4.1 Practical Cascaded Shadow Maps

FAN ZHANG, ALEXANDER ZAPRJAGAEV, ALLAN BENTHAM

INTRODUCTION

Like any other visual effect in real-time applications, practical shadow algorithms should allow developers to trade cost and quality. Although it is a popular real-time shadowing technique, *shadow mapping* [Williams78] cannot achieve this goal in complicated scenes, because it does not adequately trade performance cost for quality. Numerous extensions to this algorithm have been developed toward achieving this goal. *Cascaded shadow maps* (CSMs) [Engel06][Forsyth05] are an example of such an extension and have recently been attracting more attention with graphics hardware becoming more powerful. This technique can handle the *dueling frusta* case in shadow mapping, in which the light and view directions are nearly opposite. Furthermore, unlike warping algorithms [Stamminger02][Wimmer04][Martin04], CSMs do not require careful tuning and special tricks to handle extreme cases. We note that a similar idea was independently developed by Zhang et al. in the concurrent work on *parallel-split shadow maps* (PSSMs) [Zhang07][Zhang06]. This paper does not differentiate between the papers, as the basic idea is the same.

Although the prototyping of CSMs is quite simple, integrating CSMs into real games requires us to carefully consider several issues, some of which were not mentioned in the original CSM algorithm. This article discusses practical implementation issues of CSMs, which we have experienced in the Unigine project (see Figure 4.1.1), outlined as follows.

FIGURE 4.1.1 Cascaded shadow maps in Unigine.

- **Flickering of shadow quality:** This problem is also referred to as *swimming*, which exists in all view-dependent shadow-mapping techniques. To improve available shadow map texture resolution, a popular method is to focus the shadow map on visible objects. Such view-dependent processing can result in different rasterizations in consecutive frames. The flickering of shadow quality is thus perceived. To see this artifact, please refer to the video on the accompanying DVD-ROM. This article presents two methods to handle this issue.

- **Storage strategy:** In CSMs, shadow map textures can be stored in a variety of ways. For example, the most intuitive way is to store each shadow map in a separate texture. However, this storage strategy places considerable demand on the available interpolators. As a result, it is not an ideal choice for complex shading systems that already require a large number of simultaneous textures. Therefore, we should choose the storage strategy that best suits the requirements of the project. This article presents a detailed discussion of different strategies.

- **Non-optimized split selection:** The *practical split scheme* proposed by Zhang et al. [Zhang07] provides a practical solution for choosing the corresponding shadow map for a given pixel being rasterized. However, in some extreme cases such as when the light and view directions are nearly parallel, this scheme is not optimal. This article presents the solution we used in the Unigine project.

- **Correct computation of texture coordinates:** In typical CSM implementations, the vertex shader feeds all possible texture coordinates to the pixel shader for texture lookup. An alternative is to output the world-space position from the vertex shader. The pixel shader computes the texture coordinates from this world-space position after the split index is determined. The major advantage of this alternative is only using a single interpolator. However, as we will explain in this article, this method is mathematically incorrect because it results in an incorrect interpolation during rasterization.

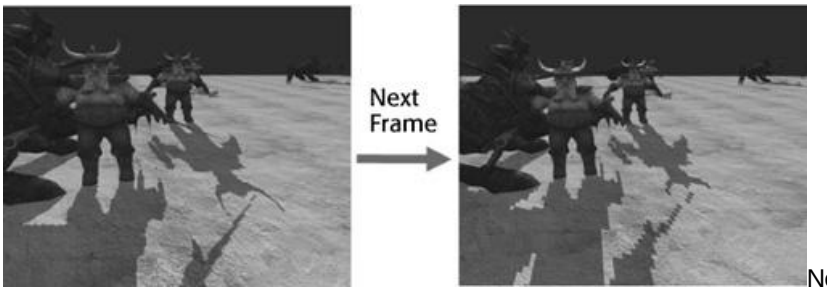
- **Filter across splits:** A filtering operation is usually needed in shadow mapping to produce anti-aliased shadows. A recent development in this field is *prefiltering* (i.e., filtering happens before depth comparison) the shadow map texture by exploiting the built-in filtering capabilities available on graphics hardware. This is made possible by the insightful ideas presented in literature on Variance Shadow Maps (VSMs) [Donnelly06][Lauritzen07b][Lauritzen08], Convolution Shadow Maps (CSMs) [Annen07], and Exponential Shadow Maps (ESMs) [Salvi07] [Annen08]. Though these filtering methods and CSMs are orthogonal, combining them requires careful consideration of discontinuity at split positions. This article explains how this problem occurs and presents two solutions.

The remainder of this article details each of the above issues and provides a summary.

FLICKERING OF SHADOW QUALITY

The flicking issue results from different rasterizations of the shadow maps as frames lapse, and is visually represented by the swimming of the shadow quality. Figure 4.1.2 illustrates the flickering problem in a scene containing a static light and a moving observer. Notice the degradation of quality between the two consecutive frames. Such a large change between two consecutive frames causes this degradation to be very noticeable to the user. Ideally, if such degradation is unavoidable, it should happen gradually.

FIGURE 4.1.2 Flickering issue. The shadow quality significantly changes between two consecutive frames, in which the light is static and the observer is moving.



Next Frame

Here we have an important note: Resolving the flickering issues for any given shadow map resolution will still not help reduce the spatial aliasing. In other words, the flickering issue aggravates the spatial aliasing in sequential frames. If a low-resolution shadow map produces jaggy shadows in a static frame, even if you handle the flickering issue when the light and/or viewer is moving, the jaggy shadow boundaries will not be sharp because spatial aliasing requires more dense sampling rates. The solutions to the flickering issue presented in this section just make your shadow quality view-independent.

To better understand how different rasterizations could aggravate the flickering issue, we demonstrate a few representative scenarios in Figure 4.1.3. This figure shows you how a point of interest (denoted by the red dot) is rasterized into texture pixels (denoted by green dots).

- **Scale:** In the top case in the figure, when the texel size (usually measured in the world space) of the shadow map changes in the next frame, the texel corresponding to the point of interest is now different. The shadow-testing result thus might be different from that in the previous frame. We refer to this case as the *scale problem*.

