

Efficient Shadow Algorithms on Graphics Hardware

by

Eric Chan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

July 2004

© Eric Chan, MMIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis document
in whole or in part.

Author
Department of Electrical Engineering and Computer Science
July 1, 2004

Certified by
Frédo Durand
Assistant Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Efficient Shadow Algorithms on Graphics Hardware

by

Eric Chan

Submitted to the Department of Electrical Engineering and Computer Science
on July 1, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Shadows are important to computer graphics because they add realism and help the viewer identify spatial relationships. Shadows are also useful story-telling devices. For instance, artists carefully choose the shape, softness, and placement of shadows to establish mood or character. Many shadow generation techniques developed over the years have been used successfully in offline movie production. It is still challenging, however, to compute high-quality shadows in real-time for dynamic scenes.

This thesis presents two efficient shadow algorithms. Although these algorithms are designed to run in real-time on graphics hardware, they are also well-suited to offline rendering systems.

First, we describe a hybrid algorithm for rendering hard shadows accurately and efficiently. Our method combines the strengths of two existing techniques, shadow maps and shadow volumes. We first use a shadow map to identify the pixels in the image that lie near shadow discontinuities. Then, we perform the shadow-volume computation only at these pixels to ensure accurate shadow edges. This approach simultaneously avoids the edge aliasing artifacts of standard shadow maps and avoids the high fillrate consumption of standard shadow volumes. The algorithm relies on a hardware mechanism that we call a *computation mask* for rapidly rejecting non-silhouette pixels during rasterization.

Second, we present a method for the real-time rendering of soft shadows. Our approach builds on the shadow map algorithm by attaching geometric primitives that we call *smoothies* to the objects' silhouettes. The smoothies give rise to fake shadows that appear qualitatively like soft shadows, without the cost of densely sampling an area light source. In particular, the softness of the shadow edges depends on the ratio of distances between the light source, the blockers, and the receivers. The soft shadow edges hide objectionable aliasing artifacts that are noticeable with ordinary shadow maps. Our algorithm computes shadows efficiently in image space and maps well to programmable graphics hardware.

Thesis Supervisor: Frédo Durand

Title: Assistant Professor

Acknowledgments

I would like to thank Frédo Durand for being an insightful and incredibly *kewl* advisor. He's also extremely ... well, French. We'll leave it at that.

I am grateful to Nick Triantos, Mark Kilgard, Cass Everitt, and David Kirk from NVIDIA and Michael Doggett, Evan Hart, and James Percy from ATI for providing early access to hardware, driver assistance, and inevitably higher office temperatures. Xavier Décoret implemented a helpful model-conversion program, and Henrik Wann Jensen wrote Dali, the software rendering system used to generate the ray-traced images. Jan Kautz, Addy Ngan, Sylvain Lefebvre, George Drettakis, Janet Chen, and Bill Mark provided valuable feedback. Special thanks to Smoothie for casting a very large shadow, and to Buddha for casting an even larger shadow. This work is supported by an ASEE National Defense Science and Engineering Graduate fellowship.

Contents

1	Introduction	15
1.1	Shadow Concepts	18
1.2	Real-Time Shadows	24
1.3	Contributions	26
2	Related Work	29
2.1	Hard Shadow Algorithms	29
2.1.1	Shadow Maps	30
2.1.2	Shadow Volumes	35
2.1.3	Hybrid Approaches	40
2.2	Soft Shadow Algorithms	41
2.3	Additional Methods	43
2.4	Summary	43
3	Hybrid Shadow Rendering Using Computation Masks	45
3.1	Algorithm	47
3.2	Implementation Issues	49
3.3	Results	50
3.3.1	Image Quality	52
3.3.2	Performance	55
3.4	Discussion	57
3.4.1	Algorithm Tradeoffs	57
3.4.2	Computation Masks	58

3.4.3	Additional Optimizations	58
3.5	Conclusions and Future Work	59
4	Rendering Fake Soft Shadows Using Smoothies	61
4.1	Overview	62
4.2	Algorithm	64
4.2.1	Computing Smoothie Alpha Values	66
4.3	Implementation	69
4.4	Results	70
4.5	Discussion	77
4.5.1	Comparison to Penumbra Maps	79
4.6	Conclusions	80
5	Conclusions	81
	Bibliography	84

List of Figures

1-1	Hair rendered with and without shadows	16
1-2	Examples of soft shadows	16
1-3	Shadows aid in understanding spatial relationships	16
1-4	Portrait lighting examples	17
1-5	Lighting examples in <i>The Godfather</i>	17
1-6	Examples of using lighting and shadows to establish mood	17
1-7	Point-to-point visibility	18
1-8	Hard shadows from a point light source	19
1-9	Partial occlusion of an area light source	19
1-10	Soft shadows from an area light source	20
1-11	Dependence of shadow softness on the size of the light source	21
1-12	Comparison of shadows due to small and large area light sources	21
1-13	Dependence of shadow softness on distance ratios	22
1-14	Comparison of shadows in different geometric configurations	22
1-15	Photograph of real soft shadows	23
1-16	Non-local shadow calculations	24
1-17	Shadows cast by square vs. triangular light sources	25
2-1	Geometric configuration of the shadow map algorithm	30
2-2	Shadow map example	31
2-3	Illustration of incorrect self-shadowing	32
2-4	Examples of bias-related artifacts in shadow maps	32
2-5	Aliasing of shadow maps	33

2-6	Geometric configuration of shadow volumes	35
2-7	Stencil shadow volume example	36
2-8	Difficult cases for stencil-based shadow volumes	37
2-9	Shadow volume fillrate consumption	38
3-1	Hybrid shadow algorithm overview	46
3-2	Example of very few silhouette pixels	46
3-3	Cg pixel shader for finding shadow silhouettes.	48
3-4	Comparison: a cylinder's shadow is rendered using (a) a 512×512 shadow map and (b) our hybrid algorithm with a 256×256 shadow map. Using a lower-resolution shadow map in our case is acceptable because shadow volumes are responsible for reconstructing the shadow silhouette.	49
3-5	Visualization of mixing shadow maps and shadow volumes	51
3-6	Three test scenes for our hybrid algorithm	52
3-7	Comparison of image quality using three shadow algorithms	53
3-8	Artifacts. These images are crops from the Tree scene, View B. The images in the left three columns were generated using our hybrid algorithm with varying shadow-map resolutions. In the top row, the regions indicated in red and green show missing or incorrect shadows due to undersampling in the shadow map. The corresponding images in the bottom row visualize the reconstruction errors using the same color scheme as in Figures 3-5b and 3-5c. The reference image in the far-right column was obtained using shadow volumes.	54

3-9	Fillrate consumption and overdraw in the Dragon Cage scene. The shadow volume polygons, shaded yellow in (a), cover the entire image and have an overdraw factor of 79; brighter yellow corresponds to higher overdraw. Our hybrid method restricts drawing shadow polygons to the silhouette pixels, shaded green in (b); these pixels cover just 5% of the image. The image on the right (c) illustrates the resulting stencil buffer: black pixels on the floor and walls are in shadow and represent non-zero stencil values.	55
3-10	Shadow volume overdraw and silhouette pixel coverage. The black bars show the percentage of pixels in the image covered by shadow volumes, and the number next to each bar is the overdraw factor (the ratio of shadow-polygon pixels to the total image size). The white, red, and blue bars show the percentage of pixels in the image that are classified as shadow silhouette pixels by our hybrid algorithm; colors correspond to different shadow-map resolutions.	56
3-11	Performance comparison. The vertical bars measure the per-frame time in milliseconds (ms) for the shadow map algorithm (SM), our hybrid algorithm (H), and the shadow volume algorithm (SV). Colored sections of a bar indicate how much time is spent in each part of the algorithm.	56
4-1	Smoothie algorithm overview	62
4-2	Construction of smoothies	63
4-3	Shadow determination using smoothies	64
4-4	Computing smoothie alpha values	67
4-5	Alpha computation	68
4-6	Smoothie corner geometry and shading	69
4-7	Cg vertex and fragment shader code for the smoothie algorithm . . .	71
4-8	Comparison of soft shadow edges	72
4-9	Comparison of smoothies vs. ray tracing	73

4-10 Multiple blockers and receivers 74
4-11 Comparison of overlapping penumbrae 74
4-12 Color images rendered using the smoothie algorithm 76

