Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

Split-Plane Shadow Volumes

Samuli Laine

Helsinki University of Technology / TML
Hybrid Graphics, Ltd.

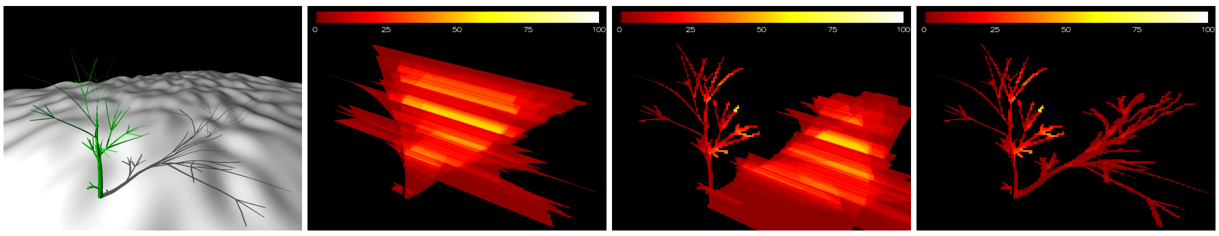


Figure 1: *A graphical comparison of stencil shadow volume algorithms with a complex shadow caster. The colors in the three rightmost figures indicate how many times the pixels are processed. Early culling is used with* 8×8 *pixel tiles. In the setup depicted, the proposed algorithm reduces the number of tiles passing the early culling stage by a factor of 6.5 (6.2) compared to Z-pass (Z-fail) algorithm. The number of pixels processed is reduced by a factor of 8.2 (7.5) and the number of stencil buffer updates by a factor of 30.9 (26.7).*

Abstract

We present a novel method for rendering shadow volumes. The core idea of the method is to locally choose between Z-pass and Z-fail algorithms on a per-tile basis. The choice is made by comparing the contents of the low-resolution depth buffer against an automatically constructed split plane. We show that this reduces the number of stencil updates substantially without affecting the resulting shadows. We outline a simple and efficient hardware implementation that enables the early tile culling stages to reject considerably more pixels than with shadow volume optimizations currently available in the hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Shadowing; I.3.1 [Computer Graphics]: Hardware Architecture – Graphics processors

1. Introduction

Shadows enhance the realism of computer-generated images and also provide information about the spatial relationships of objects. The two predominant techniques for rendering hard shadows are shadow maps [Wil78] and shadow volumes [Cro77]. Shadow maps are efficient, but they suffer from aliasing artifacts caused by their discrete resolution. In contrast, shadow volumes always generate correct shadows but are usually less efficient, primarily because of their fill rate consumption.

In this paper, we present a method for reducing the fill rate requirements of the shadow volume algorithm. We combine two well-known algorithms for rendering the shadow volumes, the so-called Z-pass and Z-fail algorithms. Z-pass algorithm is efficient when the majority of the shadow volume is hidden, while the opposite holds for Z-fail algorithm. The core observation behind our method is that since both algorithms produce identical shadows, the choice between using Z-pass or Z-fail can be made *locally*, as long as it stays consistent while rendering a single shadow volume. If the choice is made on a per-tile basis, we can cull entire pixel tiles when it can be concluded that none of the pixels in the tile would cause stencil updates.

The rest of the paper is organized as follows. We first review previous work on shadow volumes in Section 2. In Section 3, we show that the choice between Z-pass and Z-fail algorithms can be made efficiently by assigning a suitable *split depth* for each surface point and performing a *split test* for choosing the algorithm. In Section 4, we consider computing the split depths based on *split planes* constructed for the shadow volumes. Two methods for constructing the split planes are presented, followed by a robust method for performing the split tests without explicitly computing the split depth. Section 5 outlines a simple hardware extension for performing the split tests on a per-tile basis, enabling the hardware to cull multiple pixels at once. In Section 6, we discuss decomposing the shadow casters into sub-objects, which benefits our method significantly. Experimental results are presented in Section 7, followed by discussion and directions for future work in Section 8.

2. Previous Work

In this section, we focus on real-time methods for rendering shadow volumes using graphics hardware. The topic of shadow rendering in general is broad, and we refer the interested reader to the comprehensive surveys by Woo et al. [WPF90] and Haines and Akenine-Möller [HM01].

The shadow volume algorithm [Cro77] constructs the three-dimensional volumes that represent the shadowed regions, and tests whether the visible surfaces are inside these volumes or not. In the hardware-accelerated implementation of the original algorithm [Hei91], the scene is first rendered from camera with ambient light only, obtaining screen-space depths of all visible surfaces. Then, shadow volumes of the shadow casters are rasterized so that the stencil buffer is updated in the pixels where a fragment of the shadow volume boundary would be visible to the camera. Finally, a third rendering pass is made where fragments are lit with diffuse and specular light only in the pixels where stencil buffer indicates that the visible surface is not in shadow. This is commonly known as the *Z-pass* algorithm, since the shadow volume boundaries must be visible to the camera, i.e. pass the depth test in the usual sense, to cause stencil buffer update.

The Z-pass algorithm fails to calculate the shadows correctly when the near plane of the camera is partially or completely inside the shadow volume. In this case, portions of the shadow volume boundary are clipped away by the near plane of the camera. Recently, Hornus et al. [HHLH05] presented an elegant method called *ZP+* for fixing this defect in the Z-pass algorithm. The idea is to pre-fill the stencil buffer using the shadow caster triangles that lie inside the pyramid formed by the light source and the near plane of the camera. The triangles are projected onto the near plane of the camera with a carefully constructed projection matrix.

Another robust method for rendering shadow volumes is the so-called *Z-fail* algorithm that was independently discovered in slightly different forms by Bilodeau and Songy in 1999 and by Carmack in 2000 [EK02]. In Z-fail algorithm, the stencil buffer is updated for shadow volume boundary fragments that are *behind* the visible surfaces. This removes the problem of near plane clipping, but produces false results when the shadow volume is clipped by the far plane. Far plane clipping can be eliminated by pushing the far plane distance to infinity, or by clamping the depth values instead of clipping [EK02].

2.1. Shadow Volume Optimizations

A shadow volume is, by definition, a closed polyhedral volume that consists of *light cap*, *dark cap* and *side quads*. The light cap is formed by the triangles of a shadow caster that are back-facing to the light source, and the side quads are extruded from the silhouette edges of the shadow caster. The dark cap closes the shadow volume and is commonly a replica of the light cap at the extrusion distance.