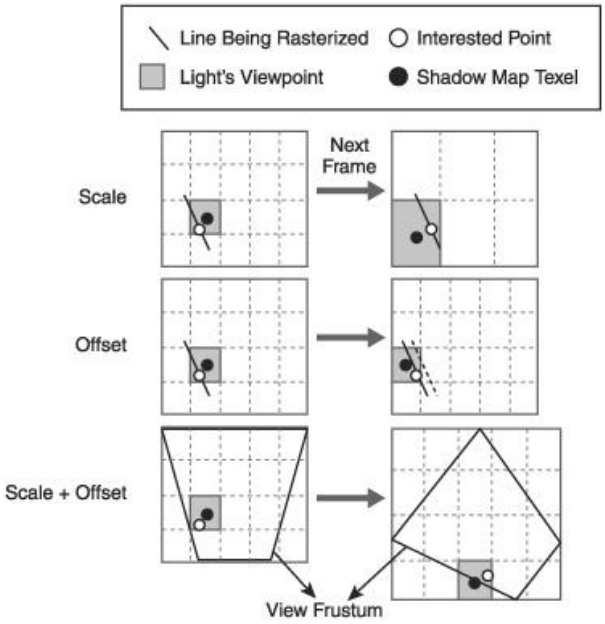
- **Offset:** The middle case illustrates different rasterizations of the point of interest, which is caused by a sub-texel offset due to the limited precision of the floating numbers. Note that this case happens almost every time your camera is changed. The reason is that transforming a point from the object space to the texture space (by vector-matrix multiplication) always results in a loss of precision. We refer to this case as the *offset problem*.

- **Scale + offset:** In practice, scale and offset problems usually happen together, as shown in the bottom case. To increase the available shadow map resolution, a popular and useful preprocessing technique is to focus the shadow map on the bounding shape (illustrated by the axis-aligned bounding box) of visible objects (simply represented by the view frustum). As we can see, due to the instability of the bounding box, when the viewer is moving, the shadow map can be significantly changed such that both the scale and offset problems simultaneously occur.

We present two methods to remove the flickering effect between different rasterizations.

Each of them has its disadvantages and advantages. Depending on if shadows are required to be filtered or not, you can choose one of these methods as the solution to this problem in your project. Our solutions to the flickering issue are intended to make your shadow rasterization view-independent. The spatial aliasing caused by insufficient shadow map resolution needs to be handled separately.

FIGURE 4.1.3 Three example scenarios of the flickering problem.



Top row: Varying of the texel size (i.e., scale problem). Middle row: Sub-texel offset (i.e., offset problem).

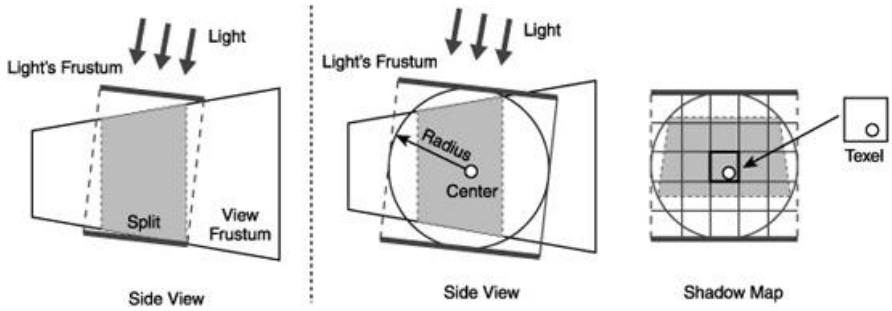
Bottom row: Focusing the shadow map on the bounding box of the view frustum results in both scale and offset problems.

EXACT SOLUTION

As explained earlier, an exact solution to the flickering needs to solve both scale and offset problems. In such exact solutions [Valient07] (see Figure 4.1.4), each shadow map is focused on the split's bounding sphere rather than the split's bounding box. The symmetric characteristic of the sphere removes the scale problem, because the shadow map size (in the world space) is now stabilized. Furthermore, to avoid the offset problem, the light's position is accordingly adjusted in the world space such that the sphere center is contained by the center of the shadow map. As a reference, we show the pseudo code for this solution in Listing 4.1.1.

The advantage of this solution is that flickering is correctly removed, as shown in Figure 4.1.5. On the other hand, although the flickering problem is well handled, a considerable portion of shadow map resolution is wasted. More importantly, most modern games use filtering techniques to help anti-aliasing; an approximate solution to the flickering issue is usually enough because a slight flickering will be hidden by filtering anyway. Finally, warping algorithms cannot use this solution due to the nonuniform texel distribution.

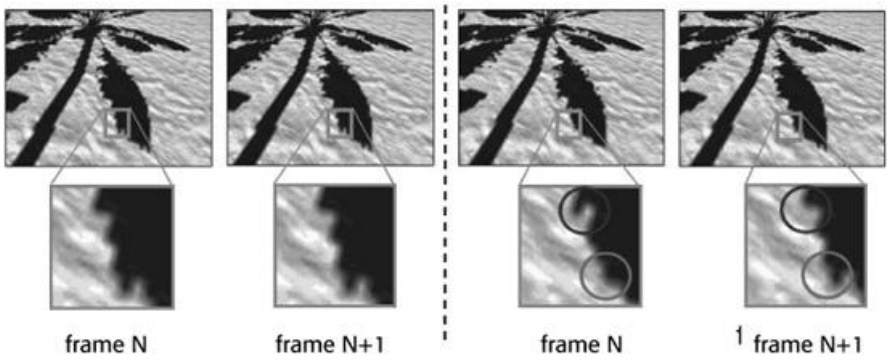
FIGURE 4.1.4 Left: Typical CSM implementations focus each shadow map on the axis-aligned bounding box in the light's post-perspective space. Right: Exact solutions focus each shadow map on the bounding sphere of the corresponding split (left), and fix the position of the sphere center (measured by the texel size) in world space.



Listing 4.1.1 Exact solution to the flickering issue

```
//Step 1: calculate the light basis

vector3 direction = normalize(WORLD.getColumn3(2)); //z axis of the light
coordinates system

vector3 side = normalize(WORLD getColumn3(1)); //x axis of the light
coordinates system

vector3 up = normalize(WORLD. getColumn3(0)); //y axis of the light
coordinates system

// Update all splits

for(int i = 0; i < numSplits; i++){ //'numSplits' is the number of spWlits

   // Step 2: calculate the bounding sphere' bs' of each split 'split[i]'

   Sphere bs = CalculateBoundingSphere(split[i]);

   // get the sphere center's position in world space

   vector3 center = INVWORLDVIEW * bs.getCenter();

   //get the sphere radius

   float radius = bs.getRadius();

   // adjust the sphere's center to make sure it is transformed into the
center of the shadow map

  float x = ceilf(dot(center,up) * sizeSM / radiu) * radius / sizeSM; //
'sizeSM' is texture size

  float y = ceilf(dot(center,side) * sizeSM / radius) * radius / sizeSM;
   center = up * x + side * y + direction * dot(center,direction);

   // Step 3: update the split-specific view matrix 'matrixView[i]' and
projection matrix 'matrixProjection[i]'

   matrixProjections[i] = MatrixOrthoProjection(-radius, radius,
radius, -radius, near, far);

   vector3 origin = center - direction * far; // get the light's position

   matrixView[i] = MatrixLookAt(origin, center, up);
}
```

FIGURE 4.1.5 Comparison of shadow quality. Left: The shadows stay the same in two successive frames using the exact solution. Right: Flickering issue occurring in the original CSM implementation. The differences are marked by colored circles.