List of Tables

4.1	Smoothie algorithm performance measurements	75
-----	---	----

Chapter 1

Introduction

Shadows play a key role in computer graphics for several reasons. First, they add realism and believability to computer-generated scenes. High-quality shadows, such as the ones shown in Figures 1-1 and 1-2, are needed in photorealistic image synthesis and in lighting simulations. Second, shadows help viewers understand spatial relationships in 3D scenes [WFG92]. For instance, it is hard to determine from Figure 1-3a whether the dragon is sitting on the floor or hanging in mid-air; shadows make it easy to distinguish the two cases (see Figures 1-3b and 1-3c). Finally, shadows are powerful story-telling devices. Photographers often use shadows to establish a desired mood or character; Figure 1-4 shows how different lighting conditions lead to significant changes in appearance. A famous example is cinematographer Gordon Willis's use of an overhead bounce lighting system in *The Godfather* to place Marlon Brando's eyes in shadow (see Figure 1-5); this technique helped to define Brando's character as the mysterious and powerful head of the Corleone family. Such lighting techniques also apply to computer graphics. Artists at animation studios, for example, carefully choose the shape, softness, and color of shadows to define a character's personality (see Figure 1-6). For these applications, the final appearance of the shadow and its effect on the scene are more important than geometric accuracy.

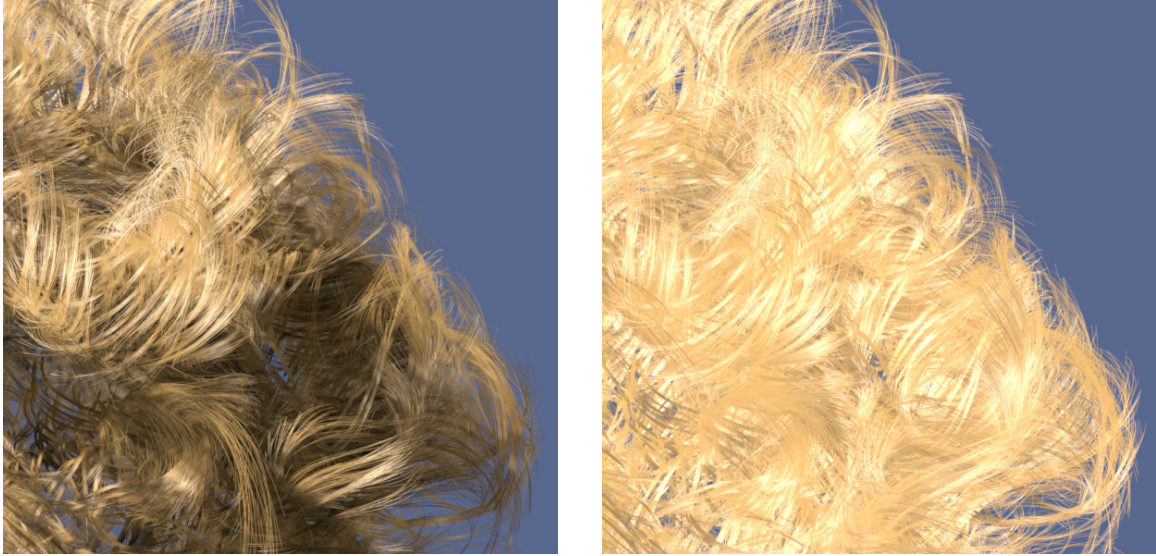


Figure 1-1: Hair rendered with and without shadows. The shadows contribute strongly to the apparent depth and realism of rendered hair. (Images rendered by Tom Lovovic and Eric Veach [LV00].)



Figure 1-2: Examples of soft shadows and their contributions to photorealistic image synthesis. (Images rendered by Henrik Wann Jensen.)

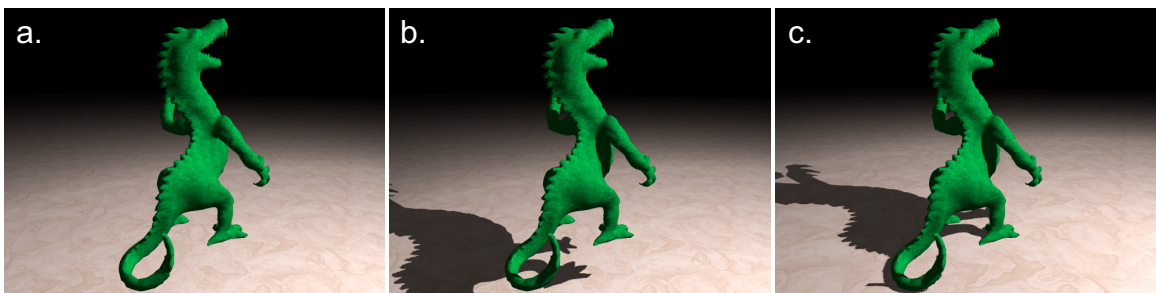


Figure 1-3: Shadows and spatial relationships. (a) Without shadows, it is hard to determine if the dragon is above or on the floor. (b) and (c) Shadows make it easy to distinguish the two cases.



Figure 1-4: Examples of portrait lighting, including (from left to right) direct on-camera flash, direct off-camera flash, flash bounced from above, and flash bounced from the side. The location and softness of the shadows can be used to add depth and emphasize facial features. (Photographs by David Arky.)



Figure 1-5: Lighting in *The Godfather*. Cinematographer Gordon Willis used a bounce lighting system attached to a low ceiling to cast shadows over Marlon Brando's eyes. This approach hides the thoughts of Brando's character from the audience and defines the mysterious and powerful head of the Corleone family. (© Paramount Pictures)

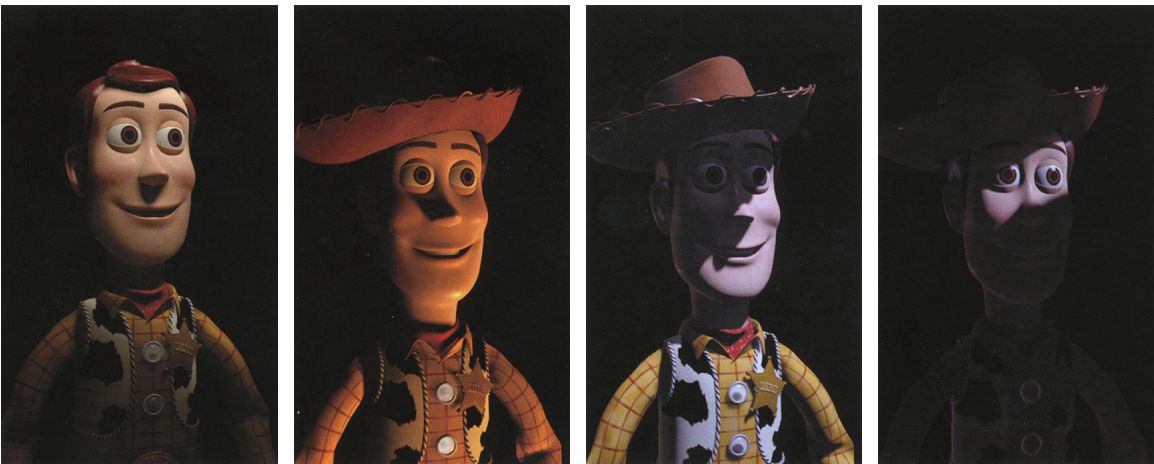


Figure 1-6: Lighting and shadow techniques also apply to computer graphics. Each of these examples shows different shadow qualities, including placement, shape, and softness. (© Disney Enterprises, Inc.)