The Z-pass algorithm does not need to rasterize the light cap, since the shadow-casting object obscures the light cap by coinciding with it. In addition, the dark cap does not need to be rasterized if the shadow volume is extruded far enough to make the dark cap hidden. Because of this, the Z-pass algorithm is often favorable over Z-fail algorithm, and hence the ZP+ algorithm [HHLH05] that enables the Z-pass algorithm to be used in all situations may provide speedup over Z-fail. Everitt and Kilgard [EK02] suggest selecting dynamically between Z-pass and Z-fail algorithms, using Z-fail only when necessary.

Reducing Rasterization Work Attempts have been made to reduce the number of pixels processed while rendering the shadow volumes. Lengyel [Len02] shows that the scissor test can be used for limiting the rasterization area when an attenuated light source is used. The *depth bounds* hardware extension [NVI03] is similar to scissor test but the test is made against the depth values stored in the depth buffer. When depth bounds db_{min} and db_{max} are set, the hardware culls the pixels where the depth value stored in the depth buffer is outside range $[db_{min}, db_{max}]$. Therefore, if the depth bounds are set so that they contain the entire shadow volume, processing the pixels that are outside the depth bounds can be avoided. Tight depth bounds are obtained when the shadow volume is nearly perpendicular to the viewing direction, but in many cases the depth bounds become very conservative. When an attenuated light source is used, the depth bounds can be clamped according to the attenuation range.

Combining scissor test [Len02] and depth bounds [NVI03] is suggested by McGuire et al. [MHE*03], who also give an algorithm for constructing a simple dark cap based on the silhouette edges of a shadow caster. A method for constructing the shadow volume entirely on graphics hardware is given by Brabec and Seidel [BS03]. Lengyel [Len05] considers a number of stencil shadow optimizations as well as rendering soft shadows. By applying *geometry scissors*, the shadow volume rendering work can be significantly reduced in common cases by taking into account the geometry of the shadow receivers. Interestingly, in the context of penumbra wedge rendering, Lengyel applies a method that is similar to the one presented in this paper. However, the method is used only for penumbra wedge rendering, and does not enable the hardware to perform early pixel tile culling as efficiently as the hardware extensions proposed in this paper.

Lloyd et al. [LWGM04] cull shadow volumes that are themselves in shadow or do not contribute to the final image. They also geometrically clamp the shadow volumes coarsely to regions that contain shadow receivers. Chan and Durand [CD04] use a shadow map [Wil78] for identifying the pixels in the image that lie near shadow discontinuities and then process these pixels using shadow volumes. Artifacts may occur when the resolution of the shadow map is not high enough to capture all the relevant features of the shadow casters. Unlike in the method of Lloyd et al. [LWGM04], the shadow volumes are sent to the graphics hardware in entirety, and the reduction in pixel processing is obtained by exploiting early culling hardware. McCool [McC00] first renders a shadow map with all shadow casters, and then constructs a shadow volume based on the shadow map. Unlike geometry-based shadow volume construction, the side quads are generated only at the outermost silhouettes of the shadow casters, potentially simplifying the shadow volume geometry substantially. However, because of the limited resolution of the shadow map, artifact-free result cannot be guaranteed.

Aila and Akenine-Möller [AAM04] propose a two-stage hierarchical method for rendering shadow volumes in hardware. In the first stage, a low-resolution shadow volume is rasterized and pixel tiles that may contain shadow volume boundaries are detected. In the second stage, the shadow volume is rasterized on pixel level only inside the boundary tiles. They also introduce a hierarchical stencil buffer for reducing the memory bandwidth requirements of stencil buffer updates.

3. Theory

The Z-pass algorithm counts the number of shadow volume enter/exit events along a ray from eye to a fragment. This is achieved by rendering the shadow volume boundary triangles so that whenever the depth value z_{frag} of the boundary triangle fragment at a pixel is smaller than the depth z_{pixel} of the pixel stored in depth buffer, the stencil buffer is updated. The adjustment is positive for front-facing boundary triangles (enter event), and negative for back-facing boundary triangles (exit event). Formally, we can express this with the following equation for stencil buffer adjustment ΔS:

$$\Delta S_{(\mathrm{ZP})} = \begin{cases} +1 & \text{if} \quad z_{frag} < z_{pixel}\ ,\ facing = front \\ -1 & \text{if} \quad z_{frag} < z_{pixel}\ ,\ facing = back \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The Z-fail algorithm counts the number of events along a ray from infinite distance to a fragment. The criterion for determining whether the stencil buffer should be updated or not is inverted, as is the sign of the adjustment based on the facing of the triangle. Thus, the equation for stencil buffer adjustment ΔS is as follows:

$$\Delta S_{(\mathrm{ZF})} = \begin{cases} -1 & \text{if} \quad z_{frag} \geq z_{pixel}\ ,\ facing = front \\ +1 & \text{if} \quad z_{frag} \geq z_{pixel}\ ,\ facing = back \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

If depth bounds are used, both Equations 1 and 2 should be augmented with a criterion that ΔS is zero whenever z_{pixel} is outside range $[db_{min}, db_{max}]$.

3.1. Split Test

The key idea of our method is to apply one more criterion for determining the stencil buffer adjustment ΔS. We take into accout a pixel-dependent *split depth*, denoted z_{split}, and compare it against the depth stored in the depth buffer, denoted z_{pixel} . We refer to this test as the *split test*, and the result of the test determines whether Z-pass or Z-fail stencil update rules are to be used. The split depth does not need to be same for every pixel (unlike depth bounds db_{min} and db_{max} are), but we require that z_{split} remains same for any given pixel as long as we are processing a single shadow volume. This ensures that the result of the split test is consistent while processing a single shadow volume.

