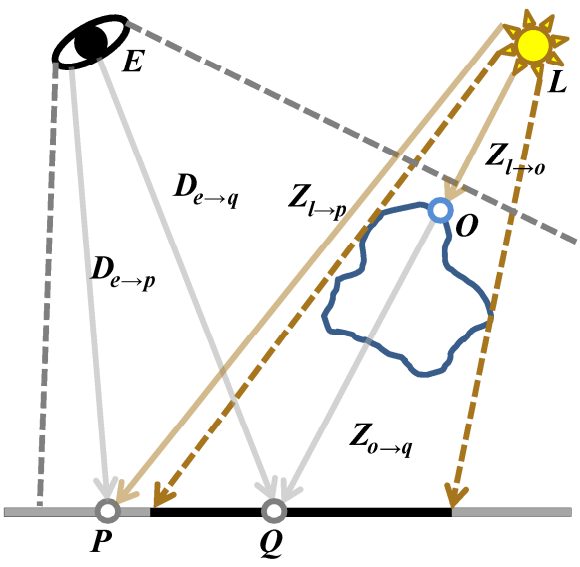
Shadow mapping is a very effective way to produce fast and realistic shadows. It is an image space technique that involves two passes:

(1) The first pass renders the scene depth from the light's point of view.

(2) The second pass renders the scene from the eye's point of view using the depth information from the first pass.

The algorithm is shown in Figure 2.

Figure 2 Shadow Map Algorithm



Render scene from Light Position L to generate a "depth map" to store depth values "Z_l" of closest pixels to the light.

Render scene from Eye Position E

- Project all depth values "D_e" of each fragment from eye position to get its depth value "D_l" from the light position.
- If $D_l = Z_l$, the fragment is not in shadow. If $D_l > Z_l$, the fragment is in shadow.

In the figure on the left:

- For the fragment corresponding to P, $D_{l\to p} = Z_{l\to p}$ so it is not in shadow.
- For the fragment corresponding to Q, $D_{l\to q} > Z_{l\to o}$ so it is in shadow.

The shadow mapping techniques have many advantages; for example, they are easy to set up and implement, and can effectively work for any geometry that can be rasterized. However, because discrete samples are considered for shadow computation, the rendered scene suffers with severe aliasing artifacts that need to be handled.

Common Shadow Mapping Techniques

A comprehensive list of shadow mapping techniques is provided in L. Bavoil, "Advanced Soft Shadow Mapping Techniques"[1] (see "Related Documentation and Other Resources").

This section describes the shadow maps that have been implemented in this tutorial. They are applied for rendering Soft Shadows using some standard filtering techniques, and suit real-time applications.

Soft Shadows

When using point light sources, the shadows generated by shadow mapping techniques result in hard shadows. However, in a real-world scenario, light sources have a finite area which results in more realistic soft shadows with a smooth transition from "no shadow" to "full shadow" in the penumbra regions. Also, rendering soft shadows provides higher quality shadows because it decreases the aliasing artifacts noticeable in traditional shadow maps.

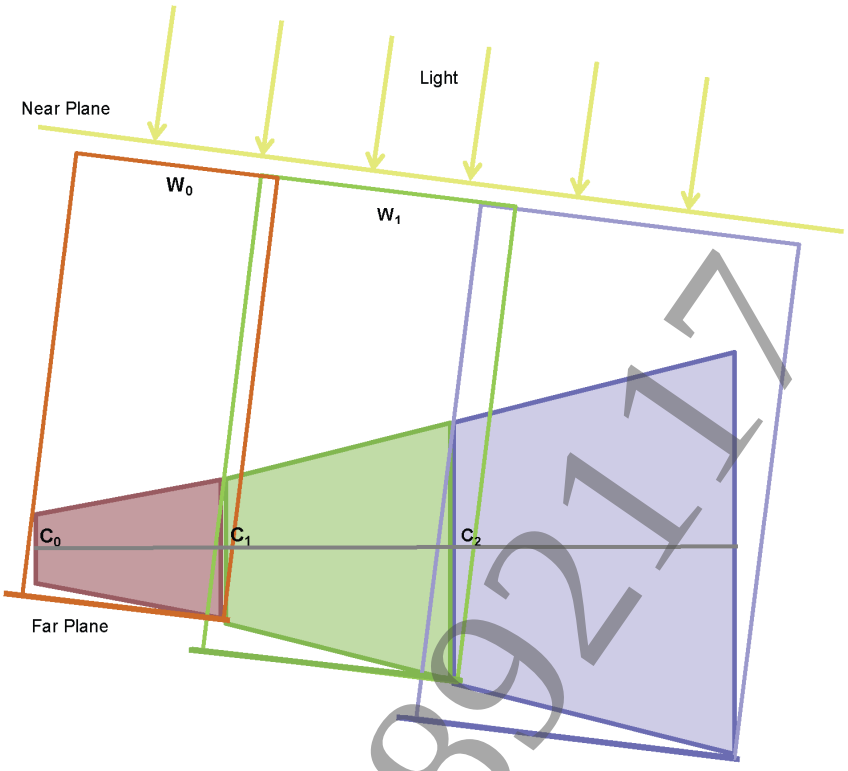
Percentage-Closer Soft Shadows (PCSS) is one of the techniques that efficiently generate perceptually correct soft shadows in real-time. Like traditional shadow maps, PCSS uses a single shadow map, is independent of scene complexity, and does not require any specific processing, thus easily replacing the standard technique but with more convincing results. The algorithm uses Percentage Closer Filtering (PCF) with an adjustable width that varies according to the shadow caster distance. Another filtering technique that can be used is Variance Shadow Maps (VSM); this technique facilitates filtering because it uses color textures instead of depth maps.

Cascaded Shadow Maps

Shadow maps are a very popular technique to obtain realistic shadows in game engines. However, when trying to use them for large spaces, shadow maps get harder to tune and will be more prone to exhibit shadow acne and aliasing. Cascaded Shadow Maps (CSM) is a technique that alleviates the aliasing problem by providing higher resolution of the depth texture near the viewer and lower resolution far away. This is done by splitting the camera's view frustum into sub-frustums, setting up the corresponding light's frustum to enclose each of the camera's sub-frustums, and generating a separate shadow map for

each split in an attempt to make the screen error constant. Figure 3 shows an example. For more details on this technique, refer to R. Dimitrov, "*Cascaded Shadow Maps*"[6] (see "Related Documentation and Other Resources").

Figure 3 Camera's View Frustum with Splits and Corresponding Light's View Frustums

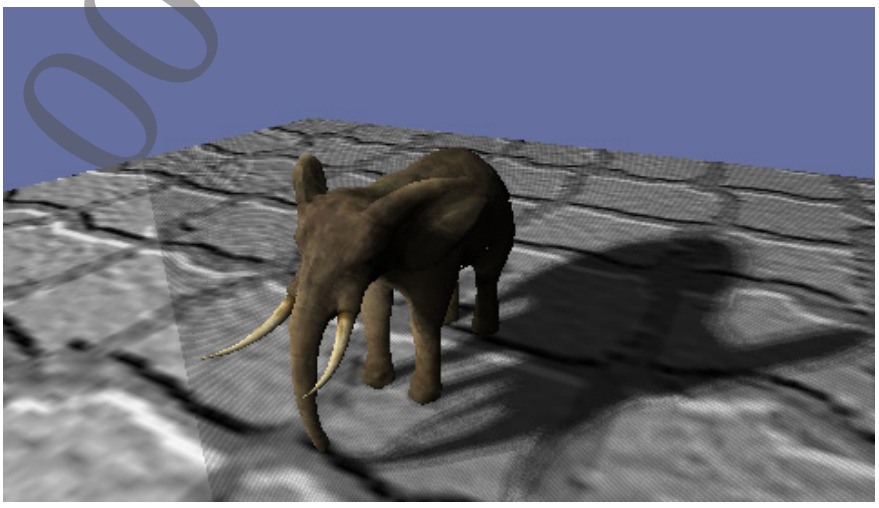


Common Filtering Techniques

Depth Bias

One of the common artifacts visible when rendering shadows is shadow acne. Shadow acne occurs due to the limited precision of the depth map, which leads to different Z and D sampled values and therefore returns an incorrect shadow test. Adding a small bias to the depth value can make the shadow test less inclusive. However, adding too much bias can result in the depth test incorrectly passing, causing a disconnect between the shadow and shadow caster.

Figure 4 Artifacts Due to Incorrect Depth Test Results



A constant bias is added for every polygon irrespective of how much its slope is relative to the light. This often leads to an incorrect bias being added for some polygons and therefore the artifacts being still visible. This limitation can be efficiently resolved using the slope-scale depth bias method, where a large bias is added to a polygon that is steep-sloped relative to the light direction, and a small bias is applied to polygons that are facing more towards the light.

Percentage Closer Filtering (PCF)

The PlayStation®Vita hardware supports PCF and works by comparing the projected pixel distance D_l with the depth value from the shadow map Z_l and its closest neighbor values, giving shadow intensity as a percentage of the shadow tests. Increasing the PCF kernel width increases the softness of the shadow. Details about this filtering technique are discussed in M. Bunnell and F. Pellacini, "*Shadow Map Antialiasing*"[3] (see "Related Documentation and Other Resources").

An extension to this technique involves varying the amount of softening based on penumbra size, which is estimated using the blocker and receiver depth and light size as parameters.