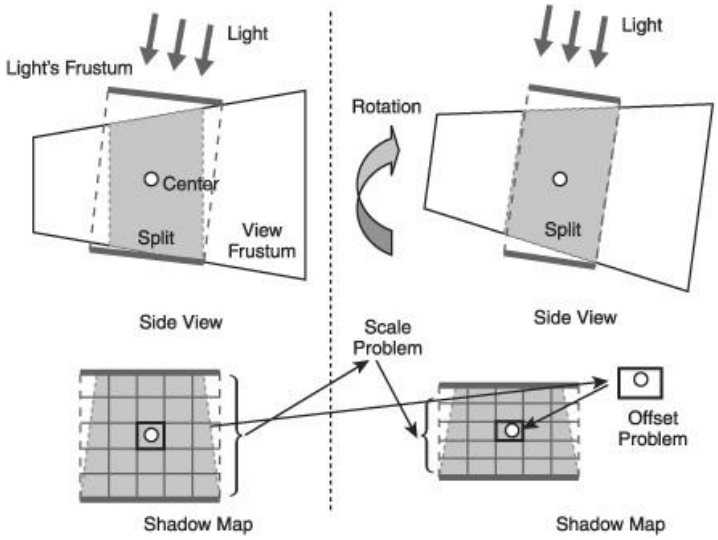
APPROXIMATED SOLUTION

As we have pointed out above, the main disadvantage of the exact solution is that a large amount of shadow map resolution is wasted. The reason is simple: The sphere is not the best approximation of the view frustum from the light's viewpoint. In many situations, it is not acceptable to sacrifice texture resolution *purely* to prevent flickering. An approximate solution that allows more efficient use of the texture resolution may be more desirable.

Let's revisit the flickering issue from a mathematical view. During the rasterization of the shadow map, every fragment's position is *quantized* in terms of the *measure unit*—the texel size. If the position is quantized differently across two frames, the offset problem happens. Therefore, removal of the offset problem requires stabilizing the quantization of positions relative to the frustum. Likewise, in order to avoid the scale problem, the same requirement is needed for the *scale value*—the change of texel size. In exact solutions, the texel size (in world space) stays the same due to the symmetry of the sphere; that is, the scale value is always 1.

To quantize the scale value, we assume the scale value varies smoothly in successive frames. This assumption is valid in many cases. With this assumption, we can quantize the scale value into a *pre-defined* range of discrete levels (see Figure 4.1.6). In our experiments, we empirically found 64 to be a good candidate for this. Quantization of offset values in exact and approximated solutions is the same. Pseudo code for the approximated solution is shown in Listing 4.1.2.

FIGURE 4.1.6 Approximated solution. The left and right columns show two frames before and after, respectively, of a slight rotation of the view frustum.



This solution results in much better utilization of shadow map resolution and much better shadow quality when combined with filtering techniques. On the other hand, the disadvantage is that, without filtering there is a possibility that some minor flickering may still occur.

Listing 4.1.2 Approximated solution to the flickering issue

```
// calculate the scale values

// Notes: in the light's post-perspective space, both width and height of
the view port are 2. The dimension of the shadow map for a specific split
is the bounding coordinates of the split's corner points in this space.
'maxX' and 'minX' stand for the maximum and minimum x values respectively.
```

```
'maxY' and 'minY' stand for the maximum and minimum y values respectively.

float scaleX = 2.0f / (maxX - minX); // scale value for x dimension

float scaleY = 2.0f / (maxY - minY); // scale value for y dimension

//the scale values will be quantized into 64 discrete levels

float scaleQuantizer = 64.0f;

// quantize scale

scaleX = 1.0f / ceilf(1.0f / scaleX * scaleQuantizer) * scaleQuantizer;

scaleY = 1.0f / ceilf(1.0f / scaleY * scaleQuantizer) * scaleQuantizer;


// calculate offset values

float offsetX = -0.5f * (maxX + minX) * scaleX; // offset value for x
dimension

float offsetY = -0.5f * (maxY + minY) * scaleY; // offset value for y
dimension

// quantize offset

float halfTextureSize = 0.5f * sizeSM;

offsetX = ceilf(offsetX * halfTextureSize) / halfTextureSize;

offsetY = ceilf(offsetY * halfTextureSize) / halfTextureSize;x
```

STORAGE STRATEGY

The original CSM algorithm does *not* impose any restriction on how to store shadow maps. These textures can be stored in separate textures [Zhang07][Engel06], packed into a single texture atlas [Valient07][Lloyd06], encapsulated into a cube map [Zhang07], or directly stored in a texture array on modern GPUs [Zhang07]. Packing CSMs into an atlas texture (see Figure 4.1.7) reserves more texture samplers for other purposes. Considering this, our project uses this strategy.

Variant storage strategies of CSMs are discussed below.

- **Texture arrays:** According to our experiments, storing CSMs in a texture array [Zhang07] works well on NVIDIA Geforce 8xxx graphics cards. However, sampling texture arrays is only supported by Direct3D 10.1 or above.

- **Cube maps:** In this method, additional shader instructions are needed for the lookup of texture coordinates and simulating the border color addressing mode [Zhang07]. Furthermore, if the number of splits is small (a typical setting is four), there's a waste of video memory because this method always allocates six shadow maps.