If z_{pixel} is smaller than z_{split}, we apply Z-pass stencil update rules, and otherwise we apply Z-fail rules. We thus have the following equation for stencil buffer adjustment ΔS:

$$\Delta S_{(\mathrm{SPSV})} = \begin{cases} +1 & \text{if} \quad z_{frag} < z_{pixel} < z_{split}\ ,\ facing = front \\ -1 & \text{if} \quad z_{frag} < z_{pixel} < z_{split}\ ,\ facing = back \\ -1 & \text{if} \quad z_{frag} \geq z_{pixel} \geq z_{split}\ ,\ facing = front \\ +1 & \text{if} \quad z_{frag} \geq z_{pixel} \geq z_{split}\ ,\ facing = back \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

From now on, we refer to this set of rules as the *split-plane shadow volume algorithm* (SPSV), for reasons that will become apparent later in this paper. Since it is required that z_{split} remains same for any given pixel while rendering a single shadow volume, the stencil update rules stay consistent for a given fragment. Therefore, we effectively perform a per-pixel choice between Z-pass and Z-fail algorithms, and since both methods provide correct shadows, so does the proposed method.

The motivation for adding the split test is that no stencil buffer update is required unless z_{pixel} is between z_{frag} and z_{split}, as can be seen from Equation 3. This generally yields a reduction in the number of stencil buffer updates, provided that the split depth z_{split} is chosen wisely. We show in Section 4 that good split depths can be obtained efficiently by assigning a suitable *split plane* for every individual shadow volume, and calculating the split depths based on the plane equation of the split plane.

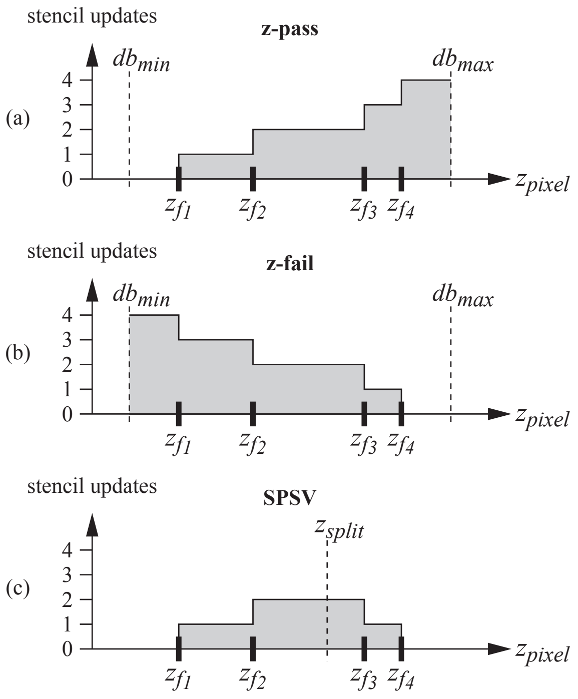


Figure 2: *Comparing the number of stencil updates. Let us consider the cost of processing a single pixel with unknown* z_{pixel}*. Depth bounds are indicated by* db_{min} *and* db_{max}*. Four shadow volume boundary fragments* $f_1 \cdots f_4$ *with depths* $z_{f_1} \cdots z_{f_4}$ *are processed at the pixel. The graphs show the number of stencil updates as a function of* z_{pixel}*. (a) Z-pass algorithm updates the stencil for every shadow volume boundary fragment with* $z_f < z_{pixel}$*, except for* $z_{pixel} > db_{max}$*, where no updates are made. (b) Z-fail algorithm updates the stencil for every shadow volume boundary fragment with* $z_f \geq z_{pixel}$*, except for* $z_{pixel} < db_{min}$*, where no updates are made. (c) The split-plane shadow volume algorithm updates the stencil for every shadow volume boundary fragment where* z_{pixel} *is between* z_{split} *and* z_f*, i.e.* $z_f < z_{pixel} < z_{split}$ *or* $z_f \geq z_{pixel} \geq z_{split}$*. Using depth bounds gives additional benefit only if* z_{split} *is outside* db_{min} *and* db_{max}*. The shaded area under the curve is proportional to the expected number of stencil updates. The optimal split position* z_{split} *is such that there are equally many boundary triangles on both sides of* z_{split}*, since this minimizes both the area under the curve and the worst-case update count. As long as* z_{split} *is between* db_{min} *and* db_{max}*, the area under the curve is always smaller than with Z-pass or Z-fail.*

3.2. Comparison of Different Algorithms

In Figure 2, we compare Z-pass and Z-fail algorithms against the proposed split-plane algorithm in the context of processing a single fragment with unknown depth. With Z-pass (Figure 2a) and Z-fail (Figure 2b) algorithms, the number of stencil buffer updates is in worst case equal to the number of shadow volume boundary triangles that overlap the fragment. In contrast, with an appropriate split distance z_{split}, the split-plane shadow volume algorithm (Figure 2c) never performs more than two updates in the case depicted. Note that the depth bounds would provide no additional benefit with the split-plane shadow volume algorithm.

What is especially important is that the split-plane algorithm reduces the number of stencil updates to zero whenever pixel depth z_{pixel} is outside range $[min\{z_{frag_1}, \ldots, z_{frag_N}, z_{split}\}, max\{z_{frag_1}, \ldots, z_{frag_N}, z_{split}\}]$. Therefore, when the shadow volume lies entirely behind or in front of visible geometry, no stencil updates are performed if the split depth is chosen suitably. Using depth bounds db_{min} and db_{max} may achieve the same result, but since the depth bounds are set globally, they are generally much more conservative. Therefore, the worst case of z_{pixel} being inside depth bounds but outside the shadow volume is quite common.

3.3. Shadow Rendering Algorithm

The basic Z-pass algorithm has well-known problems when the near plane of the camera is located inside the shadow volume. These problems concern our algorithm as well in pixels where Z-pass rules are used. As mentioned in Section 2, the recently presented ZP+ algorithm [HHLH05] corrects these problems with Z-pass algorithm. In our method, the ZP+ correction must be applied exactly in the pixels where Z-pass rules are used. In pixels that use Z-fail rules, the correction must not be applied, since incorrect results would be obtained from mixing the two algorithms.

The Z-fail algorithm needs the caps of the shadow volume to be drawn, whereas the Z-pass algorithm never needs to draw the light cap. The dark cap is required by Z-pass algorithm only when the extrusion distance of the shadow volume is not enough to guarantee that the dark cap is not in sight. In our algorithm, we can thus always limit the rendering of the light cap to those pixels that use Z-fail rules. For dark cap, the same applies depending on the extrusion distance.