Variance Shadow Map (VSM)

Variance Shadow Maps (VSMs) provide another solution to overcome the problem of aliasing, but instead of filtering in screen-space (as used in PCF) VSM facilitates the use of pre-filtered shadow maps. In addition to the 1-channel depth stored for normal shadow maps, a VSM has a second channel to store the square of this depth.

The advantage of VSM is that the 2-channel shadow map can be filtered using the standard filtering capabilities of the GPU to get the first two moments' mean $f(z)$ and variance $f(z^2)$ over a region. These two moments represent a probability distribution of depths at each shadow map texel.The visibility function over an arbitrary filter region can then be estimated using Chebyshev's Inequality, which yields an upper bound on the amount of light reaching the fragment being shaded. Additionally, VSM offers the flexibility of using a separable blur to apply a sufficient-sized Gaussian filter directly on the map, which is quite efficient and gives high quality soft edged shadows.

However, note that the VSM technique suffers from the problem of light bleeding where areas that should be fully in shadow are partially lit. This artifact is quite significant in regions of high depth variance values that mainly occur when two distant but overlapping occluders within a filtering region are present in the scene. Such scenes will require additional steps to eliminate or alleviate the issue.

Exponential Shadow Map (ESM)

Exponential Shadow Maps (ESMs) provide another way to pre-filter shadow maps to implement soft shadows. The shadow test for this is reformulated to use an exponential function to relate the receiver and occluder depths instead of directly comparing their depths as in standard shadow maps.

Similar to the technique for VSM, ESM supports increased temporal coherence and exploits pre-filtering by using separable blur and hardware texture-filtering modes to produce soft shadows. However, the ESM technique is more efficient because it uses single-term approximation, consumes less memory, and does not suffer from light-leaking artifacts as in VSM. We use this technique for filtering cascaded shadow maps.

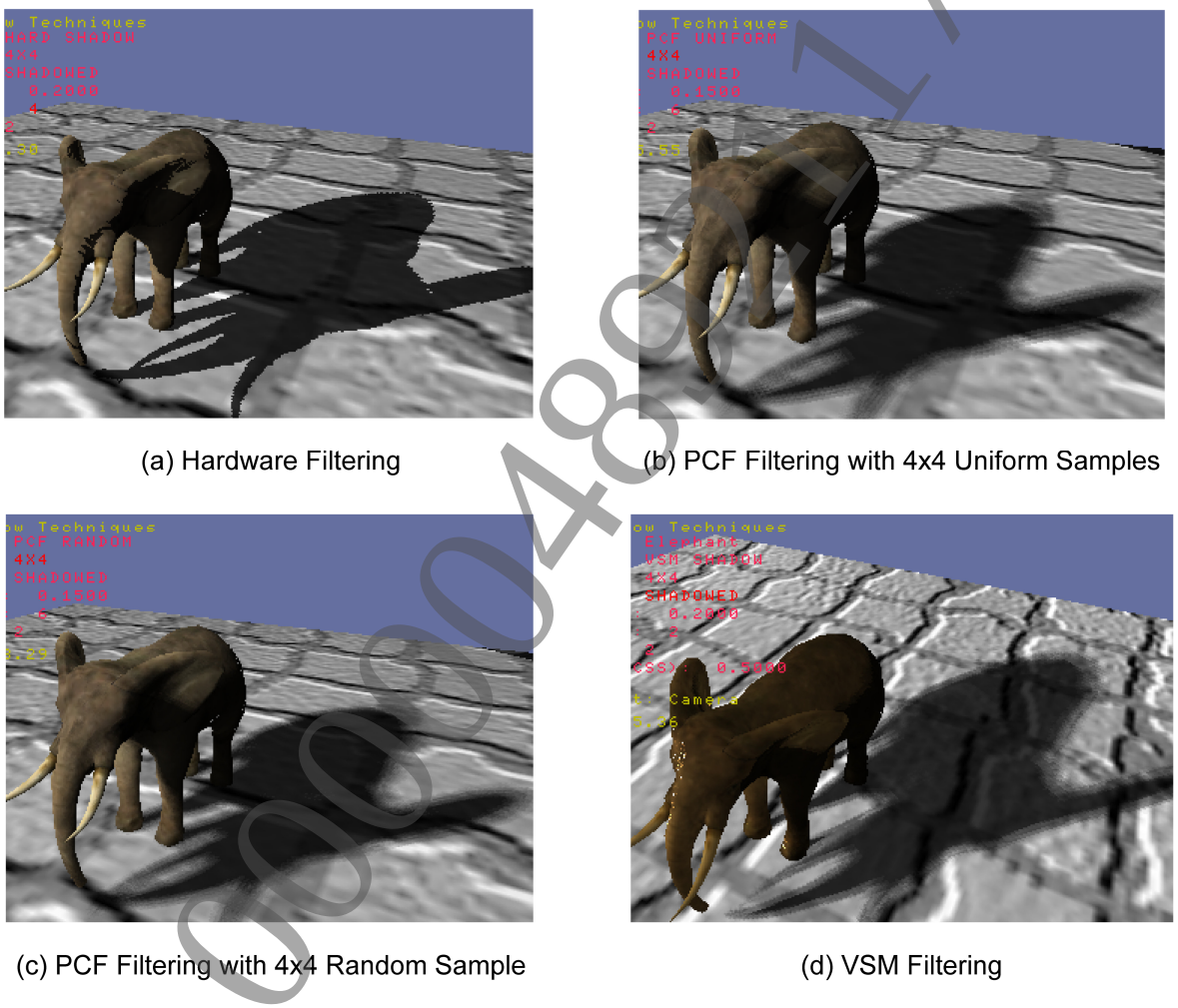
2 Tutorial Walkthrough

This chapter describes a set of algorithms that are implemented in this tutorial using the PlayStation®Vita capabilities and the libgxm graphics API features.

Overview

The tutorial implements soft shadows and shadows using the CSM technique, and explains how to improve the quality of shadows and achieve high performance using these shadow map techniques. The tutorial allows you to change parameters and see their effects so that you can choose the best setup for your application depending on your quality and performance requirements.

Figure 5 Different Filtering Techniques



(a) Hardware Filtering

(b) PCF Filtering with 4x4 Uniform Samples

(c) PCF Filtering with 4x4 Random Sample

(d) VSM Filtering

Rendered Scene

The scene used to demonstrate the shadow map techniques is a small-scale scene with a plane and a model rendered with soft shadows using a single shadow map.

Soft Shadows

Soft shadows are implemented using both a point source (PCF and VSM) and an area-light source (PCSS) to calculate the lighting of the penumbra regions. Projective texture is also used to render the scene with hard-shadows for comparison purposes.

The first step is to render the scene from the light's point of view to generate a "depth map". The PlayStation®Vita hardware supports an on-chip depth buffer and a stencil buffer of F32 and U8 precision respectively, which is used to store the map. This depth map is then used for the shadow test using the different techniques listed below.

Projective Textures

One of the techniques to render shadow is projective texture mapping. For more information about this technique, refer to C. Everitt, "*Projective Texture Mapping*"[2] (see "Related Documentation and Other Resources").

(1) The depth map from the light's viewpoint is encoded into a texture, and then projected onto the models when they are being rendered from the camera's viewpoint.

(2) A texture matrix T is generated; this is effectively the model-view-projection matrix from the light's viewpoint, and transforms a vertex position in world space to light space.

(3) Additionally, a scale and bias matrix is applied to generate the texture coordinates in the range [0, 1]. The texture matrix used is computed as:

$$T = SP_lV_lM$$

where M is the model matrix, V_l is the view matrix for the light, P_l is the projection matrix the light frustum, and S is the scale-bias matrix represented as:

$$S = \begin{bmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The texture coordinates are calculated as:

$$\begin{bmatrix} s & t & r & q \end{bmatrix}^T = T \begin{bmatrix} x_o & y_o & z_o & w_o \end{bmatrix}^T$$