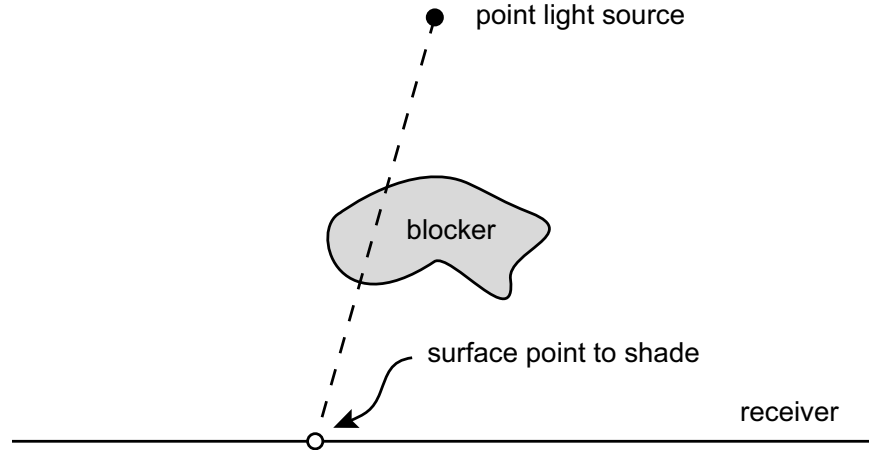


Figure 1-7: Point-to-point visibility in two dimensions. When computing shadows cast by a point light source, we must check for each surface point whether or not the light is visible from that point.

1.1 Shadow Concepts

When rendering images in computer graphics, we need to compute how much light arrives at each visible surface point. Surface points that receive no light are by definition in shadow. Seen another way, the computation of shadows boils down to a visibility problem: to determine whether or not a surface point lies in shadow, we need to know how much of the incoming light is blocked by other objects in the scene. We use the term *blocker* to refer to an object that occludes the incoming light and therefore casts a shadow; similarly, we use the term *receiver* to refer to an object onto which shadows are cast. Note that a concave surface acts both as a blocker and as a receiver; self-shadowing of concave surfaces is an important visual phenomenon, as demonstrated in Figure 1-1.

Let's consider a scene with a single point light source. To compute shadows for this scene, we need to know whether the light is visible from each surface point. Figure 1-7 illustrates this *point-to-point* visibility problem in two dimensions. Since the visibility function is binary, there is a sharp discontinuity between the shadowed and illuminated regions, as shown in Figure 1-8. Thus point light sources lead to hard shadows.

Unlike point light sources, area light sources require us to solve a *point-to-area*

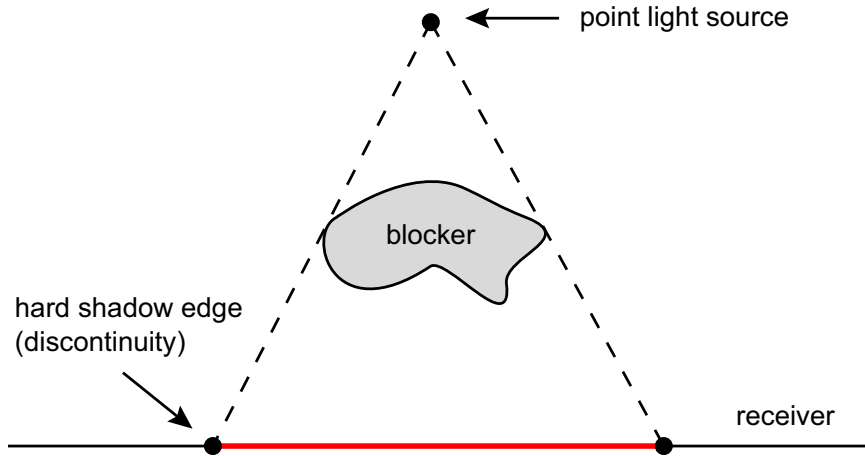


Figure 1-8: Visibility discontinuities. A blocker casts a hard shadow from a point light source onto a receiver. The shadowed region is indicated in red.

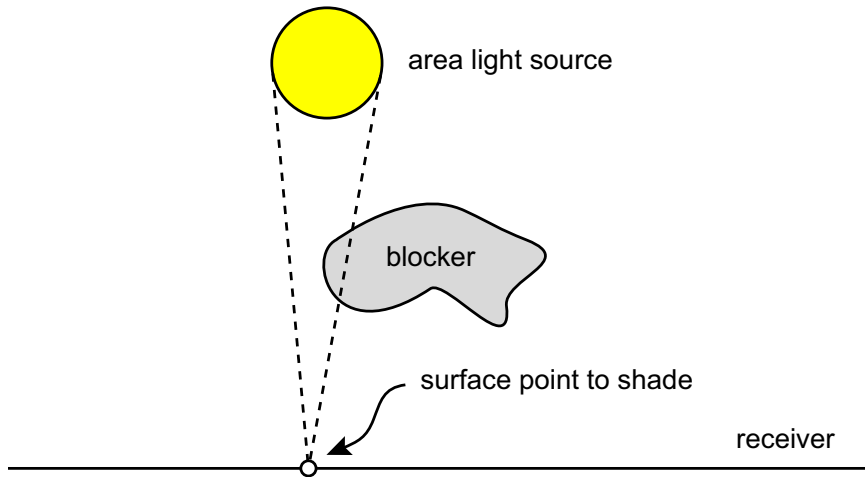


Figure 1-9: Point-to-area visibility in two dimensions. In this example, only part of an area light source is visible from the indicated surface point. This point is partially in shadow.

visibility problem. Figure 1-9 shows an example in which only part of an area light source is visible from the indicated surface point; this point lies partially in shadow. More generally, area light sources give rise to soft shadows because of smooth transitions between complete visibility and complete occlusion of the light source. Figure 1-10 depicts a common geometric configuration and highlights these transitions in blue. The surface region that receives no light is called the *umbra*; the region that sees part of the light source is called the *penumbra*. Note that the visibility function is no longer binary: it varies smoothly within the penumbra between 0 and 1.

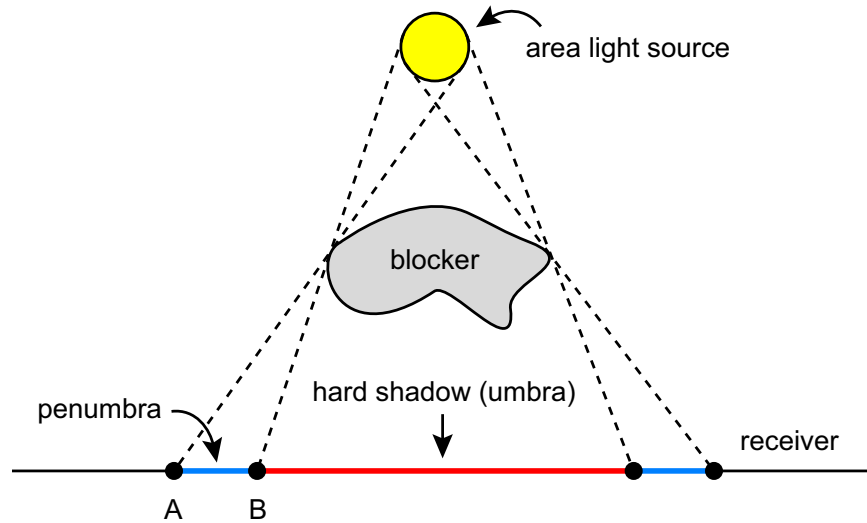


Figure 1-10: An area light source gives rise to soft shadows due to smooth transitions between complete visibility and complete occlusion of the light source. A surface region that receives no light at all is called the *umbra* (highlighted in red). The part of the region that sees part of the light source is called the *penumbra* (highlighted in blue). The visibility function is no longer binary: it varies smoothly from 1 at point A to 0 at point B.

The appearance of soft shadows depends on the geometric relationships between the light source, blocker, and receiver. For instance, Figure 1-11 shows how the shadow’s penumbra varies with the size of an area light source; larger lights lead to larger penumbræ and smaller umbræ. The photographs in Figure 1-12 illustrate this principle.

Shadow softness also depends on the ratio of distances between the area light source, blocker, and receiver (see Figure 1-13). Let b be the distance from the light source to the blocker, and let r be the distance from the light source to the receiver. When the blocker is much closer to the receiver than to the light ($b/r \approx 1$), the area light source behaves like a point light source and the resulting shadows have small penumbræ. In contrast, when the blocker is much closer to the light source than to the receiver (ratio $b/r \approx 0$), the resulting shadows have large penumbræ. This is why the shadow cast by a box sitting on the floor appears sharp near the contact point and becomes softer farther away. See Figures 1-14 and 1-15 for examples of this phenomenon.