We can thus formulate the full shadow rendering algorithm for a single shadow volume as follows:

1. render ZP+ correction geometry in Z-pass pixels
2. render light cap in Z-fail pixels
3. if dark cap may be visible, render dark cap in all pixels, otherwise render dark cap only in Z-fail pixels
4. render side quads with split-plane shadow volume algorithm (Equation 3)

All the steps above can be performed in a straightforward fashion if the graphics API allows selecting the stencil update operation based on the results of the depth test, split test and polygon facing.

4. Split Plane

In this section, we consider computing the split depths based on automatically constructed split planes. We first present two heuristic methods for determining split planes for a given object. The first method spans the plane according to a single pre-selected point associated with the object, the light position and the camera position. The second method is targeted for thin objects and it spans the plane according to the light position and two pre-selected points associated with the object. After this, we present two methods for performing the actual split test against the split plane, followed by a short discussion on the robustness of the split test and plane construction methods.

4.1. Plane Construction Method 1: Point-Camera-Light

The split plane should split the shadow volume efficiently in two halves, as seen from the camera. We can thus formulate our goal as maximizing the screen-space area of the portion of the split plane that lies inside the shadow volume. Considering that the shape of the object is not taken into account, we pursue this goal by positioning the plane so that it contains the approximated centerline of the shadow volume and is maximally orthogonal to the view rays from camera to the approximated centerline. The centerline of the shadow volume is approximated by selecting a single point $\mathbf{p}$ inside the object, and spanning a line through light position $\mathbf{l}$ and $\mathbf{p}$. In our tests, placing $\mathbf{p}$ simply at the center of the bounding box of the object worked well. The orthogonality requirement is fulfilled by orienting the plane so that the split plane is orthogonal to the plane formed by camera position $\mathbf{c}$, light position $\mathbf{l}$ and the selected point $\mathbf{p}$.

We write the homogeneous plane equation $Ax+By+Cz+Dw=0$ as a dot product between two four-vectors $\mathbf{P}$ and $\mathbf{x}$ so that $\mathbf{P}\cdot\mathbf{x}=0$, where $\mathbf{x}$ is the homogeneous point $[x,y,z,w]$ to be classified, and $\mathbf{P}$ contains the plane constants A,B,C and D. For convenience, we further decompose $\mathbf{P}$ into plane normal $\mathbf{P_n}=[\mathbf{P}_x,\mathbf{P}_y,\mathbf{P}_z]=[A,B,C]$ and plane offset $\mathbf{P}_w=D$.

Plane $\mathbf{P}$ can be computed according to the aforementioned constraints subject to points $\mathbf{p}$, $\mathbf{c}$ and $\mathbf{l}$ as follows:

$$\begin{aligned}\mathbf{P_n} &= [(\mathbf{c}-\mathbf{l})\times(\mathbf{p}-\mathbf{l})]\times(\mathbf{p}-\mathbf{l})\\ \mathbf{P}_w &= -\mathbf{P_n}\cdot\mathbf{l}\end{aligned} \tag{4}$$

The facing of the plane, i.e. whether the camera is on the positive or on the negative side of the plane, does not concern us here. Figure 3 shows two examples of split planes constructed with Point-Camera-Light method.

4.2. Plane Construction Method 2: Point-Point-Light

In cases where the shadow-casting object is long and thin, it is not useful to align the split plane so that it is maximally orthogonal with the view rays. The orthogonality requirement may tilt the plane so that the overlap with the shadow volume becomes small. Consider a thin, cylindrical object with

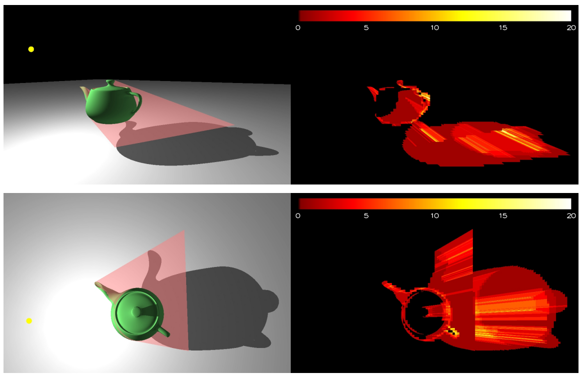


Figure 3: *Split planes constructed with Point-Camera-Light method (Section 4.1). Top row: The split plane (shown in transparent red) constructed with point* **p** *at the center of the bounding box of the teapot, and the resulting pixel processing counts when the shadow volume is rendered with split-plane shadow volume algorithm. Bottom row: A different viewpoint produces a different split plane, because the location of the camera is taken into account. The blocky features in the pixel processing count images on the right are due to per-tile culling (to be explained in Section 5).*

two points $\mathbf{p}_1$ and $\mathbf{p}_2$ placed at the endpoints of the cylinder. Spanning the plane through points $\mathbf{p}_1$, $\mathbf{p}_2$ and light position $\mathbf{l}$ is now a good choice, since it ensures that the split plane is properly aligned inside the flat shadow volume generated by the thin cylinder. Plane $\mathbf{P}$ is therefore constructed as follows:

$$\begin{aligned}\mathbf{P_n} &= (\mathbf{p}_1-\mathbf{l})\times(\mathbf{p}_2-\mathbf{l})\\ \mathbf{P}_w &= -\mathbf{P_n}\cdot\mathbf{l}\end{aligned} \tag{5}$$

Figure 4 shows a comparison of Point-Camera-Light and Point-Point-Light methods with a shadow-casting pole.

4.3. Split Test Against a Split Plane

There are essentially two ways for performing the split test against a split plane. The first approach is to explicitly compute z_{split} and then compare z_{pixel} against it. The second approach is to construct point $\mathbf{x}$ that represents the location of the surface in the pixel, and then evaluate the sign of dot product $\mathbf{P}\cdot\mathbf{x}$. For the sake of completeness, we assume that the comparison operator in the split test can be freely selected by the application. This comparison operator is denoted $\odot$, and the split test is therefore written as $z_{pixel}\odot z_{split}$.