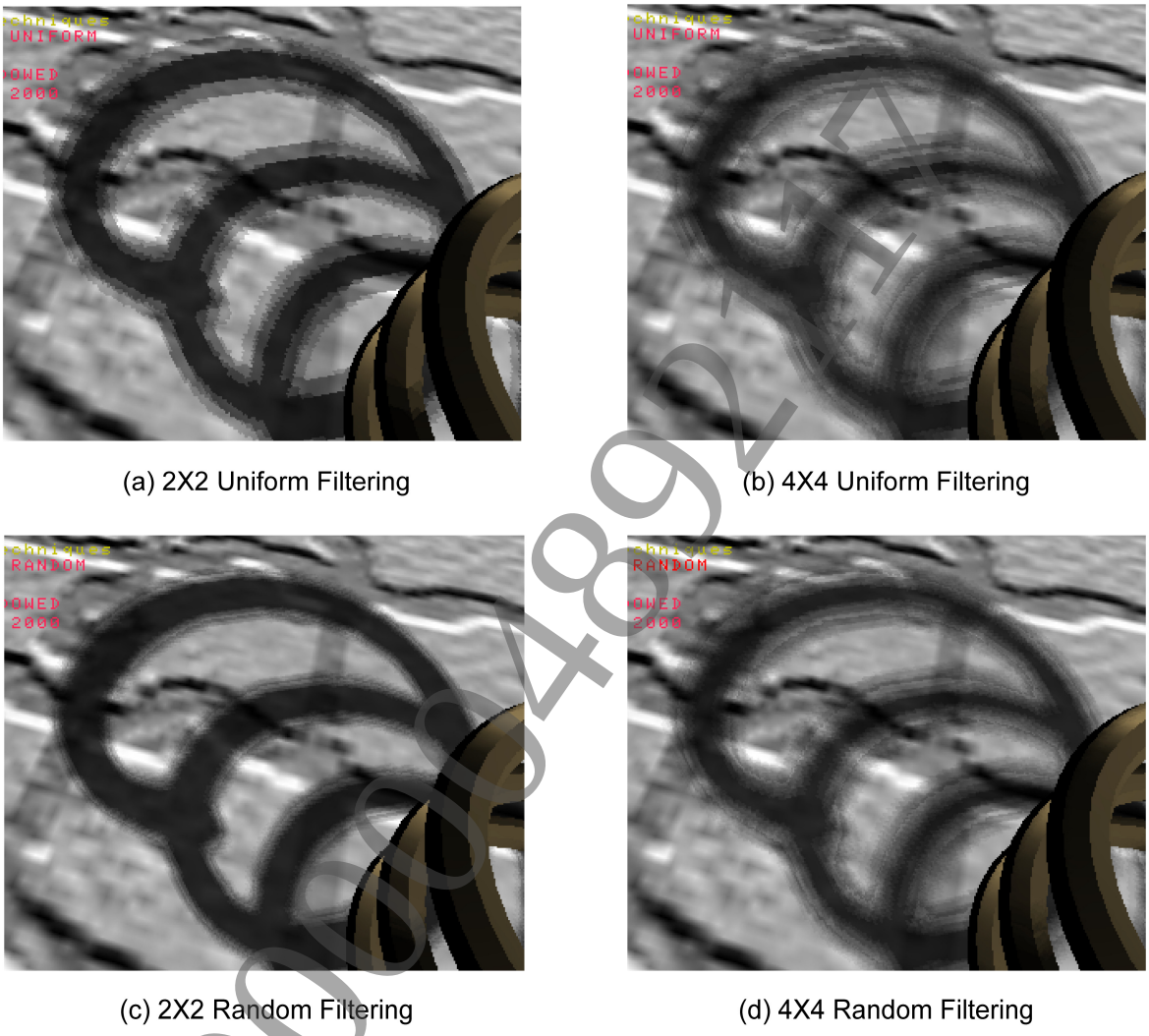
such that $\left(s/q, t/q \right)$ represents the lookup coordinates in the depth texture to sample depth value Z_l and compares it with r/q. The comparison returns a binary value 1 or 0 depending on whether $Z_l \geq r/q$ (no shadow) or $Z_l < r/q$ (shadowed).

PCF

The binary results of the shadow test above produce hard shadows, which also suffer from aliasing artifacts.

(1) Soft shadows are added to alleviate the aliasing problem by fading off towards the edge, and also to produce a more realistic looking scene. The PCF technique is used, where shadow testing is performed with a larger set of samples around the local neighborhood. PlayStation®Vita hardware provides support for bilinear filtering that uses a 2X2 (4 samples) kernel size for shadow tests.

Figure 6 PCF Shadow Results for a Helix



(a) 2X2 Uniform Filtering

(b) 4X4 Uniform Filtering

(c) 2X2 Random Filtering

(d) 4X4 Random Filtering

(2) Additionally, a 4X4 grid (16 uniform samples) is used to compare the depths, and then the averaged result of all the shadow tests is used to calculate the final shadow intensity. This blurs the shadow and produces softer shadow edges.

(3) The visual quality is further enhanced by picking up the 16 random samples for depth test.

VSM

For variance shadow maps we perform the following steps:

(1) We render into a two-channel buffer, rendering both the depth and the square of the depth.

(2) Next, pre-processing is done on the shadow map textures to facilitate filtering, and also apply blur to the VSM for further reducing aliasing and producing soft shadows. A two-pass separable Gaussian blur filter is applied for this. However, note that on performing this step will require two separate scenes (one for each pass) and as such involve the overhead cost associated with beginning a new scene.

(3) The textures are then pre-filtered to calculate the mean μ and variance σ^2 of a Gaussian distribution as:

$$\mu = E(Z)$$
$$\sigma^2 = E(Z^2) - E(Z)^2$$

where $E(Z)$ and $E(Z^2)$ represent the first and the second moments respectively of the depth distribution.

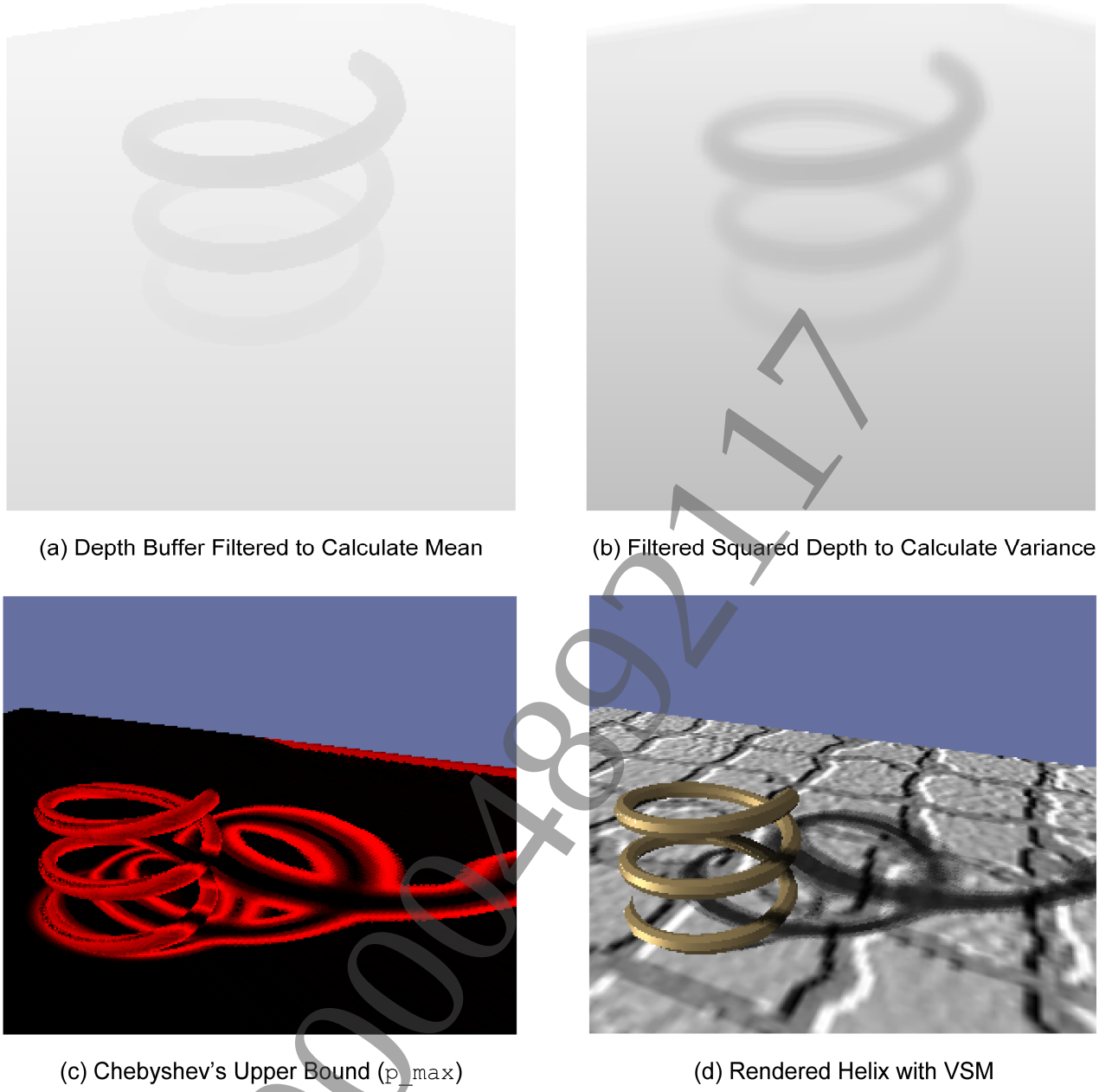
(4) Now, to perform shadow test on a fragment, we apply Chebyshev's inequality, which gives an upper bound on the probability of the fragment at depth D being in shadow. This is given as:

$$P(Z \geq D) \leq p_{\max}(D) = \frac{\sigma^2}{\sigma^2 + (D - \mu)^2}$$

The inequality is valid for $D > \mu$, in which case P is the fraction of pixels in the filter region for which the shadow test fails. If $D \leq \mu$, the fragment is considered to be fully lit.

For additional details about this algorithm, refer to A. Lauritzen, "Summed-area variance shadow maps"[4] (see "Related Documentation and Other Resources").

Figure 7 Soft Shadows with Variance Shadow Maps



(a) Depth Buffer Filtered to Calculate Mean

(b) Filtered Squared Depth to Calculate Variance

(c) Chebyshev's Upper Bound (`p_max`)

(d) Rendered Helix with VSM

PCSS

In terms of technique, PCSS is similar to normal PCF and also provides the same advantages. The main difference between PCSS and PCF is the use of a variable-width filter in PCSS that results in the correct degree of softness.