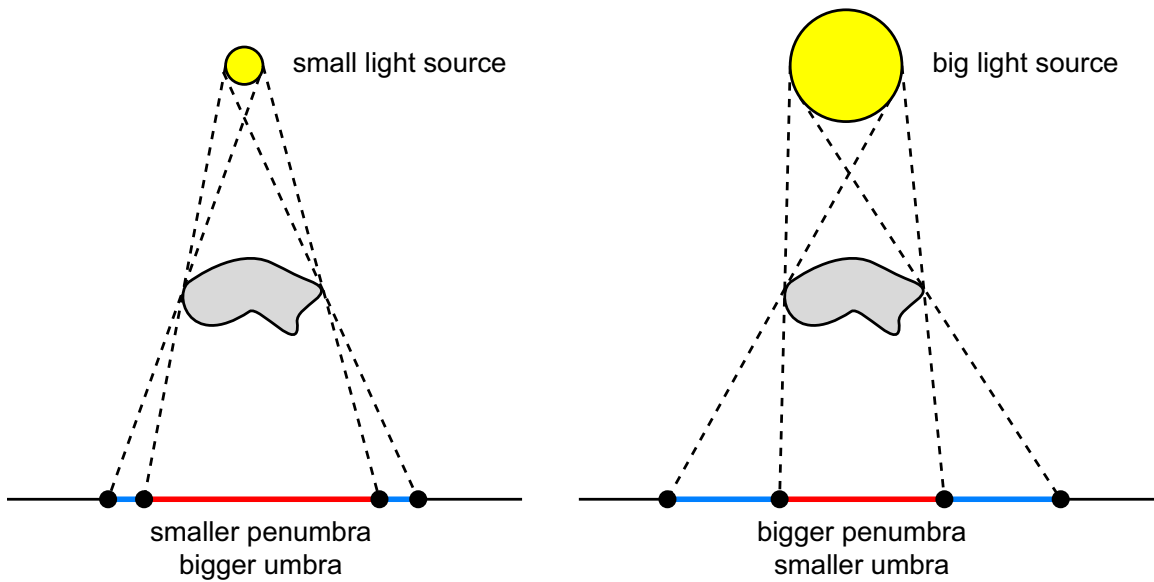


Figure 1-11: Larger area light sources lead to larger penumbræ and smaller umbræ. The two diagrams illustrate this geometric relationship.



Figure 1-12: Photographs of shadows due to small and large area light sources. (Left) The box of pepper is illuminated by a small flashlight, which leads to sharp shadows. (Right) The box is illuminated using a large desk lamp, which leads to softer shadows.

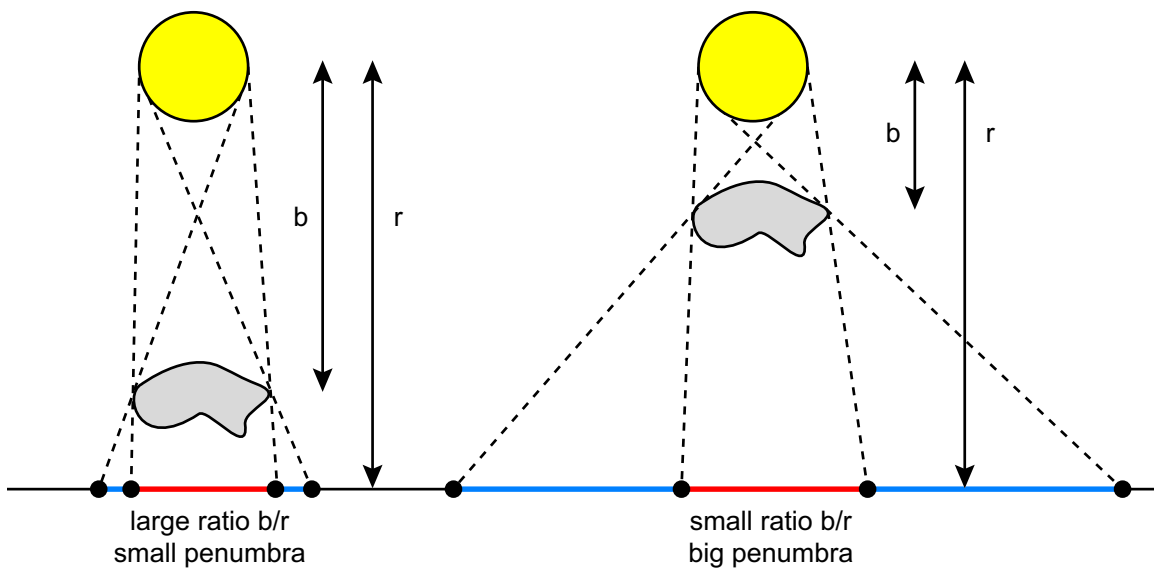


Figure 1-13: Umbrae and penumbrae depend on the ratio of distances between the light source, blocker, and receiver. When the blocker is much closer to the receiver than to the light source (left image: ratio $b/r \approx 1$), the area light source is approximately a point light source; the resulting shadows have small penumbrae. In contrast, when the blocker is closer to the light source than to the receiver (right image: ratio $b/r \approx 0$), the resulting shadows have large penumbrae.



Figure 1-14: Photographs of shadows in different geometric configurations. (Left) The light source (a desk lamp) is placed far away from the box of pepper, which leads to sharp shadows. (Right) The same light source is placed very close to the box, which leads to softer shadows.



Figure 1-15: Shadow softness depends on the ratio of distances between the area light source, blocker, and receiver. In this photograph, the box of pepper is the blocker and the floor is the receiver. Notice how the shadows appear sharp near the contact point between the two surfaces and become softer farther away.

The geometric relationships described above affect the types of shadows we often see from day to day. For instance, on a clear sunny day, the sun — despite its size — acts as a point light source because of its great distance from the earth; the resulting shadows are hard and create high contrast. On the other hand, cloudy days lead to very soft, almost non-existent shadows because the entire sky acts as a giant area light source.

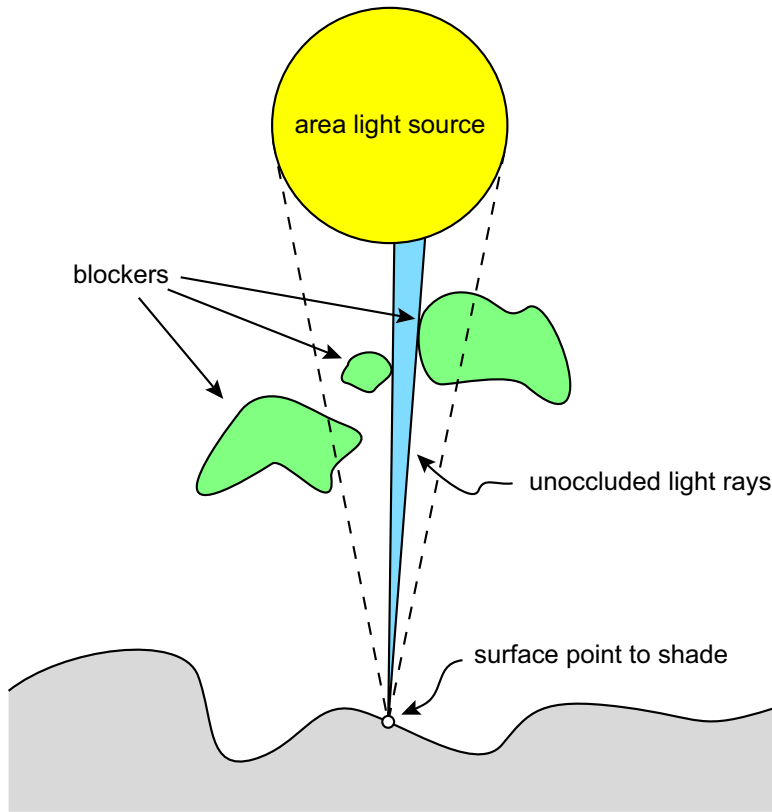


Figure 1-16: Shadow calculations are inherently non-local. In this example, the point we are trying to shade receives only a small amount of light from the large area light source due to occlusions from multiple independent blockers. Computing this quantity of light accurately, efficiently, and robustly is far from trivial.