The split test is most conveniently performed in *normalized device coordinate* (NDC) space. This is the space to which points are transformed by multiplying them with modelview and projection matrices and performing the perspective divide by w. In NDC, all coordinates of a point inside the view frustum are between -1 and $+1$. After computing the split plane $\mathbf{P}$ in world space, it can be transformed to NDC space by multiplying $\mathbf{P}$ with the inverse transpose

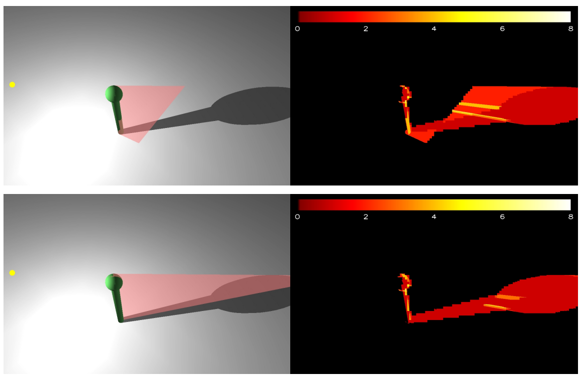


Figure 4: *Difference between Point-Camera-Light (Section 4.1) and Point-Point-Light (Section 4.2) split plane construction methods. Top row: The split plane for the pole is constructed with Point-Camera-Light method, and the resulting pixel processing counts are shown on the right. The split plane is obviously far from optimal. Bottom row: The split plane is constructed with Point-Point-Light method, with points* $\mathbf{p}_1$ *and* $\mathbf{p}_2$ *positioned at the bottom and top of the pole. The long and thin structure of the shadow caster allows this method to perform clearly better.*

of the combined modelview and projection matrix. In the following, we assume that $\mathbf{P}$ has been transformed to NDC space. Furthermore, z_{pixel} is assumed to be in NDC coordinates as well.

We first examine the approach of explicitly computing z_{split} at a given pixel. Computing z_{split} in NDC space requires solving the following equation:

$$\mathbf{P} \cdot [x, y, z_{split}, 1] = 0, \tag{6}$$

giving

$$z_{split} = -(\mathbf{P}_x x + \mathbf{P}_y y + \mathbf{P}_w)/\mathbf{P}_z, \tag{7}$$

where x and y are the NDC x and y coordinates of the pixel being processed. After solving z_{split}, we can perform the comparison operation $z_{pixel} \odot z_{split}$. A nice property of this solution is that, at least in theory, z_{split} can be interpolated across pixels just like ordinary z values. In practice, this may be impossible because of numerical imprecision and the strict requirement that z_{split} stays constant for a given pixel as long as $\mathbf{P}$ remains constant.

Another method for performing the split test is to construct point $\mathbf{x}$ that represents the NDC position of the surface present in the pixel, and then classify it against the split plane $\mathbf{P}$ by performing dot product $\mathbf{P} \cdot \mathbf{x}$. However, we must conditionally flip the sign of the dot product depending on whether the camera is on the positive or on the negative side of the split plane. In NDC space, the camera is conveniently located at $[0, 0, -1, 0]$ and the facing of the plane is therefore defined by the sign of $\mathbf{P}_z$. Hence, the result of the split test $z_{pixel} \odot z_{split}$ can be computed as follows:

$$z_{pixel} \odot z_{split} \equiv sign(\mathbf{P}_z)(\mathbf{P} \cdot [x, y, z_{pixel}, 1]) \odot 0, \tag{8}$$

where x and y are again the NDC coordinates of the pixel being processed. In this approach, no division operation is required.

4.4. Robustness

Both approaches for performing the split test run into a singularity condition when $\mathbf{P}_z = 0$, which occurs when the camera lies exactly on the split plane. Computing z_{split} in Equation 7 results in an attempt to divide by zero. In Equation 8, the situation is degenerate in the sense that the result of the split test depends only on whether the comparison operator $\odot$ includes equality or not. However, no invalid arithmetical operations are performed.

Because the only purpose of the split test is to choose between Z-pass and Z-fail stencil update rules, the result of the split test does not affect the appearance of the shadows. Therefore, as long as the split test produces consistently the same result for a given pixel and a given split plane, the shadow rendering process is inherently robust. Because of the same reason, there is no need to explicitly handle degenerate situations when computing the split planes (Equations 4 and 5).

5. Hardware Implementation

Reducing the number of stencil updates is of no great help in itself, unless we are also able to reduce the number of pixels processed. In addition, the cost of performing the split test for every pixel may well surpass the benefit gained from reducing the number of stencil updates. In this section, we discuss an efficient hardware implementation that makes it possible to cull multiple pixels at once and thereby reduce the number of pixels processed.

5.1. Per-Tile Depth Test

Modern GPUs have the ability to cull multiple pixels at once by performing a tile-based depth test. This is accomplished by maintaining a low-resolution depth buffer where aggregate depth information is stored for groups of pixels. In the following, we assume that the pixels are grouped in tiles, and for each tile, z_{min} [AMS03] and z_{max} [Mor00] values are maintained. These values contain the minimum and maximum depth values of the pixels in the tile, respectively. The low-resolution depth buffer is typically stored on-chip for fast access, whereas the per-pixel depth buffer resides in video memory and is accessed through a tile cache.

The per-tile depth test involves computing minimum and maximum depth values of the render primitive inside the area of the tile. This depth interval is compared against the $[z_{min}, z_{max}]$ interval fetched from the low-resolution depth buffer. If the intervals do not overlap, the result of the depth test is the same for all pixels in the tile. In this case, the depth values of the per-pixel depth buffer do not need to be accessed, which saves memory bandwidth. In addition, if the active render state is such that no color, depth or stencil writes can occur for any of the pixels in the tile, the pixels can be safely culled altogether [Mor00].

5.2. Per-Tile Split Test

We propose performing the split test in the per-tile stage as well. The result of the per-tile split test can be used for all pixels in a tile, eliminating the need to perform the split test for every pixel separately. Remembering that the result of the split test only chooses between Z-fail and Z-pass stencil update rules, it is perfectly safe to perform this selection on a per-tile basis.

In the tile-based split test, we compare the split depth z_{split} at the center of the tile against z_{min} or z_{max} value stored in the low-resolution depth buffer. The choice between comparing z_{split} against z_{min} or z_{max} is rather arbitrary and only affects the choice between Z-pass and Z-fail in the cases where z_{split} is in interval $[z_{min}, z_{max}]$. In our software implementation, we compare z_{split} against z_{max}, and thus favor Z-fail over Z-pass in these situations. The alternative choice of comparing z_{split} against z_{min} produced similar results in the benchmarks. Computing z_{split} explicitly for the comparison can be avoided by performing the split test according to Equation 8. No additional memory bandwidth is needed, since the z_{min} or z_{max} value required for performing the split test is needed anyway for the tile-based depth test. We also notice that the split test can be executed in parallel with the tile-based depth test.