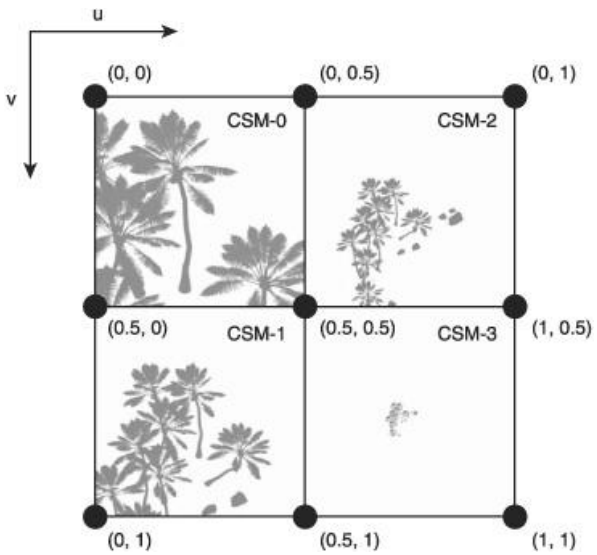
- **Multiple textures:** The simplest way is to store each shadow map in a separate texture [Zhang06]. This method, however, makes it difficult to implement filtering techniques (e.g., percentage closer filtering [PCF]) on DX9-level hardware because it requires the support of texture arrays. More importantly, considering that DX9-level hardware supports up to 16 textures, this method is not very suitable for complicated shading systems that use a large number of textures simultaneously.

- **Texture atlas:** In this method, all shadow maps are packed into a single large texture. Although the maximum texture size allowed by hardware imposes a restriction on the atlas resolution, this method brings a few practical advantages.

• Working equally on DX9 and DX10 hardware.
• Very efficient usage of video memory, especially when four splits are used.
• Reserving more texture samplers for other purposes.
• Filtering CSMs can be done in a single rendering pass.

With the above observations, the atlas strategy has become more and more popular in modern games. The default setting in Unigine uses four splits, which is enough in most cases. Although we are limited by maximum texture dimension (e.g., 2048×2048 on PS 2.0 hardware), in our experiments the shadow quality is far better *and* more robust than that of warping techniques with the same texture resolution. We show the pseudo code for computing texture coordinates in the atlas strategy in Listing 4.1.3.

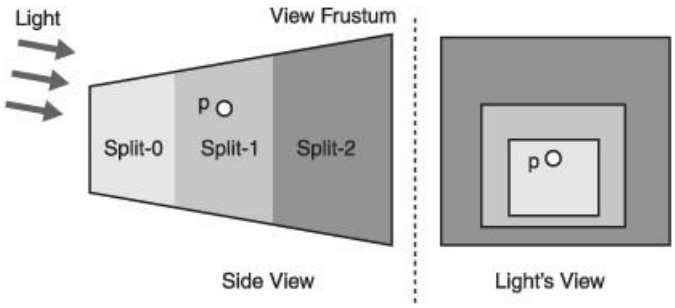
FIGURE 4.1.7 An example of the texture atlas containing four shadow maps. Texture coordinates for corner points of each shadow map are given also.



NON-OPTIMIZED SPLIT SELECTION

The original CSM algorithm uses the eye's view-space depth to select the shadow map for every pixel being rasterized. This method works well in the most general cases. However, once the light becomes almost parallel with the view direction, the shadow map selected for the current point might *not* be optimal. For example, consider a point *p* in the second split as shown in Figure 4.1.8. It is covered by all three shadow maps. On the other hand, in order to get the highest shadow quality at this position, we should choose the first (rather than the second) shadow map.

FIGURE 4.1.8 The best split index selection for *p* is 0 rather than 1 in the original CSMs.



An optimized split-selection schema in this case is illustrated in Figure 4.1.9. When the angle between the light direction and view direction is very small, the standard split selection wastes a lot of texture resolution. To further improve shadow quality in such cases, we always select the shadow map with higher resolution. The pixel shader in Listing 4.1.3 illustrates how to achieve this goal using the atlas strategy.

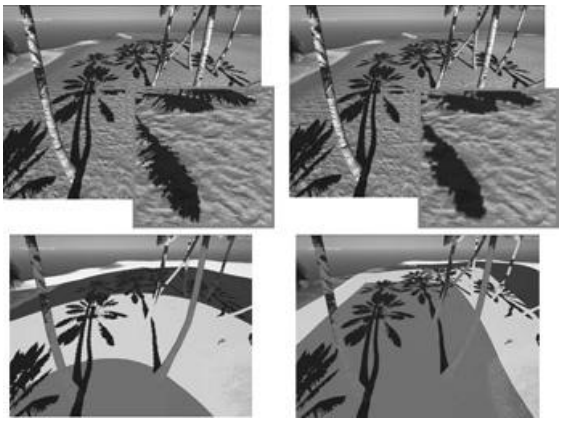
Listing 4.1.3 Pixel shader for the optimized split selection in the atlas strategy

```
//Note: 4 shadow maps are stored into an atlas as shown in Figure 4.1.4. '
matrixWorld2Texture[4]' stores texture matrices, 'PS_Input.world_position'
stores the world position in the input of pixel shader.



float shadow = 0.0;
// get the potential texture coordinates in the first shadow map
float4 texcoord = mul(matrixWorld2Texture[0], PS_Input.world_position);
// projective coordinates
texcoord.xyz = texcoord.xyz / texcoord.w;
// see Figure 4.1.4, if the range of x and y locates in [0, 0.5], then
this point is contained by the 1st SM
```

```
if(max(abs(texcoord.x - 0.25),abs(texcoord.y - 0.25)) >= 0.25) {
       // if this point is not contained by the 1st SM, then do the same
test for the 2nd SM
       // see Figure 4.1.4, 0<=x<=0.5 and 0.5<=y<=1 for the 2nd SM.
       texcoord = mul(matrixWorld2Texture[1], PS_Input.world_position);
       texcoord.xyz = texcoord.xyz / texcoord.w;
       if(max(abs(texcoord.x - 0.25),abs(texcoord.y - 0.75)) <= 0.25) {
              //test for the 3rd SM in which 0.5>=x>=1 and 0>=y>=0.5
              texcoord = mul(matrixWorld2Texture[2],
PS_Input.world_position);
       texcoord.xyz = texcoord.xyz / texcoord.w;
       if(max(abs(texcoord.x - 0.75),abs(texcoord.y - 0.25<= 0.25) {
                     //test for the last SM in which 0.5>=x>=1
and 0.5>=y>=1
             texcoord = mul(matrixWorld2Texture[3],
PS_Input.world_position);
             texcoord.xyz = texcoord.xyz / texcoord.w;
             if(max(abs(texcoord.x - 0.75),abs(texcoord.y - 0.75))
<= 0.25) {
                  shadow = 1.0;
             }
           }
        }
}
if(shadow != 1.0) {
      //get shadow value, where 'samaplerAtlas' is the sampler the atlas.
      shadow = tex2D(samaplerAtlas, texcoord);
}
```

As you can see from Figure 4.1.9, shadow rendering is improved in our optimized method. Notice that we've used the exact solution to handle the filtering issue in this figure. That's why the splits in the optimized method have circular boundaries.