The PCSS implemented in this tutorial is based on R. Fernando, "Percentage-closer soft shadows"[5] (see "Related Documentation and Other Resources").

It involves three steps:

(1) The blocker search

(2) Penumbra size estimation

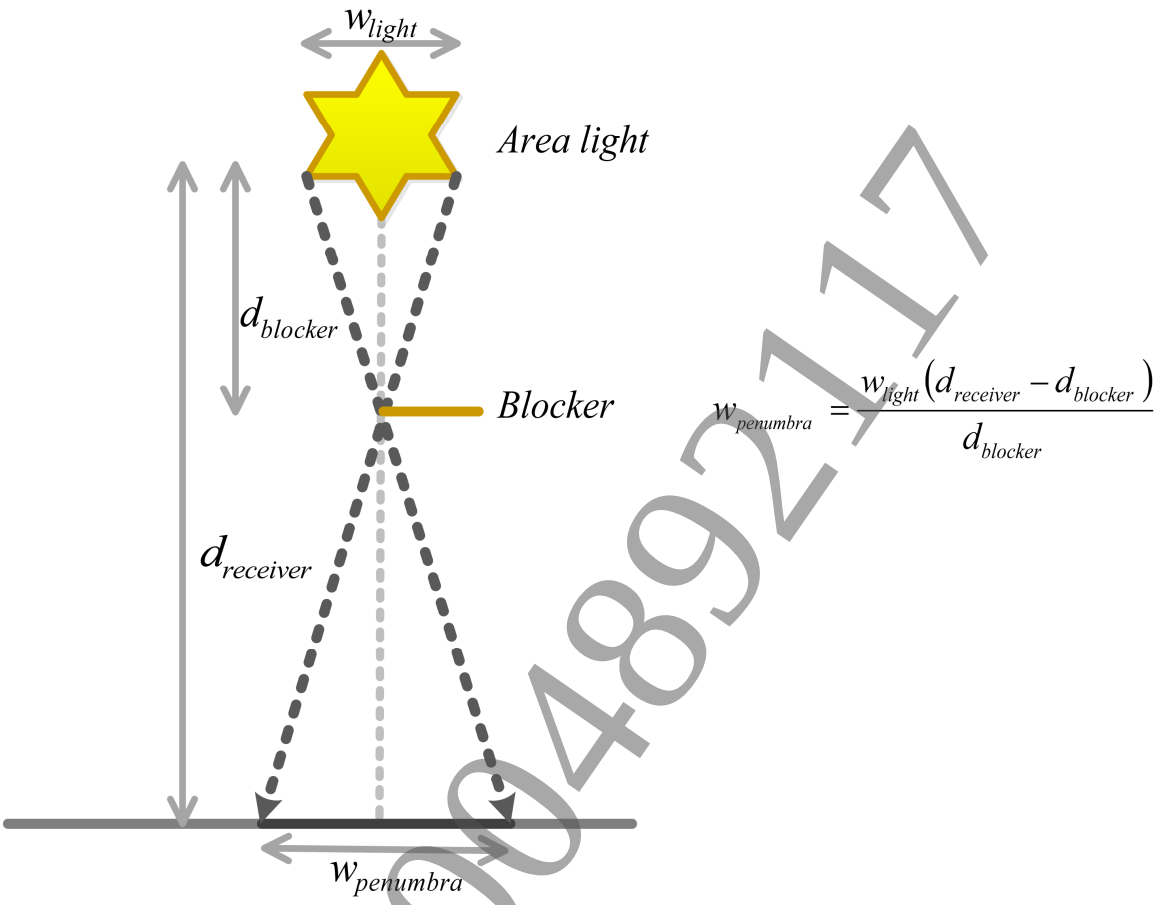
(3) Shadow filtering

The Blocker Search

(1) The blocker search samples a search region around the current shadow texel and finds the average depth of any blockers in that region. The search region size is proportional to both the light size and the distance to the light.

(2) The standard depth test is used to determine if the blocker is closer to the light than the current fragment.

Figure 8 Blocker Search



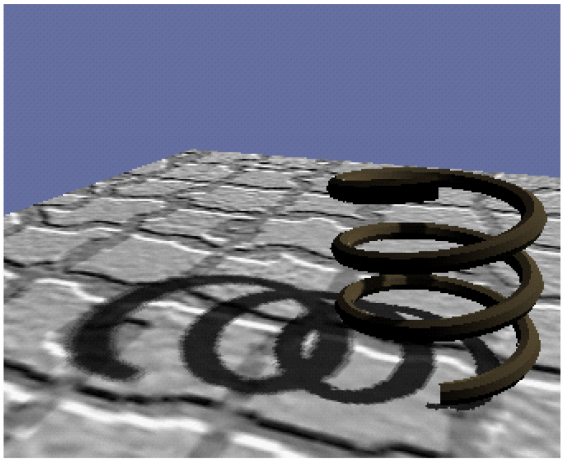
where, w_{light} refers to the light size.

Penumbra Size Estimation

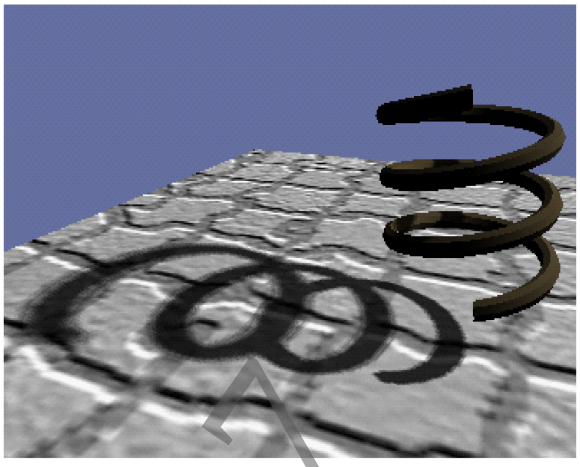
This step of the algorithm uses the previously computed blocker depth $d_{blocker}$ to estimate the size of the penumbra, based on a parallel planes approximation.

In Figure 9(a) and (b), as the blocker helix moves away from the receiver floor and the penumbra radius increases, the shadows become softer. Figure 9(c) shows the result of shadow with PCF random filtering. Figure 9(d) shows PCSS for the skeleton. Note that with PCSS, the skeleton parts closer to floor have harder shadows than parts farther from the floor.

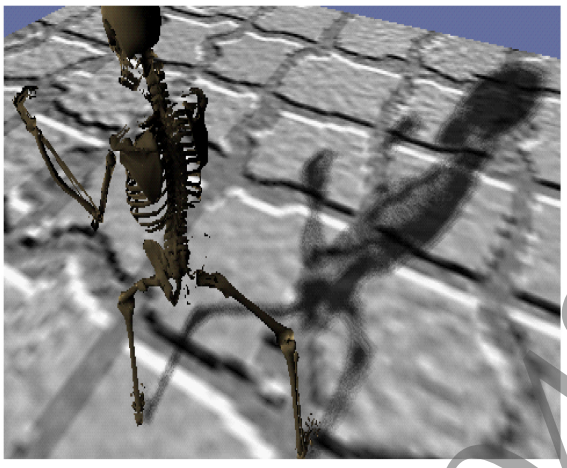
Figure 9 PCSS Results



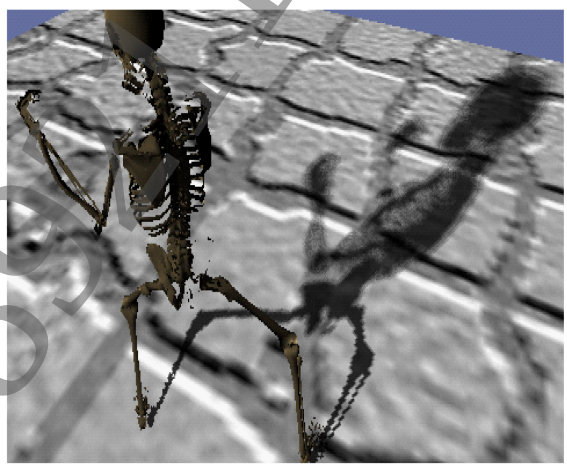
(a) Closer to the Receiver with Harder Shadows



(b) Softer Shadows as Helix Moves Away



(c) PCF Random Filtering Applied to Full Shadow



(d) Variable Sized Filter Used to Get PCSS

Shadow Filtering

This step is same as when performing PCF on the shadow map, but with a kernel size proportional to the penumbra estimate from the previous step.

Though PCSS generates perceptually correct soft shadow, the implementation involves an extra blocker search step that uses some additional samples besides the standard shadow filtering. This makes it relatively expensive technique with a higher performance cost, so should be used after careful evaluation.

Cascaded Shadow Maps

This technique uses multiple shadow maps for generating shadows in a landscape. The shadow maps corresponding to each cascade can be either stored on multiple individual depth/stencil surfaces or on a large single depth/stencil surface as an atlas by choosing the correct viewport. When using individual surfaces, the depth for each of the cascade is rendered in separate scenes. However, on PlayStation®Vita setting up separate scene adds up an overhead firmware cost.