1.2 Real-Time Shadows

Although shadows are conceptually easy to understand, they are difficult to compute efficiently in a rendering system. To see why, let's consider the nature of the required calculations. Unlike surface reflectance, which depends only on local properties such as the normal vector, shadow calculations are inherently non-local. To determine whether or not a point on a surface lies in shadow, we must check if other (possibly far away) surfaces block the incident light. Soft shadows are particularly difficult to compute because multiple blockers may contribute to partial occlusion of an area light source. A 2D example is shown in Figure 1-16.

Designing a real-time shadow algorithm is challenging because of several conflicting goals. Let's discuss five of the most important ones.



Figure 1-17: Photographs of shadows cast by square vs. triangular light sources. (Left) The box of pepper is lit by a square light source. (Right) The box is lit by a triangular light source. While the shadow penumbrae differ in shape, these differences are rather small.

Image quality. Synthesized images should be free of aliasing artifacts and other “digital signatures.” When rendering soft shadows, geometric accuracy is desirable but sometimes unnecessary. As we discussed in the previous section, the thickness of the penumbra depends greatly on the size of the light source and the ratio of distances between the light, blocker, and receiver; thus a soft shadow algorithm should attempt to capture this phenomenon. In contrast, the exact *shape* of an area light source is less important. Consider, for example, the shadow cast by a square light source versus the shadow cast by a triangular light source (see Figure 1-17). The differences in the shape of the penumbrae are relatively insignificant, particularly in animated scenes.

Robustness. Developers that work with offline rendering systems for movie production have the liberty of knowing their viewpoints and scene configuration in advance. If necessary, they can manually tweak algorithm parameters on a per-scene or per-frame basis. Unfortunately, this is not the case for interactive dynamic applications, because the scene configuration and viewing parameters may change unpredictably from frame to frame. Therefore, it is important for real-time shadow algorithms to be robust to such changes.

Efficiency. A shadow algorithm intended to be used in real-time applications must be simple enough to implement using graphics hardware. This requirement

places a number of constraints on the design of the algorithm. For instance, current graphics architectures lack an efficient *scatter* operation, which is used to write data to an arbitrarily-computed memory address. It is also important to realize that an algorithm that maps well to the hardware may not scale well if it places unreasonable demands on hardware resources; as we will see in the next chapter, this is a major issue with the shadow volume method. Scalability, rather than efficiency, is key to making a shadow algorithm practical in complex, heavily-shadowed scenes.

Generality. A shadow algorithm should obey the “anything-can-shadow-anything” principle, meaning that the algorithm should treat all surfaces in the scene as potential blockers and receivers. For example, an algorithm should handle self-shadowing for concave surfaces. In addition, a shadow algorithm should support blockers and receivers regardless of their model representation. As we will see, some algorithms are flexible and support any primitive that can be rasterized into a depth buffer, whereas others require watertight polygonal models.

Support for dynamic scenes. Interactive applications such as 3D games are moving towards fully dynamic environments. The viewpoint, light sources, characters, and the environment itself may change from frame to frame. Shadow algorithms that require extensive scene pre-processing are too expensive for dynamic scenes.

Not surprisingly, no known shadow algorithm has all of these desirable properties! Designing a practical shadow algorithm involves finding a reasonable tradeoff among these goals.

1.3 Contributions

This thesis describes two real-time algorithms, one for rendering hard shadows and one for rendering soft shadows. We will refer to our hard shadow algorithm as the *hybrid* algorithm and refer to our soft shadow algorithm as the *smoothie* algorithm.

In Chapter 3, we present a hybrid algorithm for rendering hard shadows accurately and efficiently. Our method combines the strengths of two existing techniques, shadow maps and shadow volumes. We first use a shadow map to identify the pix-

els in the image that lie near shadow discontinuities. Then, we perform the shadow volume algorithm only at these pixels to ensure accurate shadow edges. This approach simultaneously avoids the edge aliasing artifacts of standard shadow maps and avoids the high fillrate consumption of standard shadow volumes. The algorithm relies on a hardware mechanism that we call a *computation mask* for rapidly rejecting non-silhouette pixels during rasterization.

In Chapter 4, we present a method for the real-time rendering of soft shadows. Our approach builds on the shadow map algorithm by attaching geometric primitives that we call *smoothies* to the objects' silhouettes. The smoothies give rise to fake shadows that appear qualitatively like soft shadows, without the cost of densely sampling an area light source. In particular, the softness of the shadow edges depends on the ratio of distances between the light source, the blockers, and the receivers. The soft shadow edges hide objectionable aliasing artifacts that are noticeable with ordinary shadow maps. Our algorithm computes shadows efficiently in image space and maps well to programmable graphics hardware.

We designed these algorithms while keeping in mind the criteria presented in Section 1.2. The algorithms attempt to balance the competing quality and performance requirements using a mix of image-space and object-space approaches. Both algorithms minimize aliasing artifacts, and the smoothie algorithm generates plausible (though not geometrically-correct) penumbrae. Furthermore, both techniques are designed to support dynamic environments and to scale well to large scenes. These advantages come at a cost, however. Since parts of the algorithms require image-space calculations using a discrete buffer, potential undersampling artifacts prevent us from guaranteeing robustness. Similarly, parts of the algorithms perform geometric calculations in object space, which limits the class of 3D models that we can handle. In summary, we have attempted to maximize image quality and scalability at the cost of some generality. Note that our algorithms are designed to run in real-time on graphics hardware, but they are also applicable to offline rendering systems.

Chapter 2

Related Work

As we discussed in the previous chapter, accurate shadows are hard to compute efficiently and robustly. Several algorithms have been proposed over the past three decades; the two methods proposed in this thesis build on many of these techniques. We focus our discussion below on the most relevant real-time methods and refer the reader to Woo et al.’s paper [WPF90] for a broader survey of shadow algorithms. Akenine-Möller and Haines’s book [AMH02] also provides a good overview of real-time shadow algorithms.

We first discuss algorithms that compute hard shadows from point light sources, then show how these algorithms have been extended to generate soft shadows. Furthermore, we will describe how our methods relate to these existing techniques.

2.1 Hard Shadow Algorithms

Ray casting is a natural way to compute hard shadows. For each surface point, send a *shadow ray* towards the light source and check if the ray intersects a surface before reaching the light. Although intersection tests can be optimized using hardware acceleration [PBMH02, Pur04] and various spatial data structures, these optimizations do not support dynamic scenes efficiently. Furthermore, as we discussed in the previous chapter, the visibility calculations are non-local: they require global access to the scene database, which is hard to achieve using graphics hardware. For these reasons,

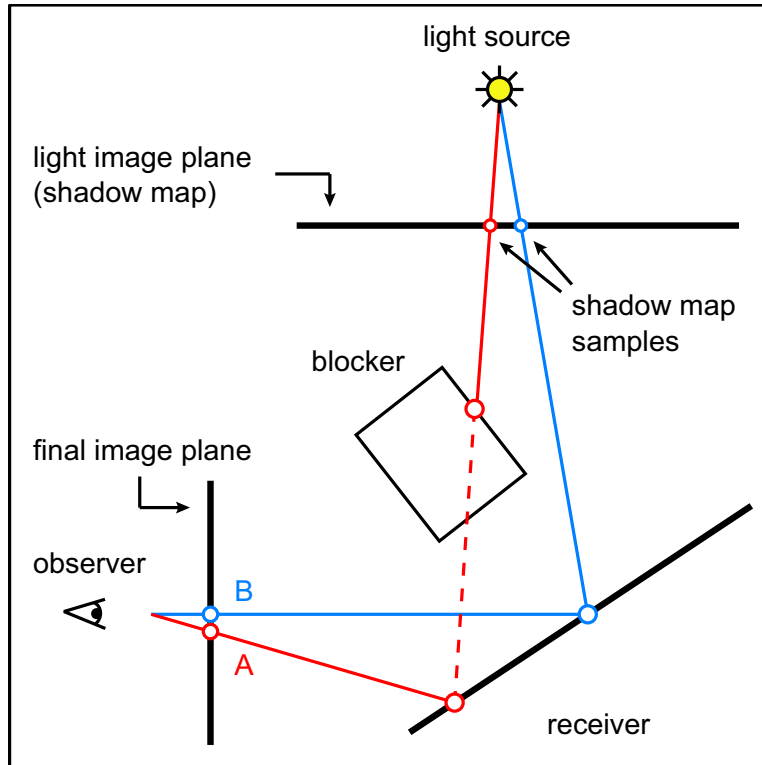


Figure 2-1: Geometric configuration of the shadow map algorithm. Consider the two samples labeled A and B. Sample A is considered in shadow because the point on the receiving plane is farther from the light source than the point stored in the shadow map (which belongs to the blocker). In contrast, sample B is illuminated.

ray casting is fine for offline rendering but is too expensive for real-time applications.