FIGURE 4.1.9 Top row: shadow quality in the optimized method and original method. The shadow details are zoomed in the colored rectangles. Bottom row: colorized visualization of splits in two methods.



CORRECT COMPUTATION OF TEXTURE COORDINATES

Typical CSM implementations (e.g., [Zhang07]) output all possible texture coordinates from the vertex shader to the pixel shader. Subsequently, in the pixel shader, once we've determined which split the current pixel is in, the associated texture coordinates are used for texture lookup. Refer to the pseudo code for "Method 1" in Listing 4.1.4. The main disadvantage of this method is that several interpolators are used. For example, three interpolators are required for CSM(3) in Listing 4.1.4, where CSM(3) stands for splitting the view frustum into three splits. This situation becomes more serious when the number of interpolators is limited, which is very common for complex shaders.

In order to reduce the number of interpolators required in CSMs, another popular (but incorrect) alternative is to output only the world-space position from the vertex shader to the pixel shader, and then accordingly compute the texture coordinates in the pixel shader after the split index is determined. As you can see, the code in Listing 4.1.3 also uses this method. However, this method is *mathematically* incorrect (but NOT visually noticeable in most practical cases). This method is illustrated as Method 2 in Listing 4.1.4. In the next section, we will show that Method 2 actually results in an incorrect interpolation during rasterization. One note for

the pseudo code is that you can use a simple dot product to determine the split index in practice.

Listing 4.1.4 Comparison of two methods for computing texture coordinates

```
 Note: we assume 3 shadow maps are used in the following pseudo code.


/*************************Method 1*********************************/
//DATA STRUCTURE
struct VS_OUTPUT
{
   float4 position : POSITION; // screen-space posit
   float4 tex0 : TEXCOORD0; // texture position for CSM0
   float4 tex1 : TEXCOORD1; // texture position for CSM1
   float4 tex2 : TEXCOORD2; // texture position for CSM2
}
//VERTEX SHADER
float4 posWorldSpace = mul(VSInput.position, WORLD); // world-space position
VSOutput.position = mul(posWorldSpace, matrixViewProj); // screen-space
position
VSOutput.tex0 = mul(posWorldSpace, matrixTexture[0]); // texture position
for CSM0
VSOutput.tex1 = mul(posWorldSpace, matrixTexture[1]); // texture position
for CSM1
VSOutput.tex2 = mul(posWorldSpace, matrixTexture[2]); // texture position
for CSM2
//PIXEL SHADER
float shadow; //final illumination result
int split = ...; //we ignore the code for computing the split index 'split'
here

if (split > 1) //if the current pixel is in the first split
      shadow = tex2DProj(samplerCSM0, PSInput.tex0); //depth comparison
else if... //the rest code is ignored



/*************************Method 2*********************************/
//DATA STRUCTURE
struct VS_OUTPUT
{
   float4 position : POSITION; // screen-space posit
   float4 tex0 : TEXCOORD0; // world-space position
}
//VERTEX SHADER
VSOutput.position = mul(VSInput.position, WORLDVIEWPROJ); // screen-space
position
VSOutput.tex0 = mul(VSInput.position, WORLD); // world-space position

//PIXEL SHADER
float shadow; //final illumination result
float4 texCoords; //texture coordinates
int split = ...; //we ignore the code for computing the split index 'split'
here
if (split > 1) //if the current pixel is in the first split
{
     texCoords = mul(PSInput.tex0, matrixWorld2Texture[split]); //compute
texture coordinates
     shadow = tex2DProj(samplerCSM0, texCoords); //depth comparison
}
else if... //the rest code is ignored
```

Any attributes passed to the pixel shader are *perspective-interpolated* from the vertex shader output. Listing 4.1.5 shows that the texture coordinates used in the pixel shader are different between the two methods. It means that the interpolated texture coordinates in the second method are wrong in theory. The resultant visual artifacts are usually not noticeable when the scene is well tessellated. However, the problem can become noticeable for scenes that contain large triangles.

Listing 4.1.5 Comparison of the perspective-interpolation in two methods

```
/*******************Method 1 (correct perspective interpolation)
**********************/

//VERTEX SHADER
//denote 'eposition' and 'etex0' as two vectors, shown as follows
VSOutput.position ↔ (x, y, z, w)
VSOutput.tex ↔ (s, t, u, v)

//PIXEL SHADER
//texture coordinates are perspective-interpolated from the output of the
```

```
vertex shader,
PSInput.tex0 ↔ (lerp(s/w) / lerp(1/w), //x component
                lerp (t/w) / lerp(1/w), //y component
                lerp (u/w) /lerp(1/w), //z component
                lerp (v/w) /lerp(1/w) ) //w component
/*******************Method 2 (wrong perspective interpolation)
**********************/
//VERTEX SHADER
//denote "position" and "tex0" as two vectors, shown as follows
VSOutput.position ↔ (x, y, z, w)
VSOutput.tex0 ↔ (xworld, yworld, zworld, wworld)

//PIXEL SHADER
//world-space positions are perspective-interpolated from the output of
the vertex shader,
PSInput.tex0 ↔ ( lerp(xworld /w) / lerp(1/w), //x component
               lerp (yworld /w) / lerp(1/w), //y component
               lerp (zworld /w) /lerp(1/w), //z component
               lerp (wworld /w) /lerp(1/w) ) //w component
// texture coordinates are then
texCoords = mul(PSInput.tex0, matrixWorld2Texture) ≠ (s, t, u, v)!
```

As we explained in Listing 4.1.5, the correct computation of texture coordinates in the second rendering pass requires us to get the correct world-space position for each fragment. We have a short note here: There are various ways to correctly construct the world-space position as part of a *deferred shading* pass, which are beyond the scope of this article. This will require one extra vector-matrix multiplication, that is, the vector of the screen-space coordinates and the inverse matrix from screen-space to world space. Various methods exist for calculating the world position from screen-space coordinates. In particular, readers can refer to deferred shading methods for more details. Finally, you could simply pass the inverse matrix from your application to the GPU.