Combining the results of tile-based depth test and tile-based split test gives us the possibility to cull entire tiles. If it can be concluded that no stencil updates are to be made according to Equation 3, the tile can be rejected. In the pixel processing stages, the split test must not be performed for each pixel individually. Instead, the result of the corresponding per-tile split test must be used for choosing the stencil update rules.

6. Shadow Caster Decomposition

In this section, we show that decomposing the shadow casting objects into smaller sub-objects and assigning separate split planes for each sub-object can further reduce the pixel processing work in the split-plane shadow volume algorithm. Shadow caster decomposition is also used by Lloyd et al. [LWGM04] for better spatial resolution in the culling of shadow volumes.

We assume that the shadow caster is a closed, non-self-intersecting mesh, or composed of a number of such meshes. Bergeron [Ber86] shows that the original shadow volume algorithm can be extended to handle non-closed meshes. The solution requires that both ± 1 and ± 2 stencil updates can be performed, depending on the number of triangles a silhouette edge is connected to. Currently, only ± 1 stencil updates are supported in hardware, making this method inapplicable in practice.

We observe that if a mesh is *originally closed*, it can be decomposed into sub-objects that are non-closed, and the shadow volumes of these sub-objects can be rendered separately with ± 1 stencil updates. This is achieved simply by ignoring the silhouette edges that are connected to a single triangle that is front-facing to the light source. With this method, the shadow volume of each sub-object becomes closed. More side quads are rendered when processing the shadow volumes, since all edges whose neighboring triangles belong to different sub-objects become silhouette edges.

By decomposing the shadow casters into sub-objects, we are able to assign each sub-object a separate split plane. This is illustrated in Figure 5, where a complex shadow caster is decomposed into four parts. The tree of Figure 1 was decomposed into 49 sub-objects, and the split plane for each branch was constructed with Point-Point-Light method (Section 4.2). The significance of performing the decomposition is examined in Section 7.

7. Experimental Results

We compared the split-plane shadow volume algorithm against Z-pass and Z-fail algorithms with depth bounds optimization. The benchmarks were performed with a software rasterizer equipped with per-tile depth and split tests. We measured the number of tiles processed, the number of tiles that survived the per-tile culling stage, the number of pixels processed in those tiles, and the number of actual stencil updates performed.

The benchmarks were measured by rendering animation sequences of four test scenes. In all scenes, the light source and the viewpoint are moving, and in KNIGHT and 16KNIGHTS scenes, the objects are also moving. The simplest KNIGHT scene features a single shadow-casting object with 634 triangles. In 16KNIGHTS scene there are 16 similar objects, giving a total of 10144 triangles. The TREE scene contains a single shadow-casting object with 2173 triangles. The TREE-DECOMPOSED scene is exactly the same, but the shadow caster is decomposed into 49 sub-objects so that each branch of the tree is a separate sub-object. In the original tree model, all branches were separately constructed as closed surfaces, and therefore no additional silhouette edges were introduced by the decomposition.

In scenes KNIGHT, 16KNIGHTS and TREE, the split planes were constructed with Point-Camera-Light method (Section 4.1), with **p** placed at the center of the bounding box of each object. In TREE-DECOMPOSED scene, the split plane was computed for each branch with Point-Point-Light

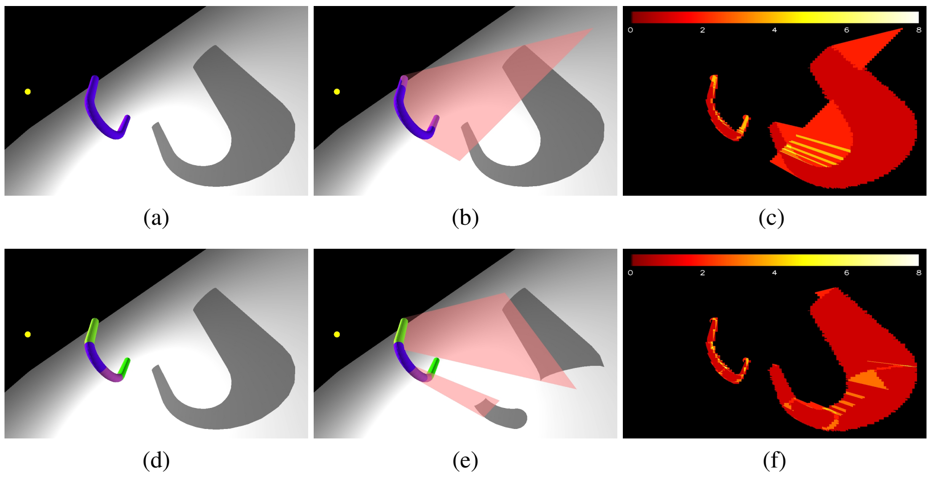


Figure 5: *Shadow caster decomposition. Top row: (a) A shadow caster is presented as a single object. (b) The split plane computed with Point-Camera-Light method (Section 4.1) with* **p** *at the center of the bounding box of the object. (c) The resulting pixel processing counts. Bottom row: (d) The same object is decomposed into four sub-objects for shadow volume rendering. (e) The split planes for the curved parts are computed with Point-Camera-Light method (Section 4.1), and the split planes for the straight parts are computed with Point-Point-Light method (Section 4.2) with* $\mathbf{p_1}$ *and* $\mathbf{p_2}$ *placed at the ends of the cylinders. For clarity, the shadows and split planes are shown for two sub-objects only. (f) The resulting pixel processing counts when all four sub-objects are processed. Compared to the upper row, the number of tiles processed by the low-resolution rasterizer is increased by 42% because of the additional silhouette edges. However, the number of pixels processed is decreased by 36% and the number of stencil updates by 40%. The resolution of all images is* 1024×640 *pixels, and early culling is performed in* 8×8 *pixel tiles.*

method (Section 4.2), with $\mathbf{p}_1$ and $\mathbf{p}_2$ placed approximately at the endpoints of the branch.

In the comparison methods, the depth bounds for an object are initialized with the bounding box of the object. Then, the shadow volume of the bounding box is constructed, and the depth bounds are extended to contain the intersection between this volume and the view frustum. The extrusion distances of the shadow volumes are computed according to the diameter of the scene. The Z-pass comparison method was augmented with the ZP+ correction algorithm [HHLH05] for correct results.