Instead of using ray casting to compute shadows, developers of real-time applications often use *shadow maps* or *shadow volumes*, both of which were originally proposed in the late 1970's.

2.1.1 Shadow Maps

Shadow maps were introduced by Williams in 1978 [Wil78]. The algorithm, outlined in Figure 2-2, involves two rendering passes. In the first pass, we render a depth map of the scene from the light's viewpoint; this depth map is called the *shadow map*. In the second pass, we use the depth map to determine which samples in the final image are visible to the light. To accomplish this, we project each image sample into light space. If the sample's depth in light space is greater than the value stored in

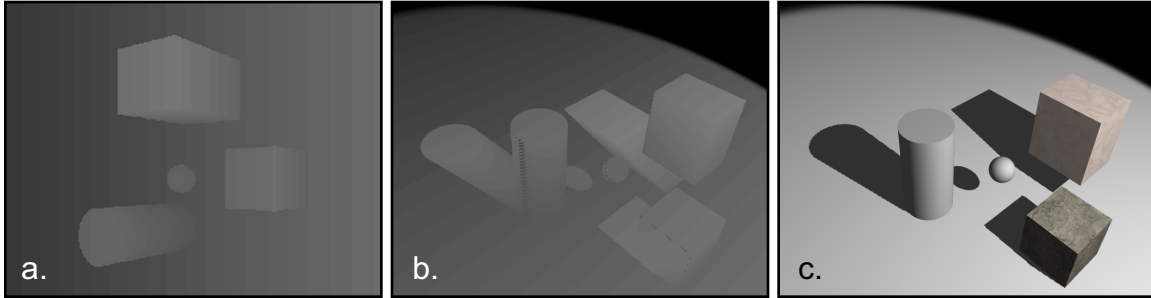


Figure 2-2: Visualization of the shadow map algorithm. (a) Depth map seen from the light’s viewpoint. (b) Depth map projected onto the surfaces, seen from the observer’s viewpoint. (c) Final shadow computed using depth comparisons against the shadow map.

the depth map, then the sample is considered to be in shadow. Figure 2-1 illustrates this process in two dimensions. Consider, for instance, the two samples (labeled A and B) that lie on the observer’s image plane. Sample A is in shadow because the corresponding point on the receiver is farther away from the light source than the blocker’s point in the shadow map. In contrast, sample B is illuminated.

Shadow maps have several nice properties. They map easily to graphics hardware because they use only projective texture mapping [SKvW+92] and simple depth comparisons. In addition, they are efficient because shadow calculations are performed in image space and involve simple texture lookups. Finally, they naturally decouple shadow computations from the scene geometry and support any geometric primitive that can be rasterized into a depth buffer. A limitation of shadow maps, however, is that the algorithm assumes that the light source is a directional light, such as a spotlight. Omnidirectional light sources require additional rendering passes to cover the entire sphere of directions.

The most serious drawback of shadow maps is that depth values are represented in a discrete buffer. This leads to undersampling such as incorrect self-shadowing. To make this easier to understand, consider the 2D example shown in Figure 2-3. During shadow-map generation, surface depth values z_A and z_B are stored at discrete sample positions A and B (shown in red). When rendering the final image, a sample in the observer’s space is transformed into light space at position C (shown in blue), which lies between A and B . The associated depth value z_C is greater than the nearest

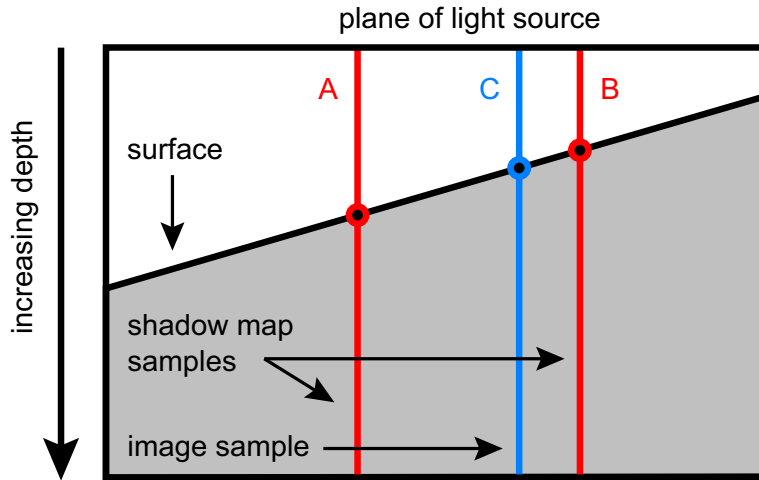


Figure 2-3: Illustration of incorrect self-shadowing in two dimensions. After being transformed into the light’s coordinate system, the image sample C lies in between the two nearest shadow map samples, A and B . Since the depth value at C is greater than the depth value of the nearest shadow map sample B , the image sample is determined incorrectly to be in shadow.

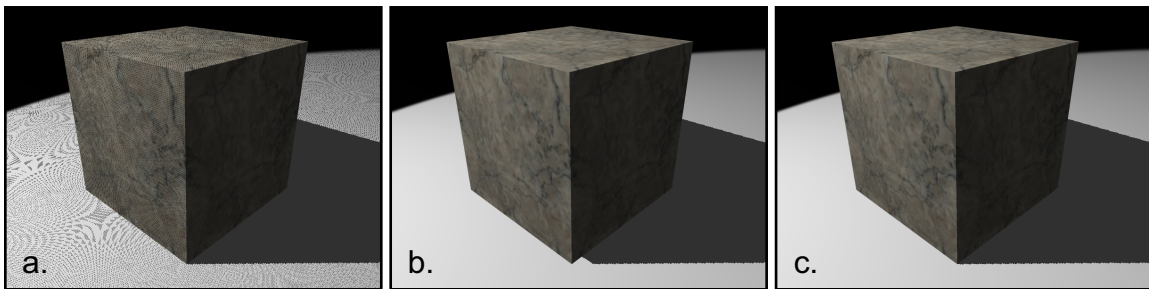


Figure 2-4: Examples of bias-related artifacts in shadow maps. (a) Too little bias is applied, leading to incorrect self-shadowing. (b) Too much bias is applied, causing the shadow boundary to move away from the contact point between the box and the floor. (c) The bias has been manually fine-tuned to avoid self-shadowing artifacts and to obtain the full shadow along the box-floor boundary.

stored depth z_B , so the current sample lies (incorrectly) in shadow. Notice that the difference between z_C and z_B depends on both the slope of the surface relative to the light source and the spacing between adjacent shadow map samples.

A common technique for avoiding self-shadowing artifacts is to apply a bias to the depth value of the current sample. This has the effect of pushing forward the depth values that might otherwise lie just behind a visible surface. Figure 2-4 shows, however, that choosing the appropriate amount of bias can be tricky. Too little bias

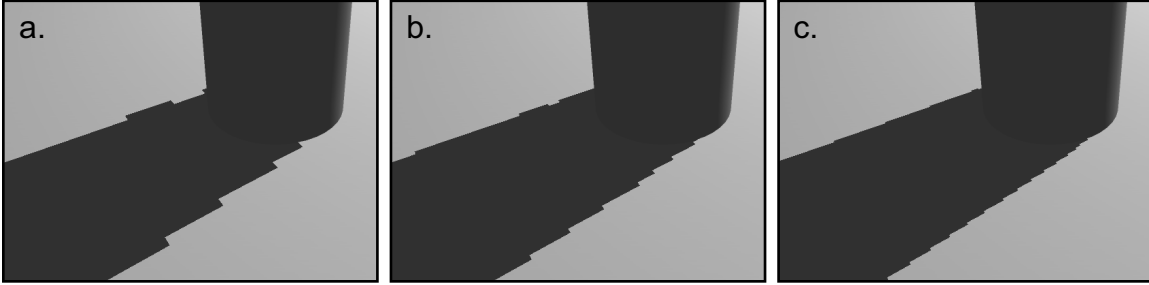


Figure 2-5: Aliasing of shadow maps. This example shows the shadow of a cylinder cast onto a plane. Figures (a), (b), and (c) use 256×256 , 512×512 , and 1024×1024 shadow maps, respectively. Aliasing is evident in all three images.