According to our experience in practice, in the case that the interpolators are quite important and limited, we suggest that developers use the alternative Method 2 *but* with a clear understanding of the potential problem behind it. A similar example in computer graphics i s *Gouraud shading*. Although the bilinear interpolation used in Gouraud shading is not theoretically correct, the shading result is usually fine for diffuse objects.

FILTERING ACROSS SPLITS

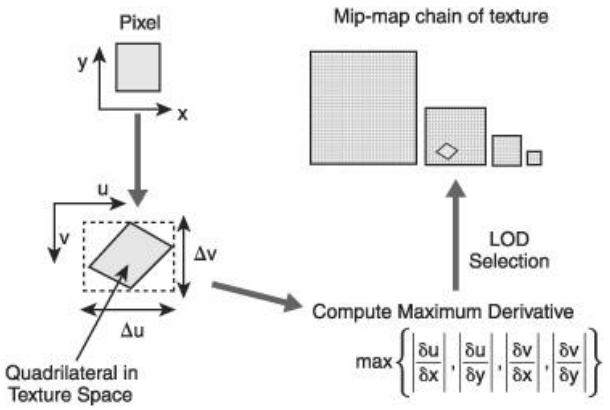
The splitting planes in CSM introduce discontinuities that can interfere with texture filtering. We need to pay special attention to filtering at split boundaries to avoid visual artifacts. This problem is very similar to filtering cube maps [Isidoro05]. Current hardware regards each face in a cube map as a separate 2D texture and does not filter across cube map faces. This can cause noticeable discontinuities to appear at face borders. Therefore, we need a practical solution for this issue in CSMs.

A significant advantage of pre-filtering techniques over PCF is that shadow maps can be filtered like standard textures by using the GPU hardware's built-in filtering functionality.

As illustrated in Figure 4.1.10, graphics hardware computes the screen-space texture derivatives (for a transformed quad) to determine the texture filtering width and *levels-of-detail* (LODs) selection in the *mipmap* chain [Akenine02]. This procedure works well for filtering a single texture, but it does not work across splits in CSMs.

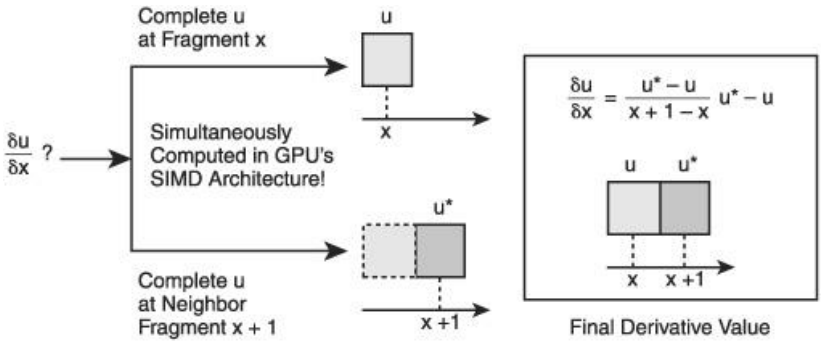
To understand why CSM splits do not work with hardware filtering, it will help to understand how the derivative instruction is implemented in graphics hardware. As illustrated in Figure 4.1.11, when computing the derivative of a variable *u*, the value of *u* is computed at both the current fragment and the neighbor fragment.

FIGURE 4.1.10 Mipmap LOD selection in hardware.



The difference between the two values is returned as the derivative result. Notice that, due to the nature of the SIMD (i.e., single-instruction-multiple-data) architecture in GPUs, the derivative computation is very efficient.

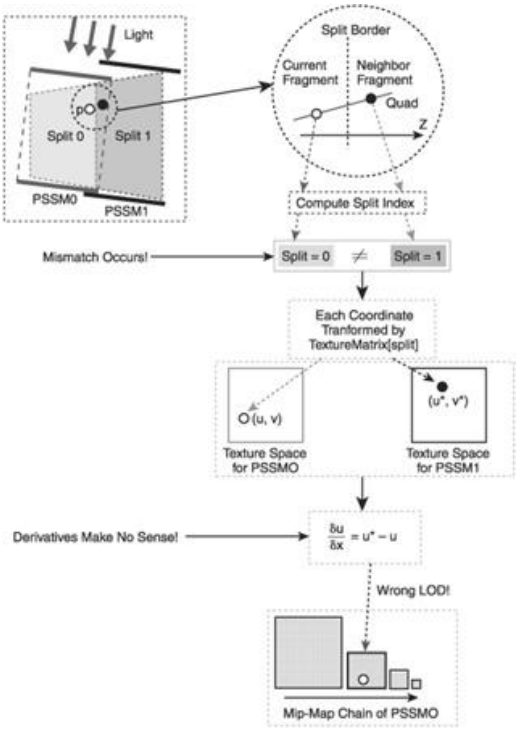
FIGURE 4.1.11 How a modern GPU computes derivatives.



The reliance of the derivative instruction on neighboring pixels causes it to produce invalid results across CSM splits (see Figure 4.1.12). Fragments on the same quad can belong to two different shadow maps along split boundaries, so the texture coordinates transformed by different texture matrices result in meaningless derivatives. As a result, noticeable "seam" artifacts appear at split boundaries.

Notice that this problem does not appear in techniques that filter *after* the depth comparison, such as *percentage closer filtering* (PCF) [Reeves87]. Filtering after the depth comparison avoids the hardware's built-in filtering and therefore avoids the derivative computation and split boundary artifacts.

FIGURE 4.1.12 Wrong LOD selection can happen when fragments in the same quad belong to different splits.



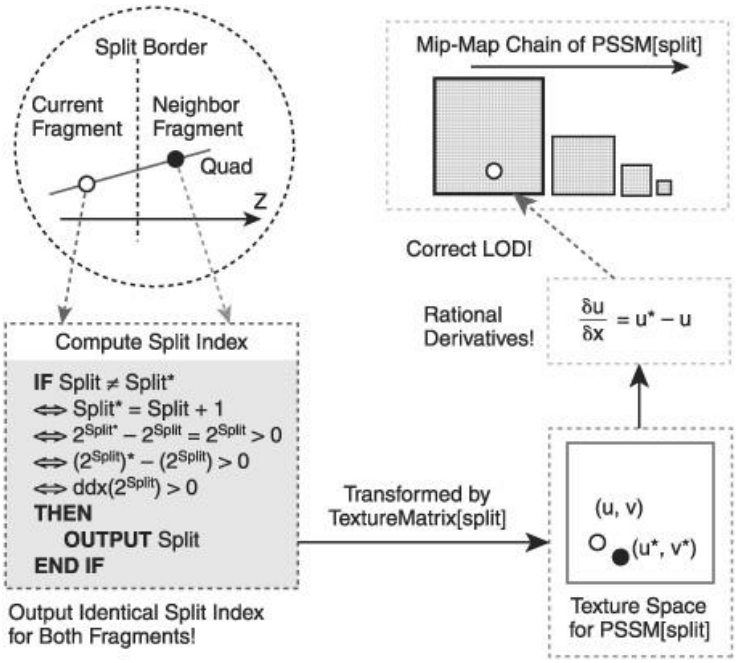
From our previous analysis, artifacts at split boundaries occur when an incorrect LOD is used in texture filtering. This problem was first analyzed and solved in *parallel-split variance shadow maps* (PSVSMs) [Lauritzen07a]. In theory, an analytic solution is needed to select the correct LOD in the mipmap chain. In practice, the approximate method proposed in PSVSMs solves this problem well without sacrificing performance. You can choose either one according to your implementation. In the following, we outline both methods.