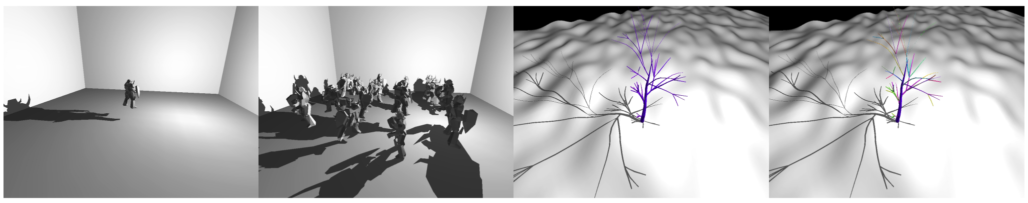
The results are summarized in Table 1. The proposed split-plane shadow volume algorithm outperformed both Z-pass and Z-fail algorithms with depth bounds optimization in all scenes. The number of non-culled tiles was reduced by a factor of 1.83 – 3.78. The number of pixels processed was decreased by a factor of 1.87 – 5.26, and the number of stencil updates by a factor of 2.07 – 14.09. The smallest improvements in all respects were obtained in TREE scene, and the largest in TREE-DECOMPOSED. This highlights the usefulness of shadow caster decomposition. Interestingly, the comparison methods also benefited notably from the decomposition due to tightened depth bounds. The results in scenes KNIGHT, 16KNIGHTS and TREE indicate that even without shadow caster decomposition, the proposed algorithm performs better than the comparison methods.

Figure 6 shows the pixel processing counts in the animation sequences of scenes 16KNIGHTS and TREE-DECOMPOSED. In all animation frames, the proposed method processed fewer pixels than either of the comparison methods. The large variance in the number of pixels processed is caused by rapidly moving light source and viewpoint.

8. Discussion and Future Work

In the closing section of the ZP+ paper [HHLH05], the authors pose the following question: is it possible to efficiently decide (in a given situation) whether Z-pass or Z-fail is more efficient in terms of fill rate? In this paper, we have answered this question by presenting a method for making good choices *locally*.

When compared against the hierarchical shadow volume algorithm by Aila and Akenine-Möller [AAM04], our method does not reduce the fill requirements inside shadowed areas. We conjecture that such an optimization always requires a multi-stage algorithm, which in turn does not fit entirely naturally in the stream processing paradigm of current GPUs. Our split-plane shadow volume algorithm does not require partitioning the shadow volume rendering process into a number of distinct stages, and therefore does not hinder the flow of primitives and fragments inside the GPU in any way.



	KNIGHT			16KNIGHTS			TREE			TREE-DECOMPOSED		
Algorithm	**ZP+DB**	**ZF+DB**	**SPSV**	**ZP+DB**	**ZF+DB**	**SPSV**	**ZP+DB**	**ZF+DB**	**SPSV**	**ZP+DB**	**ZF+DB**	**SPSV**
Tiles processed	75.7	75.7	75.7	997.2	997.2	997.2	243.3	243.3	243.3	243.3	243.3	243.3
Non-culled tiles	22.2	20.8	7.7	326.5	286.1	131.1	128.8	96.6	52.8	71.8	70.8	19.0
Pixels processed	512.3	490.8	170.4	8016.1	6425.5	2738.7	4008.8	2724.6	1454.9	2170.4	1976.9	412.5
Pixels updated	452.7	441.4	129.1	6744.7	5167.6	1966.5	3881.4	2456.3	1187.5	1925.5	1681.0	136.6
Rel. non-culled tiles	2.87	2.70	1.00	2.49	2.18	1.00	2.44	1.83	1.00	3.78	3.73	1.00
Rel. pixels processed	3.00	2.88	1.00	2.93	2.35	1.00	2.76	1.87	1.00	5.26	4.79	1.00
Rel. pixels updated	3.51	3.42	1.00	3.43	2.35	1.00	3.27	2.07	1.00	14.09	12.30	1.00

Table 1: *Comparison between Z-pass with depth bounds* (ZP+DB)*, Z-fail with depth bounds* (ZF+DB) *and split-plane shadow volume* (SPSV) *algorithms. All counts are in thousands. The relative counts in the lower part of the table are normalized so that* SPSV *algorithm has value 1. The results are averages over animation sequences with moving light source and viewpoint. In* KNIGHT *and* 16KNIGHTS *scenes, the objects were also moving. All frames were rendered in* 1024×768 *resolution and per-tile culling was performed in* 8×8 *pixel tiles.*

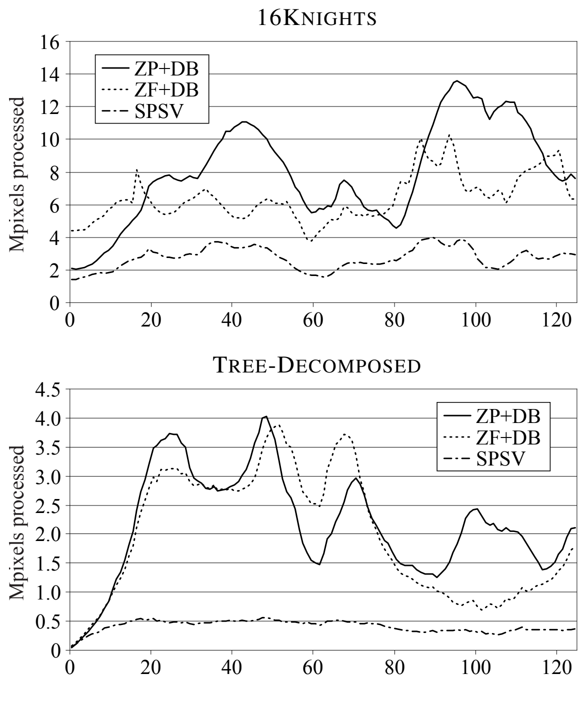


Figure 6: *Plots of per-frame pixel processing counts in the animation sequences of scenes* 16KNIGHTS *and* TREE-DECOMPOSED*. In every frame, the split-plane shadow volume algorithm processed fewer pixels than either of the comparison methods.*

We feel that the hardware requirements for using our method are quite small compared to the reduction in pixel processing work. Assuming that per-tile depth test is already performed by the hardware, the only major additional component is the per-tile split test unit that computes one dot product per pixel tile (Equation 8). In the pixel processing stages, a tiny bit of additional logic is also needed for choosing the stencil operation.