Using a single surface gives a better performance as it reduces the scene-setup overhead corresponding to each cascade's shadow map. Additionally, you will also have the advantage of fetching sample for depth comparison in the main render scene from a single shadow map rather than 3 separate shadow map textures and get better efficiency. The algorithm proceeds as follows:

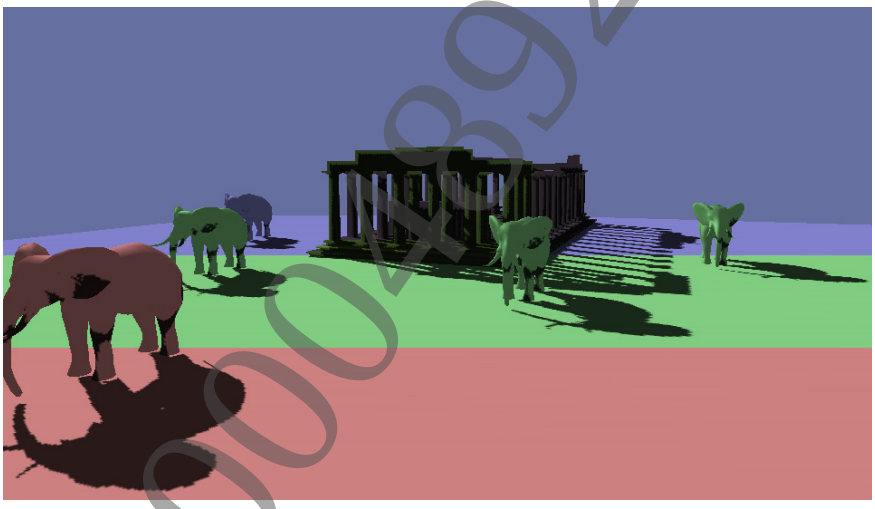
- Partition the camera's view frustum into subfrusta, and calculate the far and near planes for each.
- For each subfrusta:
  - Compute the light's view frustum enclosing the corresponding subfrusta and find minimum and maximum extends for the frustum
    Additionally, to avoid empty spaces within the light's frustum and increasing shadow map resolution, focus the light's frustum to include just the potential shadow casters for that split.
  - Compute a crop projection matrix using the extents calculated in the previous step. Also compute the view projection and texture scale bias matrix for each sub-frustum.
  - For every light's frustum/cascade, set up the viewport on the single shadow map and render the scene depth from the light's point of view in separate regions of the depth buffer.
- Render the scene from the camera's point of view. Depending on the fragment's depth value in the view space, pick an appropriate shadow map for depth sample look up and calculate shadow term.

The view frustum is automatically partitioned in the camera space where the split positions for the specified near and far plane range $[D_{\min}, D_{\max}]$ are calculated using an exponential distribution given by:

$$D_i = \lambda n\left(D_{\max}/D_{\min}\right)^{i/N} + (1-\lambda)\left(D_{\min} + \left(i/N\right)\left(D_{\max} - D_{\min}\right)\right)$$

where λ is the factor to control split correction term to counter sudden resolution changes at the split points, and N is the total number of splits. Figure 10 shows the scene with three partitions.

Figure 10 Camera's View Frustum Split into Three Subfrusta

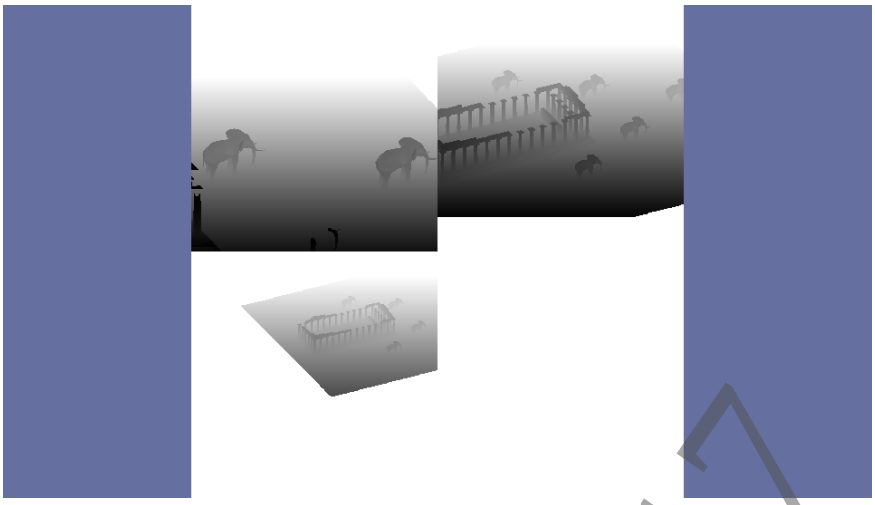


Next we calculate the light's frustum bounds by projecting the selected split frustums into the light's clip space and then use them to build a crop matrix for the light as:

$$C_i = \begin{pmatrix} S_x & 0 & 0 & O_x \\ 0 & S_y & 0 & O_y \\ 0 & 0 & S_z & O_z \\ 0 & 0 & 0 & 1.0 \end{pmatrix}, \text{ where } \begin{aligned} S_x &= 2.0/(X_{\max} - X_{\min}) \\ S_y &= 2.0/(Y_{\max} - Y_{\min}) \\ S_z &= 1.0/(Z_{\max} - Z_{\min}) \end{aligned} \text{ and } \begin{aligned} O_x &= -0.5*(X_{\min} + X_{\max})*S_x \\ O_y &= -0.5*(Y_{\min} + Y_{\max})*S_y \\ O_z &= -Z_{\min}*S_x \end{aligned}$$

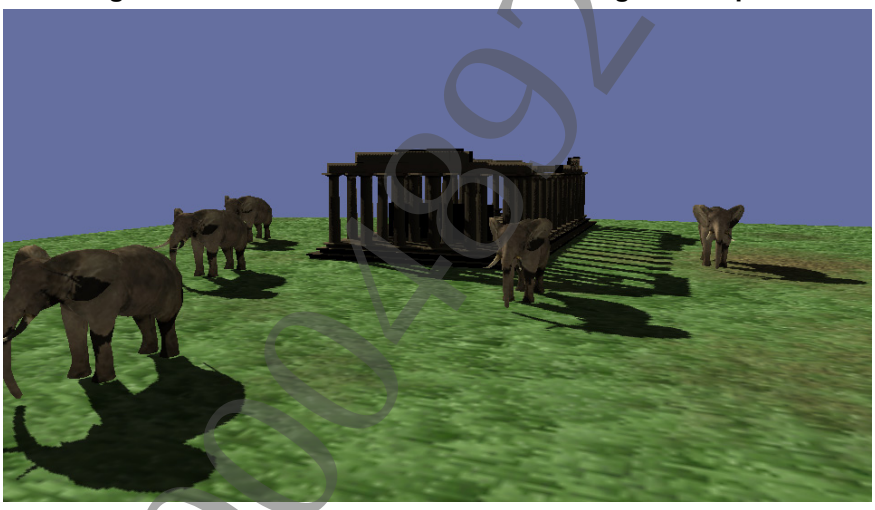
with $[X_{\min}, X_{\max}]$, $[Y_{\min}, Y_{\max}]$ and $[Z_{\min}, Z_{\max}]$being the light's bounds along x, y and z axis. The light's view-projection matrix for this split $M = C_i P_l V_l$ is then used to render the scene to the sub-region of the depth surface (Figure 11) assigned for this split, just as in the standard shadow map technique.

Figure 11 Shadow Maps Generated for Each Split on a Single Depth/Stencil Surface



For the final scene synthesis, for each fragment rendered from camera space its distance is used to find the split it belongs to and then perform the standard shadow test using the depth from the shadow map corresponding to that split.

Figure 12 Final Scene with Shadow Using Three Splits



ESM

As mentioned previously, we use this technique for filtering cascaded shadow maps.

For exponential shadow maps, based on the assumption $D \geq Z$, the shadow test $f(D,Z)$ is:

$$f(D,Z) = \lim_{\alpha \to \infty} e^{-\alpha(D-Z)}$$

which is approximated by using a large positive constant c to give the separable test function depending only on D and Z as:

$$f(D,Z) = e^{-c(D-Z)} = e^{-cD}e^{cZ}$$

To implement this algorithm, we perform the following steps:

(1) Render the linear occluder depth Z from the light position into a single-channel F32 buffer. The value stored is the distance of light to vertex divided by the far plane for light's frustum.

(2) Next, we apply two-pass separable blur on either direction by pre-filtering depth and use it to produce soft shadows. This can be done by either using the standard filter or using the logarithmic filter with filtered depth Z_f calculated as:

$$Z_f = Z_0 + \log\left(w_0 \sum_{k=1}^{n} w_k Z_k \right)$$

However, like VSM, performing this step requires two separate scenes for each of the separable pass and as such will involve an overhead firmware cost for each scene.