METHOD USED IN PSVSMS

The idea is based on the observation that the problem results from different split indices being used for fragments in the same quad. In other words, the filtering discontinuity can be avoided by keeping split indices consistent over the quad.

Figure 4.1.13 illustrates how we can use the hardware's derivative instructions to implement this idea. For simplicity, let's consider the fragment in 1D space first. When the mismatch of split indices at neighboring fragments occurs, there's a difference of 1 between the two indices, denoted by *split* and $split^*$. To detect if the mismatch occurs, we just need to check if ***ddx****(split)* is non-zero because ***ddx****(split)* = $split^*$ – *split* (see Figure 4.1.11). Our goal is to keep the split index the same for both fragments, for example, *split* (notice that you can also choose $split^*$). However, since ***ddx****(split)* always returns 1, the problem is how to deduce the value of *split*. We use another smart trick employed with PSVSMs. Let's treat 2^{split} as a variable, ***ddx***$(2^{split}) = 2^{split^*} - 2^{split} = 2^{split}$. A simple *log* will give you the original split value. Even simpler, you can replace the *log* operation with a predefined lookup table to improve performance.

FIGURE 4.1.13 Solving mismatch of split indices in PSVSMs.



Generalizing the analysis to 2D screen-space, we need to compute all derivatives to deduce a consistent split index, as shown in Listing 4.1.6.

Listing 4.1.6 Keep the consistent split index fragments on the same quad

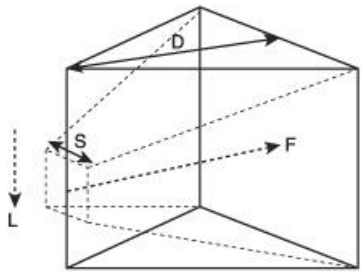
```
// CONSTANTS
const int SPLIT_NUM = 3; //number of splits

// PIXEL SHADER
int powerSplitIndex = pow(2, splitIndex); // 2splitIndex
int dx = abs(ddx(powerSplitIndex)); // d(2splitIndex)/ dx
int dy = abs(ddy(powerSplitIndex)); // d(2splitIndex)/ dy
int dxy = abs(ddx(dy)); // d2(2splitIndex)/dxdy
int split = max(dxy, max(dx, dy)); // get the maximum derivative value
if(powerSplitIndex>0) // if mismatch happens
   splitIndex = log(powerSplitIndex); // update split index
```

ANALYTIC METHOD

In this method, the derivative is no longer computed in the split-specific texture space because it will cause the wrong LOD selection, as previously explained. The idea of this method is based on the observation that there's obviously no problem for hardware to compute derivatives when using a single shadow map texture. For a given shadow map in CSMs, as long as we can find the relationship between the derivatives in standard shadow mapping (for the whole view frustum) and the derivatives in the texture space of the given shadow map (for a split), the correct split-specific derivatives should be derivable from the former. This idea is illustrated by Figure 4.1.14. On the other hand, it's very easy to tell there's only a scale difference between the derivatives in the two texture spaces. In Figure 4.1.14, you can easily tell there's a difference of translation and scale between the texture coordinates

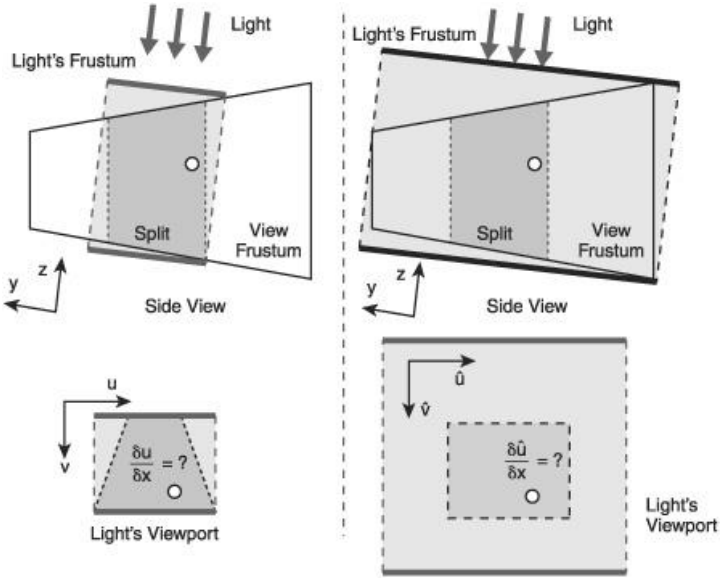
$d = \frac{D+S}{D}$ (for the whole view frustum) and u (for the split only).

Therefore, the difference between the derivatives $\mathbf{M} = \begin{bmatrix} d & 0 & 0 & F_x \\ 0 & d & 0 & F_y \\ 0 & 0 & d & F_z \\ 0 & 0 & 0 & S \end{bmatrix}$ and



(remember this is what we want) is a scale ratio. The corresponding pseudo code is illustrated in Listing 4.1.7.

FIGURE 4.1.14 Analytic method to compute derivatives.