In our method, the shadow volumes are submitted to the graphics hardware in entirety, as in the hybrid shadow volume algorithm of Chan and Durand [CD04]. As with their method, the reduction in the rasterization work comes from enhanced early tile culling. Compared to the Z-pass and Z-fail algorithms with depth bounds, our method is able to cull far more tiles in the early culling stage. This poses additional requirements for the efficiency of the per-tile processing stage, since enough non-culled tiles must be fed to the pixel processing stages to keep them busy.

It should be noted that computing the split planes is actually easier than computing the depth bounds for the shadow volumes. Our method can also be used for accelerating the rendering of penumbra wedges in soft shadow volume algorithms [AAM03].

8.1. Where to Construct the Split Planes?

There are a number of possible places for constructing the split plane. The most obvious choice is to construct the split planes for each shadow caster on the CPU, after which they can be passed to the GPU via render state constants or auxiliary vertex streams. This is the easiest and potentially most efficient approach, especially if the shadow volumes are also constructed on the CPU.

Another option is to construct the split plane in the programmable vertex processing unit of the GPU. This might be the most feasible choice if the shadow volume is also constructed on GPU. The drawback is that the same split plane equation is redundantly computed multiple times. However, if the shadow caster is decomposed into multiple sub-objects, separate split planes may be constructed for each sub-object as shown in Section 6. In this situation, constructing the split planes on the GPU may become feasible, since the amount of redundant computation is decreased.

8.2. Avoiding Render State Changes

In order to enable the hardware to process data in large batches, it is necessary to avoid unnecessary render state changes. This is especially important when the shadow casters are decomposed into multiple sub-objects, yielding a number of different split planes to be used per shadow caster.

We propose making it possible to route the split plane equation to the per-tile split test unit from the vertex shader outputs instead of render state constants. This would enable the split planes to be transferred to the hardware via an auxiliary vertex stream and eliminate the need for state changes when changing the split plane. In addition, constructing the split planes on the GPU would be possible.

8.3. Future Work

In our test scenes, the split plane construction methods for the shadow casters were chosen based on intuition, and with the Point-Point-Light method (Section 4.2), points $\mathbf{p}_1$ and $\mathbf{p}_2$ were placed manually. The tree model in Figure 1 and the shadow caster in Figure 5 were also decomposed into sub-objects by hand. We feel that it should be possible to develop algorithms for performing all of these tasks automatically, most probably yielding in better results than manual pre-processing.

It might be feasible to decompose the shadow casting objects dynamically, depending on the relative positions of the camera, light and the shadow caster. Finally, it seems rather obvious that the two split plane construction methods presented in Section 4 are not the only possible ones, and more sophisticated split plane construction methods could certainly be developed.

Acknowledgements The author would like to thank Timo Aila and the 3Dr team at HUT for comments and discussions. Perilith knight model courtesy of James Green, www.perilith.com/pknight. This work has been partially funded by the National Technology Agency of Finland, Bitboys, Hybrid Graphics, Nokia and Remedy Entertainment.

References

[AAM03] ASSARSSON U., AKENINE-MÖLLER T.: A Geometry-based Soft Shadow Volume Algorithm using Graphics Hardware. *ACM Transactions on Graphics, 22*, 3 (2003), 511–520.

[AAM04] AILA T., AKENINE-MÖLLER T.: A Hierarchical Shadow Volume Algorithm. In *Graphics Hardware* (2004), pp. 15–23.

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics, 22*, 3 (2003), 801–808.

[Ber86] BERGERON P.: A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics and Applications 6*, 9 (1986), 17–28.

[BS03] BRABEC S., SEIDEL H.-P.: Shadow volumes on programmable graphics hardware. In *Proceedings of Eurographics* (2003), vol. 22, pp. 433–440.

[CD04] CHAN E., DURAND F.: An efficient hybrid shadow rendering algorithm. In *Proceedings of the Eurographics Symposium on Rendering* (2004), Eurographics Association, pp. 185–195.

[Cro77] CROW F.: Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 77)* (July 1977), ACM, pp. 242–248.

[EK02] EVERITT C., KILGARD M.: Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. *http://developer.nvidia.com* (2002).

[Hei91] HEIDMANN T.: Real Shadows, Real Time. *Iris Universe*, 18 (November 1991), 28–31.

[HHLH05] HORNUS S., HOBEROCK J., LEFEBVRE S., HART J.: ZP+: correct z-pass stencil shadows. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (2005), ACM Press, pp. 195–202.

[HM01] HAINES E., MÖLLER T.: Real-Time Shadows. In *Proceeding of Game Developers Conference* (March 2001), pp. 335–352.

[Len02] LENGYEL E.: The Mechanics of Robust Stencil Shadows. *http://www.gamasutra.com* (October 2002).

[Len05] LENGYEL E.: Advanced Stencil Shadow and Penumbral Wedge Rendering. *Presentation at Game Developers Conference 2005*, http://www.terathon.com/gdc_lengyel.ppt (2005).

[LWGM04] LLOYD B., WENDT J., GOVINDARAJU N. K., MANOCHA D.: CC Shadow Volumes. In *Proceedings of the Eurographics Symposium on Rendering* (2004).

[McC00] MCCOOL M. D.: Shadow Volume Reconstruction from Depth Maps. *ACM Transactions on Graphics, 19*, 1 (2000), 1–26.

[MHE*03] MCGUIRE M., HUGUES J. F., EGAN K. T., KILGARD M., EVERITT C.: *Fast, Practical and Robust Shadows*. Tech. Rep. CS03-19, Brown University, October 2003.

[Mor00] MOREIN S.: ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings* (August 2000), ACM SIGGRAPH/Eurographics.

[NVI03] NVIDIA: *NVIDIA GeForceFX 5900, 5700 and Go5700 GPUs: UltraShadow Technology*. Tech. rep., http://www.nvidia.com, 2003.

[Wil78] WILLIAMS L.: Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (Proceedings of ACM SIGGRAPH 78)* (August 1978), ACM, pp. 270–274.

[WPF90] WOO A., POULIN P., FOURNIER A.: A Survey of Shadow Algorithms. *IEEE Computer Graphics and Applications 10*, 6 (November 1990), 13–32.