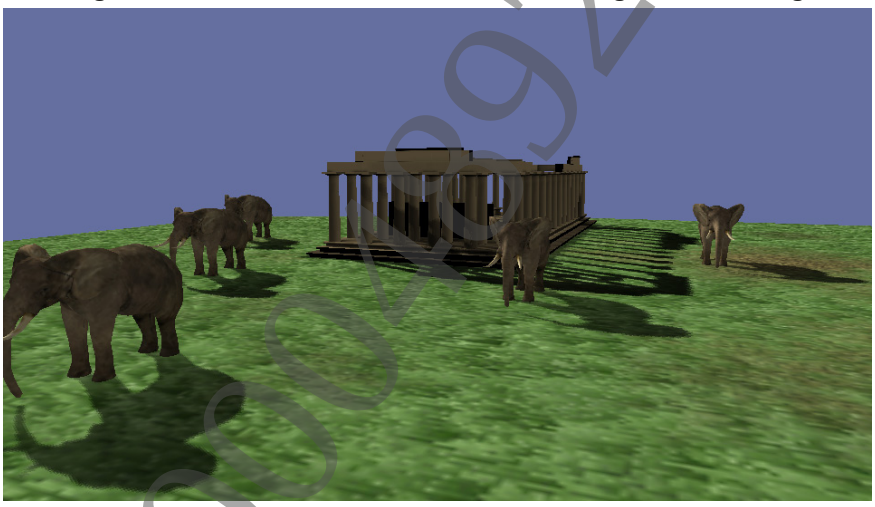
(3) When performing the shadow test for a fragment, the pre-filtered depth values for the occluder are used to sample filtered occluder depth Z_f and compute $f(Z_f)$, and the depth value of the current fragment is used to calculate $f(D)$ as:

$$f(Z_f) = e^{-cZ_f} \quad \text{and} \quad f(D) = e^{-cD}$$

(4) We can then write the shadow test as $f(D, Z_f)$ defined above, and clamp the results of the test to the `[0, 1]` range to ensure that the results are not brightened too much:

$$fShadowTerm = saturate\left(f(D) * f(Z_f)\right)$$

Figure 13 Final Scene with Shadows Using ESM Filtering



For additional details about this algorithm, refer to Marco Salvi's, "Rendering filtered shadows with exponential shadow maps"[7] (see "Related Documentation and Other Resources").

3 Implementation

This chapter describes implementation details for the shadow techniques discussed in the previous chapter, demonstrating the capabilities of the PlayStation®Vita hardware and usage of the libgxm graphics API features.

The code for both of the shadow-technique samples, `soft_shadows` and `cascaded_shadow_maps`, is located at:

```
SDK/target/samples/sample_code/graphics/tutorial_shadow_techniques
```

Shadow Map Setup

The first step is to initialize the depth/stencil surface. The memory is allocated to store the data, and the depth surface is set. The dimensions of the shadow map used for this implementation are 512x512.

```
m_pShadowMapBufferData = graphicsUtilAlloc(
	&m_graphicsData			// graphics utility context data
	GRAPHICS_UTIL_HEAP_TYPE_LPDDR_RW	// allocation on LPDDR for RW
	alignedWidth*alignedHeight*4,
	MAX(SCE_GXM_TEXTURE_ALIGNMENT,
	SCE_GXM_DEPTHSTENCIL_SURFACE_ALIGNMENT)
	);

// Initialize a depth/stencil surface for rendering light view depth.
err = sceGxmDepthStencilSurfaceInit(
	&m_shadowMapDepthSurface,
	SHADOW_MAP_DEPTH_FORMAT,
	SCE_GXM_DEPTH_STENCIL_SURFACE_TILED,
	alignedWidth,
	m_pShadowMapBufferData,
	NULL);
```

By default, the depth/stencil surface is set up not to store depth values. The following call enables the depth stencil to be written:

```
sceGxmDepthStencilSurfaceSetForceStoreMode(
	&m_shadowMapDepthSurface,
	SCE_GXM_DEPTH_STENCIL_FORCE_STORE_ENABLED);
```

To save the depth information onto a texture for depth fetch, we set up the texture and create a render target onto which the scene is rendered.

```
// Set up the texture.
err = sceGxmTextureInitTiled(
	&m_shadowMapTexture,
	m_pBlurMapBufferData,
	SHADOW_MAP_TEXTURE_FORMAT,
	SHADOW_MAP_DIMENSION,
	SHADOW_MAP_DIMENSION,
	1);

// Create a render target.
m_pShadowMapRenderTarget = graphicsUtilCreateRenderTarget(
	&m_graphicsData,
	SHADOW_MAP_DIMENSION,
	SHADOW_MAP_DIMENSION,
	SCE_GXM_MULTISAMPLE_NONE);
```

Generating Depth Map

A scene with offscreen render target initialized above for the shadow map is set. Only the back faces of the objects are rendered from the light's view point.

```
// Cull front faces.
sceGxmSetCullMode(pContext, SCE_GXM_CULL_CW);

// Set light view projection matrix.
m_shadowMapViewProjMatrix = m_lightCamera.buildProjectionMatrix() *
                            m_lightCamera.getViewMatrix();
```

Projective Shadow Mapping

To perform projective texturing, the texture matrix is calculated:

```
m_shadowMapTexViewProjMatrix = Matrix4(Matrix3::scale(Vector3(0.5f, -0.5f,
                               0.5f)), Vector3(0.5f, 0.5f, 0.5f))*
                               m_shadowMapViewProjMatrix;
```

The vertex shader (`scene_render_v.cg` is standard for all filtering techniques) and the fragment shader (`scene_render_f.cg`) for projecting shadows are as follows:

```
void main (
       float3 aPosition,
       float2 aTexCoord,
       float3 aNormal,
       uniform float4x4 world,
       uniform float4x4 viewProj,
       uniform float4x4 shadowTexViewProj,
       out float4 vPosition : POSITION,
       out float2  vTexCoord : TEXCOORD0,
       out float4  vShadowCoord : TEXCOORD1)
{

        float4 worldPos = mul(float4(aPosition, 1.f), world);

       // Calculate vertex position.
       vPosition = mul(worldPos, viewProj);

       // Calculate shadow map texture coordinates.
       vShadowCoord =  mul(worldPos,  shadowTexViewProj);

       // Calculate diffuse and specular light terms, and set color

       // Set texture coords
       vTexCoord = aTexCoord;
}

half Projective_shadow(uniform sampler2D texShadowMap,
                       float4 vShadowCoord)
{
       // Fetch color from the projective texture.
       half shadowTerm = f1tex2Dproj(texShadowMap, vShadowCoord);
       return shadowTerm;
}
```

Removing Artifacts

The libgxm graphics API provides the following function that uses a slope-based bias to remove the shadow acne. This removes the usual banding artifact that occurs because of incorrect shadow test due to limited depth buffer precision.

```
// m_shadowOffset - slope value, m_shadowBias - bias depth value.
sceGxmSetFrontDepthBias(pContext, m_shadowOffset, m_shadowBias);
```

PCF

A 2X2 kernel is used around the current fragment for filtering the depth map. Each sample undergoes a shadow test where its value is compared with fragment's depth and a binary result is given out which is combined and averaged to give the percentage value used for attenuating the light intensity for the fragment. The fragment shader (`pcf_shadow_f.cg`) function for this is as follows:

```
// This function calculates the shadow contribution for a fragment based on
// PCF filtering of the depth map.
half PCF_Filter(uniform sampler2D texShadowMap, // depth map
                        float4 coords,  //projective texture coordinates
                        half2 fParams) // fParams.x - grid size
                                       // fParams.y - texel size
{

        coords = coords / coords.w;
        float3 offset = fParams.x* fParams.y * coords.w* float3(1.0, 1.0, -1.0);

        float c =(coords.z <= tex2D<float>(texShadowMap, coords.xy -
                                           offset.xy).r) ? 1 : 0; // top left
        c += (coords.z <= tex2D<float>(texShadowMap, coords.xy + offset.xy).r)
                                           ? 1 : 0; // bottom right
        c += (coords.z <= tex2D<float>(texShadowMap, coords.xy + offset.zy).r)
                                           ? 1 : 0; // bottom left
        c += (coords.z <= tex2D<float>(texShadowMap, coords.xy - offset.zy).r)
                                           ? 1 : 0; // top right

        return c * 0.25;

}
```

For random sampling, the `PCF_Random()` function is used. See the "Percentage Closer Soft Shadows" section.

VSM

This implementation requires both depth and squared depths to be stored. We use a `SCE_GXM_COLOR_FORMAT_F32F32_GR` format color surface to store both values. We then apply separable blur on this render target and again store the results in a `SCE_GXM_COLOR_FORMAT_F32F32_GR` type surface.

The depth is calculated in the vertex shader (`depth_map_v.cg`) by getting the distance between the light position and the vertex and then dividing this distance by the far clip plane distance of the light to obtain depth in the range `[0, 1]`.

```
float vert2Light = length(lightPos.xyz - worldPos.xyz);
oDepth = (vert2Light / lightPos.w) + DEPTH_BIAS;
```

The depth and the squared depth are then stored into the render target for each pixel in the fragment shader (`depth_map_f.cg`).

```
float2 main(float vDepth : TEXCOORD) : COLOR
{
    return float2(vDepth, vDepth*vDepth);
}
```

The next step applies a separable blur filter on the resulting render target, as shown in the following filtering sample code (`blur_vsm_f.cg`):

```
float2 main( uniform half4 fParams, // xy defines the x or y direction for the
                                    // separable blur; .zw stores the filter
                                    // size and the texel size.
             float2 vTexCoord : TEXCOORD0,
             uniform sampler2D vDepthTexture : TEXUNIT0
    ) : COLOR
{
    float2 fragColor = tex2D<float2>(vDepthTexture, vTexCoord);
    half c = (fParams.z == 2);
    fragColor += tex2D<float2>(vDepthTexture, vTexCoord - 2.0 * offset);
    fragColor += tex2D<float2>(vDepthTexture, vTexCoord - offset);
    fragColor += tex2D<float2>(vDepthTexture, vTexCoord + offset);
    fragColor += tex2D<float2>(vDepthTexture, vTexCoord + 2.0 * offset);


    fragColor *= (1.0 - 0.8 * c);


    return fragColor;
}
```

By using the same buffer to store depth and squared depth, we get high-precision averaged values in a single step, which improves the performance of the VSM technique.