Listing 4.1.7 Analytic method for computing derivatives

```
// CONSTANTS
// Refer to [Zhang07], transforming from the texture space for the whole
view frustum to the texture space for a specific split only requires a
SCALE transformation and a TRANSLATION transformation.
float2 scale; //scale values in x and y dimensions
float2 translate; //translation values in x and y dimensions
// PIXEL SHADER
// Step 1: Take derivatives in standard shadow mapping
// NOTE: 'PosLight' stands for the position in light's clip space for the
whole view frustum!
float2 LightTexCoord = PSInput.PosLight.xy / PSInput.PosLight.w; //
project texture coordinates
float2 dtdx = ddx(LightTexCoord); //take derivative
float2 dtdy = ddy(LightTexCoord); //take derivative


// Step 2: Apply split-specific scale/translate to texture coordinates and
derivatives
scale = scale * float2(0.5, -0.5);
translate = translate * float2(0.5, -0.5) + 0.5;
LightTexCoord = scale * LightTexCoord + translate; //NDC to texture space
dtdx *= Scale;
dtdy *= Scale;
```

Finally, as an optional improvement when you combine CSMs and filtering techniques, you can use a self-adjusting filtering kernel size when using filtering. Ideally, generating a consistent blurring effect over all splits needs a *varying* kernel size due to the foreshortening effect. Although a small and constant filtering width (e.g., 2×2) can still blur shadows well, a self-adjusted filtering size is necessary in some cases such as when applying warping algorithms into each shadow map. It's actually very easy to provide such a varying kernel. We just need to scale the original constant filtering width by the ratio of the size of the current split and to the size of whole view frustum from the light's point of view, as shown inListing 4.1.8.

Listing 4.1.8 Self-adjusted filtering kernel size

```
// Scale values for x and y respectively
float scaleX, scaleY;
```

```
// Compute the required scale for the current split in light's postperspective
space
// 'maxX' and 'minX' stand for the maximum and minimum x values respectively.
// 'maxY' and 'minY' stand for the maximum and minimum y values respectively.
scaleX = 2.0f / (maxX - minX);
scaleY = 2.0f / (maxY - minY);
// Assume the original constant filtering size is 2x2
float2 vFilteringKenerlSize(2.0f, 2.0f);
// Update the filtering size
vFilteringKenerlSize.x *= scaleX;
vFilteringKenerlSize.y *= scaleY;
```

Conclusion

This chapter has discussed a few practical issues in *cascaded shadow maps* (CSMs). For each of these issues, we have analyzed the cause and subsequently presented the solutions. We hope the methods presented in this article can help developers improve their CSM implementations and motivate further research on this topic.

Acknowledgments

Thanks to Sam Martin for his valuable review comments. Thanks to David Lam, Andrew Lauritzen, Adrian Egli, Oskari Nyman, and Marco Salvi for their early reviews. Special thanks to David Lam for his patient and careful proofreading, and to Andrew Lauritzen for insightful suggestions and sample code and for inspiring the second method for the filtering issue.

References

[Akenine02] Thomas Akenine-Moller and Eric Haines. *Real-Time Rendering* (2nd Edition). A. K. Peters Limited, 2002.

[Annen07] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. “Convolution Shadow Maps.” In *Proceedings of the Eurographics Symposium on Rendering 2007*, pp. 51–60.

[Annen08] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. “Convolution Shadow Maps.” In *Proceedings of Graphics Interface 2008*, pp. 155–161.

[Donnelly06] William Donnelly and Andrew Lauritzen. 2006. “Variance Shadow Maps.” In *Proceedings of the Symposium on Interactive 3D Graphics and Games 2006*, pp. 161–165.

[Engel06] Wolfgang Engel. “Cascaded Shadow Maps.” *ShaderX*^{5}, edited by Wolfgang Engel. Charles River Media, 2006, pp. 197–206.

[Forsyth05] Tom Forsyth. “Shadowbuffer Frustum Partitioning,” *ShaderX*^{4}, edited by Wolfgang Engel. Charles River Media, 2005, pp. 289–297.

[Isidoro05] John R. Isidoro. “ Filtering Cubemaps - Angular Extent Filtering and Edge Seam Fixup Methods,” Siggraph’05 Sketch Presentation, 2005. (also available at http://ati.amd.com/developer/SIGGRAPH05/Isidoro-CubeMapFiltering.pdf).

[Lauritzen07a] Andrew Lauritzen. “Parallel-Split Variance Shadow Maps,” In Proceedings of Graphics Interface 2007 (poster).

[Lauritzen07b] Andrew Lauritzen. “Summed-Area Variance Shadow Maps.” *GPU Gems3*, edited by Hubert Nguyen. Addison Wesley. 2007, pp. 157–182.

[Lauritzen08] Andrew Lauritzen and Michael McCool. “Layered Variance Shadow Maps,” In *Proceedings of Graphics Interface 2008*, pp. 139–146.

[Lloyd06] Brandon Lloyd, David Tuft, Sung-Eui Yoon, and Dinesh Manocha. 2006. “Warping and Partitioning for Low Error Shadow Maps.” In *Proceedings of the Eurographics Symposium on Rendering 2006*, pp. 215–226.

[Martin04] Tobias Martin and Tiow-Seng Tan. “Anti-Aliasing and Continuity with Trapezoidal Shadow Maps.” In *Proceedings of the Eurographics Symposium on Rendering 2004*, 2004, pp. 153–160.

[Reeves87] William Reeves, David Salesin, and Robert Cook. “Rendering Antialiased Shadows

with Depth Maps." In *Computer Graphics* (Proceedings of SIGGRAPH 1987), 1987, 21(3), pp. 283–291.

[Salvi07] Marco Salvi. "Rendering Filtered Shadows with Exponential Shadow Maps." *ShaderX*^{6}, edited by Wolfgang Engel. Charles River Media, 2007, pp. 257–274.

[Stamminger02] Marc Stamminger and George Drettakis. "Perspective Shadow Maps." In *Proceedings of SIGGRAPH 2002*, 2002, pp. 557–562.

[Valient07] Michal Valient. "Stable Rendering of Cascaded Shadow Maps," *ShaderX*^{6}, edited by Wolfgang Engel. Charles River Media, 2007, pp. 231–238.

[Williams78] Lance Williams. "Casting Curved Shadows on Curved Surfaces." In *Computer Graphics* (Proceedings of SIGGRAPH 1978) 12(3), 1978, pp. 270–274.

[Wimmer04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. "Light Space Perspective Shadow Maps." In *Proceedings of the Eurographics Symposium on Rendering 2004*, pp. 143–152.

[Zhang06] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. "Parallel-Split Shadow Maps for Largescale Virtual Environments," *In Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications* (VRCIA'2006), 2006, pp. 311–318.

[Zhang07] Fan Zhang, Hanqiu Sun, and Oskari Nyman. "Parallel-Split Shadow Maps on Programmable GPUs." *GPU Gems3*, edited by Hubert Nguyen. Addison Wesley. 2007, pp. 203–237.