(Figure 2-4a) leads to clearly-visible self-shadowing artifacts; too much bias (Figure 2-4b) removes parts of the shadow, leading to incorrect depth cues.

Kilgard and many others have noted that the amount of bias needed for a given sample depends on many factors, including the local depth slope and the projected shadow-map pixel area [Kil01]. Determining the appropriate bias is particularly troublesome for interactive applications that render dynamic environments. This is because the projected shadow map pixel area for each sample depends on the viewing parameters and scene configuration, which may change from frame to frame. In contrast, developers that work with offline rendering systems for movie production have the liberty of knowing their viewpoints and knowing the scene configuration in advance. They can also adjust the bias manually on a per-scene or per-frame basis if necessary.

Another consequence of undersampled shadow maps is aliasing. Figure 2-5 shows an example of aliasing in the shadow cast by a cylinder onto a plane. The three images were computed using shadow-map resolutions of 256×256 , 512×512 , and 1024×1024 ; aliasing along the shadow edges is clearly visible in all three images. Although aliasing can be reduced by increasing the resolution of the shadow map, in practice shadow maps are limited in size because of memory constraints. Furthermore, the aliasing problem of shadow maps is harder to address than the usual aliasing problem that arises in image synthesis, because the shadow map algorithm projects the depth map from the light's viewpoint onto the scene; the projected depth map is then

resampled onto the observer’s image grid. Depending on the scene configuration, the aliasing artifacts that result from this projection and resampling process can become arbitrarily severe. For example, the aliasing artifacts in Figure 2-5c can be magnified by moving the observer closer to the cylinder and zooming in on the shadow edges.

Researchers have developed many techniques for addressing shadow-map aliasing. Approaches based on filtering and stochastic sampling [LV00, RSC87] produce nice antialiased shadows. Unfortunately, effective filtering requires a large number of samples per pixel, which is expensive for real-time applications. Furthermore, using a large filter width aggravates incorrect self-shadowing artifacts. In contrast, our smoothie algorithm performs shadow edge antialiasing by projecting filtered texture maps onto the scene; self-shadowing artifacts with this approach are no worse than with ordinary shadow maps.

Other methods reduce aliasing by increasing the effective shadow map resolution. Adaptive shadow maps [FFBG01] detect and redraw undersampled regions of the shadow map at higher resolutions. Unfortunately, the required data structures and host-based calculations preclude real-time performance for dynamic scenes. Perspective shadow maps [SD02, Koz04] are simpler: they just require an additional perspective transformation that effectively provides more resolution in the shadow map to samples located close to the viewer. This method is simple and fast, but it does not reduce aliasing in all cases. For instance, when the light source and the viewer face each other, the perspective transforms mutually cancel and the result is a standard uniform shadow map.

Sen et al. [SCH03] observed that shadow-map aliasing is only problematic near the shadow silhouettes, i.e. discontinuities between shadowed and lit regions. They propose the silhouette map, a 2D data structure that provides a piecewise-linear approximation to the true geometric silhouette seen from the point of view of the light source. The silhouette map provides an excellent reconstruction of the shadow silhouette and eliminates shadow map aliasing in many cases. Since only one silhouette point may be stored per texel in the silhouette map, however, artifacts may appear when multiple shadow boundaries meet. Our hybrid shadow algorithm uses shadow

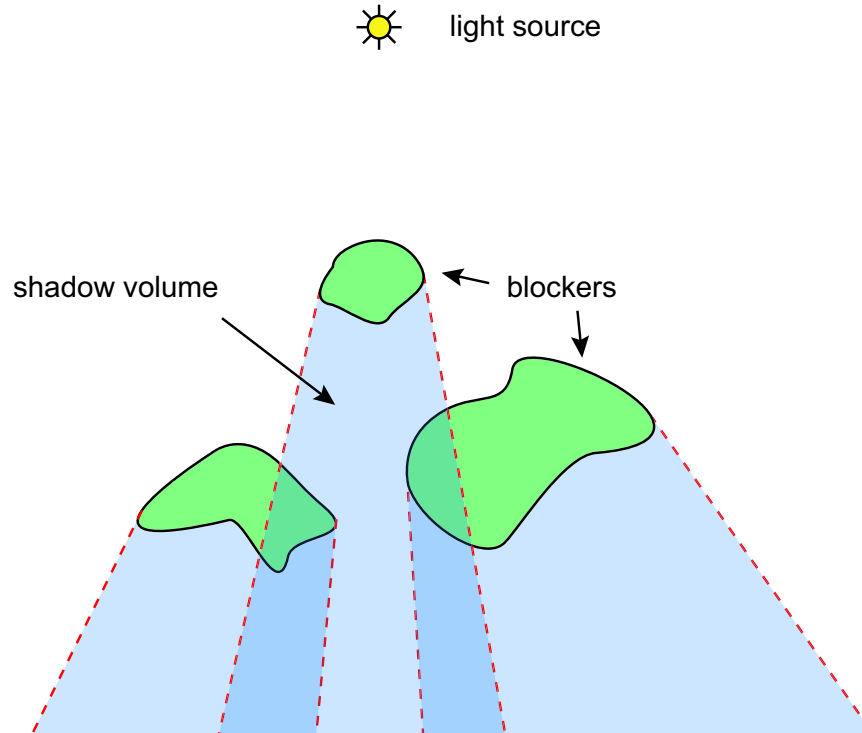


Figure 2-6: Shadow volume configuration in two dimensions. Line segments (dotted red) are extended from the blockers' silhouettes away from the light source. The enclosed region (shaded blue) is the shadow volume.

volumes for edge reconstruction and thereby avoids these artifacts.

2.1.2 Shadow Volumes

Unlike the shadow map, which relies on a discrete shadow representation, Crow's shadow volume algorithm [Cro77] works in object space by drawing polygons to represent the boundary between illuminated and shadowed regions of 3D space. The *shadow volume* itself is the volume bounded by these polygons (see Figure 2-6), and a shadow query involves checking if a point in the image lies within the volume. Bergeron [Ber86] generalized the method to handle open models and non-planar surfaces.

One way to perform shadow queries is to use ray casting, as shown in Figure 2-7. To determine whether a surface point lies within the shadow volume, we use a simple counting argument. For each image ray, we begin with a count of zero. Each time the ray intersects a shadow polygon that is front-facing with respect to the observer,

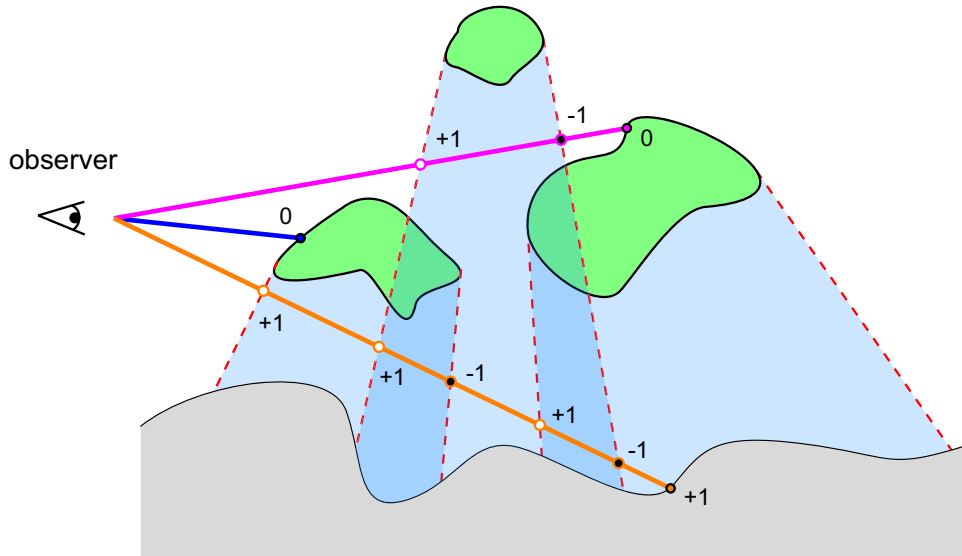


Figure 2-7: Stencil shadow volume example in two dimensions. Consider the three image rays, shown in magenta, blue, and orange. The samples corresponding to the top and middle rays are illuminated because the total stencil count along each ray is 0. In contrast, the sample corresponding to the bottom ray is in shadow because the final stencil count is +1.

we increment the count. Similarly, each time the ray intersects a back-facing shadow polygon, we decrement the count. If the ray intersects an object and the count is non-zero, the intersection point lies in shadow; otherwise the point is illuminated. As an example, Figure 2-7 shows three image rays: the top and middle rays intersect surface points that are illuminated, and the bottom ray intersects a point that lies in shadow.