When the filtered color buffer is available, the variance-based shadow contribution is computed using the following function:

```
// Variance Shadow Mapping:
half VSM_Filter(uniform sampler2D vsmDepthMap,   // Stores mean depth and
                                                 // squared depth.
    half4 coords, // Texture coordinates.
    float zVal)    // Light to current fragment distance.

{
    half2 uvCoord = coords.xy / coords.w;

    float2 moments = tex2D<float2>(texDepthMap, uvCoord);

    float E_x2 = moments.y;
    float Ex_2 = moments.x * moments.x;
    float variance = E_x2 - Ex_2;
    variance = max(variance, VSM_EPSILON);
    float mD = (moments.x - zVal);
    float mD_2 = mD * mD;
    float p = variance / (variance + mD_2);
    half shadowTerm = max( p, zVal <= moments.x);

   return shadowTerm;
}
```

Percentage Closer Soft Shadows

Blocker Search

This step returns average blocker depth in a search region along with the number of blockers. This requires sampling the depth map along the light direction and a search for any blockers. A maximum of 4 samples are used, which are picked from the poisson-disk sample set. The fragment shader `pcss_shadow_f.cg` contains the code for this, as follows:

```
#define POISSON_COUNT 4

float2 FindBlocker(uniform sampler2D shadowTex,
                   float4 coords,     // Shadow map texture coords.
                   float2 params      // .xy contain search width and texel
                                       // size.
)
{
    float blockerSum = 0;
    float blockerCount = 0;

    float3 texCoord = coords.xyz / coords.w;
    for (int i = 0; i < POISSON_COUNT; i++ ){
         float2 offset = poissonDisk[i] * params.x * params.y;
         float shadowMapDepth = f1tex2D(shadowTex, texCoord.xy + offset);

         float c = ( shadowMapDepth < texCoord.z )
         blockerSum += c*shadowMapDepth;
          blockerCount+=c;
    }

    return float2(blockerSum,  blockerCount);
}
```

Penumbra Estimation

The blocker depth from the above step is used to compute the filter radius, which gives the extent of the penumbra. The function is:

```
float FilterRadius(float zReceiver, // Receiver depth
                   float2 zBlocker, // Calculated blocker depth
                   float lightSize,  // Light size
                   float  zNear)     // Distance of the near plane
{
    float penumbraRatio = (zReceiver - zBlocker.x) / zBlocker.x;
    return penumbraRatio * lightSize * zNear / zReceiver;
}
```

Filtering

The filter radius defines the region around the current fragment over which the samples can be selected using the PCF technique. A poisson disk is used to select random samples within this region. The function is the same as that used in the PCF implementation above, except for the use of the calculated filter radius from penumbra estimation for the sample search:

```
// This function performs PCF filtering but uses random samples distributed
// over a Poisson Disk.
float PCF_Random( uniform sampler2D texShadowMap, // Depth Map
                  float4 coords,    // Projective texture coords
                  float gridSize,   // Filter size
                  float texelSize) // Texel size
{
```

```
        coords = coords / coords.w;
        float2 offset = texelSize * gridSize * cords.w * float2(1.0);

        float c =(coords.z <= tex2D<float>(texShadowMap, coords.xy -
                                poissonDisk[0] * offset).r) ? 1 : 0; // top left
        c += (coords.z <= tex2D<float>(texShadowMap, coords.xy + poissonDisk[0]
                                * offset).r) ? 1 : 0; // bottom right
        c += (coords.z <= tex2D<float>(texShadowMap, coords.xy + poissonDisk[0]
                                * offset).r) ? 1 : 0; // bottom left
        c += (coords.z <= tex2D<float>(texShadowMap, coords.xy - poissonDisk[0]
                                * offset.zy).r) ? 1 : 0; // top right

        return c * 0.25;
}
```

Cascaded Shadow Map

The samples of cascaded shadow maps are located at:

```
SDK/target/samples/sample_code/graphics/tutorial_shadow_techniques/cascaded_
shadow_maps/
```

The implementation first requires splitting the view frustum into multiple cascades. This is done by passing the maximum number of splits for the scene and the weight factor for setting up the split widths:

```
m_csmTechnique.setSplitDistance(m_splitCount, m_splitWeight);
```

Frustum Splitting

Based on the split count and the weight for setting up the width, the near and far planes for each of the splits are computed:

```
float camNear = viewCam.getNearClip();
float camFar = viewCam.getFarClip();
float lambda = m_splitWeight;
float ratio = camFar/camNear;
m_shadowSplits[0].m_csmFrustum.m_nearPlane = camNear;
for(int i=1; i<m_splitsCount; i++)
{
    float coeff = i / (float)m_splitsCount;
    m_shadowSplits[i].m_csmFrustum.m_nearPlane = lambda*(camNear *
        powf(ratio, coeff)) + (1-lambda)*(camNear + (camFar - camNear)* coeff);
    m_shadowSplits[i-1].m_csmFrustum.m_farPlane =
    m_shadowSplits[i].m_csmFrustum.m_nearPlane * 1.005f;

}
m_shadowSplits[m_splitsCount-1].m_csmFrustum.m_farPlane = camNear;
```

Setting Up Shadow Maps

Next, the depth/stencil surface that is used to store the shadow map for each frustum is initialized. We use a single depth/stencil surface to render the depth map for each of the cascades. Memory is allocated to store the data, and the depth stencil surface is enabled to force the write of depth values to memory. The allocation is the same as for the Soft Shadow implementation, except that the map dimensions are set to 1024x1024, with each cascade having a map size of 512x512:

```
SceGxmDepthStencilSurface m_shadowMapDepthSurface;
void* m_pShadowMapBufferData;
// set up the surface
retCode = sceGxmDepthStencilSurfaceInit(
          &m_shadowMapDepthSurface,
          SHADOW_MAP_DEPTH_FORMAT,      // SCE_GXM_DEPTH_STENCIL_FORMAT_DF32
          SCE_GXM_DEPTH_STENCIL_SURFACE_TILED,
```

```
            alignedWidth,     // set to 1024 for CSM
            m_pShadowMapBufferData,
            NULL);

sceGxmDepthStencilSurfaceSetForceStoreMode( &m_shadowMapDepthSurface,
         SCE_GXM_DEPTH_STENCIL_FORCE_STORE_ENABLED);
```

Calculating Light's View Frustum

Next calculate the corner points for each frustum split. The corner points of each camera frustum are then transformed using the light's model view matrix, and max/min bounds for each light frustum are computed so as to enclose each camera frustum slice. These max/min extends are used to compute the orthographic projection matrix. A view projection matrix and a shadow projection matrix are also computed for each frustum.

```
for(int i=0; i<m_csmTechnique.getSplitCount(); i++)
{
    // for each camera frustum, compute corner points in world space
    m_csmTechnique.updateFrustumPoints(m_csmTechnique.getSplitFrustum(i),
                          m_cameraPosition, m_cameraViewTarget);
    m_csmTechnique.calculateLightData(lightDir, i);
    // compute the light's view frustum slice to enclose camera's view frustum
    m_csmTechnique.getCropMatrix(i, m_meshRenderData, m_doTightFrustum,
                          m_doStabilizeShadowmap, SHADOW_MAP_DIMENSION);

    // Store shadow transformation matrices for the active buffer, with the offset
    // and scale included for each split and already transposed
    Matrix4 *shadowMatrices = m_pShadowMatrices[s_activeBuf];
    shadowMatrices[i] = transpose(m_pOffsetMatrices[i] *
                            m_csmTechnique.getShadowMapMatrix(i));

}

// For each frustum, the function below calculates the orthographic projection
// matrix,the view projection matrix, and the shadow projection matrix.
float CSMTechnique::getCropMatrix(uint32_t splitIndex, MeshRenderData *m,
          bool tightFrustum, bool stabilizeShadowmap, uint32_t shadowMapRes)
{

    Vector3 minBBPos(10000.0f);
    Vector3 maxBBPos(-10000.0f);

    Matrix3 tMat = m_shadowSplits[splitIndex].m_lightViewMatrix.getUpper3x3();
    Matrix4 lightViewMatrix = Matrix4(tMat, Vector3(0,0,0));

    // Transform camera view frustums' corner points to light's space to calculate
    // light frustum extends. You can also add steps to stabilize shadow
    // to avoid jittering artifacts when moving the camera, as well as use the
    // option to reduce empty regions by correctly mapping frustum to the object
    // within the cascade. For detailed code, check the sample tutorial in the
    // SDK.
    for(int i=1; i<8; i++)
    {
        Vector4 lightSpacePos = lightViewMatrix *
                        Vector4(m_shadowSplits[splitIndex].m_csmFrustum.
                        m_points[i], 1.0f);
        minBBPos = minPerElem(minBBPos, lightSpacePos.getXYZ());
        maxBBPos = maxPerElem(maxBBPos, lightSpacePos.getXYZ());
    }
```

```
        // Calculate orthographic matrix for the calculated extends
        m_shadowSplits[splitIndex].m_projectionMatrix =
                                Matrix4::orthographic(minBBPos.getX(),
                                maxBBPos.getX(),minBBPos.getY(), maxBBPos.getY(),
                                -maxBBPos.getZ() - zEps, -minBBPos.getZ() + zEps);

        // Calculate the view projection matrix for the frustum
        m_shadowSplits[splitIndex].m_viewProjectionMatrix = m_lightViewMatrix *
                                m_shadowSplits[splitIndex].m_projectionMatrix;

        // Use bias matrix to compute shadow's texture projection matrix
        Matrix4 biasMatrix = Matrix4::scale(Vector3(0.5f, -0.5f, 0.5f));
        biasMatrix.setTranslation(Vector3(0.5f, 0.5f, 0.5f));

        m_shadowSplits[splitIndex].m_shadowMapMatrix =  biasMatrix *
    m_shadowSplits[splitIndex].m_viewProjectionMatrix;

    }
```

Generating Shadow Maps

The depth surface and the view projection matrix are set corresponding to the current frustum slice, and the scene showing only the back faces of the objects are rendered from the light's view point. This is done for each frustum slice. Because we are using a single shadow map, we first need to set the viewport for each cascade and then render the scene. The viewport is computed at initialization corresponding to each cascade so that all sub-maps can be packed in. When generating a shadow map for a cascade, we set the corresponding viewport as:

```
// enable viewport transform
sceGxmSetViewportEnable(pContext, SCE_GXM_VIEWPORT_ENABLED);

// set region clip mode and area on surface for given cascade
sceGxmSetRegionClip(pContext, SCE_GXM_REGION_CLIP_OUTSIDE,...);

// set values to define map viewport to the region for that cascade
sceGxmSetViewport(pContext, ...);
```

After the viewport is set to the sub-region of the single-depth surface, the corresponding cascade is rendered onto that region:

```
Matrix4 worldViewProj = m_csmTechnique.getViewProjectionMatrix(index);

// cull front faces
sceGxmSetCullMode(pContext, SCE_GXM_CULL_CW);

// set depth bias and offset
sceGxmSetFrontDepthBias(pContext, m_shadowOffset, m_shadowBias);

// set shader and shader parameters for model with index modelIndex
m_sceneDepthShader[modelIndex].setShaderParams(pContext, &worldViewProj,
                                    m_lightPosition);

// Render model with index modelIndex
sceGxmSetVertexStream( pContext, 0, m_meshRenderData[modelIndex].meshData.
                            pMeshTriangles->getVertexBuffer()->getData());
sceGxmSetVertexStream( pContext, 1, m_meshRenderData[modelIndex].meshMat );
sceGxmDrawInstanced( pContext, SCE_GXM_PRIMITIVE_TRIANGLES,
                  ...// parameters to draw indexed geometry with multiple
                     // instances);
```

ESM

This implementation requires rendering the depth to a color surface corresponding to each cascade. We use a `SCE_GXM_COLOR_FORMAT_F32_R` format color surface to store this value. We then apply separable blur on the render target corresponding to each cascade and again store the results in a `SCE_GXM_COLOR_FORMAT_F32_R` surface by setting up the corresponding viewport.

The depth is calculated in the vertex shader (`depth_map_v.cg`) by getting the distance between the light position and the vertex and then dividing this distance by the far clip plane distance of the light to obtain depth in the range `[0, 1]`.

```
float vert2Light = length(lightPos.xyz - worldPos.xyz);
oDepth = vert2Light / lightPos.w;
```

In the fragment shader (`depth_map_f.cg`) linked with the vertex shader, the depth with a bias is stored in the render target during the pass for each cascade.

```
float main(float vDepth : TEXCOORD) : COLOR
{
    return float(vDepth + DEPTH_BIAS);
}
```

The next step applies a two-pass separable blur filter on the resulting render target, as shown in the following sample code for filtering (`blur_esm_f.cg`); this code uses 3 texture fetches in each direction:

```
float main(  float2 vTexCoord0 : TEXCOORD0,
             float2 vTexCoord1 : TEXCOORD1,
             float2 vTexCoord2 : TEXCOORD2,
             uniform sampler2D vDepthTexture : TEXUNIT0
    ) : COLOR

{
    float fragColor = tex2D<float>(vDepthTexture, vTexCoord0);
    fragColor += tex2D<float>(vDepthTexture, vTexCoord1);
    fragColor += tex2D<float>(vDepthTexture, vTexCoord2);

    fragColor *= 0.333;

    return fragColor;
}
```

The step is performed only on the active cascades used for the shadow.

Final Scene Rendering

For the final rendering of the shadows, the vertex streams are set, the uniform buffers (shadow texture projection matrix), and all texture buffers for the vertex and fragment shaders:

```
Matrix4 worldViewMatrix = m_viewMatrix * worldMatrix;
Matrix4 vpMatrix = m_camera.buildProjectionMatrix() * worldViewMatrix;

// set up main ESM shader and shader parameters
m_esmShadowShader[modelIndex].setShaderParams(pContext, &vpMatrix,
          &m_meshRenderData[modelIndex].meshMat[0],... // other shader params
);

sceGxmSetVertexStream( pContext, 0, m_meshRenderData[modelIndex].meshData.
                                pMeshTriangles->getVertexBuffer()->getData());
sceGxmSetVertexStream( pContext, 1, m_meshRenderData[modelIndex].meshMat );
sceGxmSetFragmentUniformBuffer(pContext, 0, m_pShadowMatrices[s_activeBuf]);
```

In the vertex shader (`scene_render_v.cg`), the shadow's projective texture coordinates are set corresponding to each frustum slice's shadow sub-map.

```
void main(float3 aPosition,
float2 aTexCoord,
```

```
    float3 aNormal,
    float4x4 aInstancedParam,
    uniform float4x4 viewProj,
    ...
    out float3  vTexCoord : TEXCOORD0,
    out float4  vWorldPos : TEXCOORD1
    )

    {
        vWorldPos = mul(float4(aPosition, 1.f), aInstancedParam);

        vPosition = mul(vWorldPos, viewProj);


        // Compute the lighting/ color terms

        ...

        vTexCoord.xy = aTexCoord;
        vTexCoord.z = length(lightVec) / lightPos.w;
    }
```

The final shadow computation is either done using the standard shadow-filtering technique or the pre-filtered ESM. The fragment shader (`scene_render_f.cg`) uses the standard technique, with the far plane bounds passed to it for each frustum slice based on which shadow texture coordinates are selected for sampling the depth. The shadow texture coordinates are used for fetching the depth sample from the corresponding sub-map on the depth/stencil surface and computing the shadow term, which is then combined with the color to render the final scene with shadows.

```
#define SPLIT_COUNT 4

uniform float4x4 shadowMatrices[SPLIT_COUNT] : BUFFER[0];

  half4 main(uniform half4 shadowParams,
             uniform float4 farBound,
             float4 vFragDepth : WPOS,
             half4 vColor : COLOR0,
             float3 vTexCoord : TEXCOORD0,
             float4 vWorldPos : TEXCOORD1,
             uniform sampler2D modelTex : TEXUNIT0,
             uniform sampler2D shadowTex : TEXUNIT1
             ): COLOR
  {

      half4 albedoCol = tex2D(modelTex, vTexCoord.xy);

      half4 splitCol[4] =
      {
           half4(0.8, 0.5, 0.5, 1.0),
           half4(0.5, 0.8, 0.5, 1.0),
           half4(0.5, 0.5, 0.8, 1.0),
           half4(0.5, 0.8, 0.8, 1.0)
      };

      // find the appropriate depth map to look up in based on the depth of this
      // fragment
       int cascadeIdx = dot(float3(vFragDepth.zzz > farBound.xyz), 1);

       float4 shadowCoord = mul(shadowMatrices[cascadeIdx], vWorldPos);
```

```
        // Calculate shadow contribution
        half shadowTerm = h1tex2D(shadowTex, shadowCoord.xyz); ;
        shadowTerm = lerp(1.0, shadowTerm, shadowParams.z);
        shadowTerm = 0.2 + shadowTerm * 0.8;
        half4 fragmentCol = lerp(albedoCol, splitCol[cascadeIdx], shadowParams.y)
                                                * vColor * shadowTerm;

    return fragmentCol;
}
```

For ESM, the shadow contribution `shadowTerm` is calculated using the function `ESM_Filter()` in the fragment shader (`esm_shadow_f.cg`).

```
    half shadowTerm = ESM_Filter(esmDepthTex, shadowCoord, vTexCoord.z);
```

The shader renders the cascades by applying an exponential-based shadow test using the filtered color buffer. The function `ESM_Filter()` used is:

```
// Exponential Shadow Mapping:
float ESM_Filter(uniform sampler2D texDepthMap, // Stores exponential depth
    half4 coords, // Texture coordinates.
    half zVal)    // Light to current fragment distance.

{
    half depth_scale = 30.0;
    half overdark = 0.9;
    half biasEpsilon = 0.2;


    half occDepthVal =  tex2Dlod<half>(texDepthMap, half4(coords.xy, 0, 0));
    half recDepthVal = depth_scale * zVal;

    half shadowTerm =  exp(overdark * (depth_scale * occDepthVal - recDepthVal));
    shadowTerm = min(1, shadowTerm);

    return shadowTerm;
}
```