Heidmann [Hei91] realized that this counting approach could be accelerated using a hardware stencil buffer. Unfortunately, the original algorithm for stencil-based shadow volumes suffered from numerous robustness issues. Figure 2-8 shows examples of tricky cases, such as when the observer is in shadow or when a shadow polygon is clipped by the view frustum's near plane. Some of these issues were addressed by Diefenbach [Die96] and Carmack [Car00]. Recently, Everitt and Kilgard [EK02]

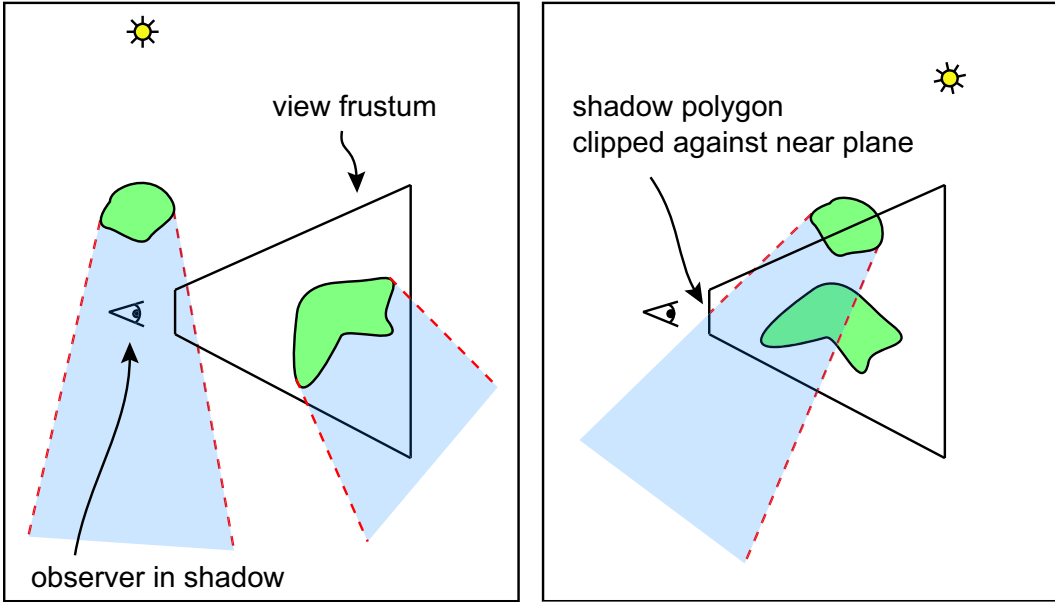


Figure 2-8: Tricky cases for stencil-based shadow volumes. In the left case, the observer is in shadow, so the outcome of stencil counts must be inverted. In the right case, a shadow polygon is clipped by the frustum’s near plane, which leads to inverted shadow calculations in some regions of the image.

described a robust implementation using modern graphics hardware.

Robust stencil shadow volumes offer two advantages over shadow maps. The algorithm is accurate because it computes shadows in object space for every pixel of the final image. Thus, the resulting shadows do not suffer from undersampling artifacts such as aliasing and incorrect self-shadowing. Also, unlike shadow maps, shadow volumes naturally handle omnidirectional light sources.

On the other hand, shadow volumes do not provide the same level of generality as shadow maps. Whereas shadow maps support any type of geometry that can be rasterized into a depth buffer, a robust implementation of shadow volumes places numerous restrictions on the geometry representation. In particular, surfaces must be polygonal models that are oriented and watertight.

The most serious drawback to shadow volumes is that the algorithm does not scale well to scenes with high shadow complexity. The method involves drawing extra geometry, but the main problem is the large *fillrate* required. There are two reasons for this. First, shadow volume polygons occupy substantial screen area, as

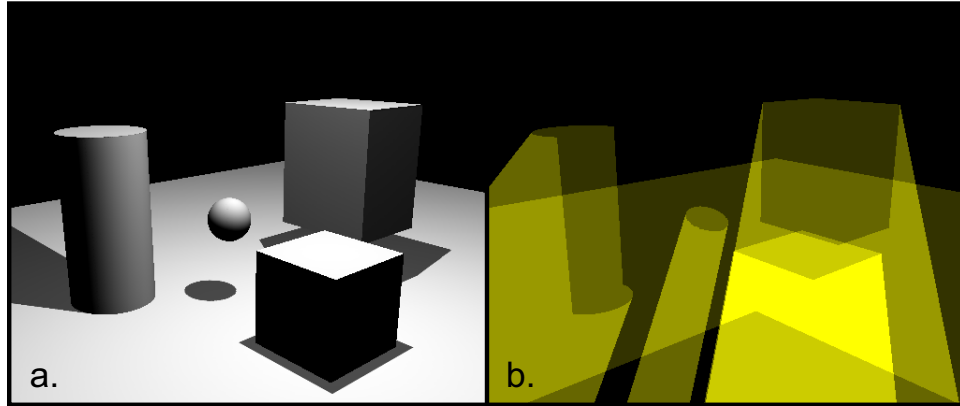


Figure 2-9: A simple scene with a few objects (a) can lead to high fillrate consumption when using shadow volumes. (b) The shadow polygons (shown in yellow) occupy substantial screen area and overlap in screen space. Lighter shades of yellow correspond to regions with higher overdraw.

shown in Figure 2-9, especially in heavily-shadowed scenes or when the observer is in shadow. Second, the shadow polygons overlap in screen space, and every rasterized pixel of every shadow polygon potentially updates the stencil buffer. This degree of overlap, combined with the large screen coverage of shadow polygons, grows rapidly as shadow complexity increases, leading to an explosion in fillrate consumption.

Lengyel [Len02], Everitt and Kilgard [EK03], and McGuire et al. [MHE⁺03] describe a number of culling techniques for optimizing stencil shadow volumes to reduce fillrate. One method estimates the shadow extent on a per-blocker basis and uses scissoring to discard shadow polygon pixels that lie outside the computed scissor rectangle. A related method also considers the depth range of the shadows cast by individual blockers and discards pixels from shadow volume polygons that lie outside of this range. The key idea here is to discard pixels early in the pipeline without performing stencil updates, thereby accelerating rasterization and saving valuable memory bandwidth.

The above techniques are useful, but they are less effective for heavily-shadowed scenes. This is because pixels that lie in shadow are precisely those that lie within the depth ranges and scissor rectangles computed in the above optimizations; thus such pixels do not benefit from these optimizations. Scenes in which shadow polygons have large depth ranges are also problematic. In contrast, our hybrid method performs

rasterization and stencil updates only for pixels that lie near shadow silhouettes. Thus even for scenes with many shadows, the fillrate consumed by shadow volume polygons remains small. Note that our hybrid method is complementary to the aforementioned optimizations.

Researchers have recently proposed several new methods for tackling the fillrate problem of shadow volumes. Aila and Akenine-Möller [AAMar] describe a two-level hierarchical shadow volume algorithm. Their approach is similar to ours in that they identify shadow-boundary pixels and rasterize shadow volumes accurately only at those pixels. There are two important differences, however. First, their method detects tiles of boundary pixels in *object space* by checking for triangle intersections against the tiles' 3D bounding boxes, whereas our method identifies the boundaries in *image space* using a shadow map. Second, an efficient implementation of their method requires numerous changes to hardware, including a modified rasterizer, logic for managing tile updates, and the addition of a delay stream [AMN03]. In contrast, our method relies on existing culling hardware to reduce shadow-volume fillrate.

Lloyd et al. [LWGMar] take a different approach to reducing shadow-volume rasterization costs. One of their techniques is to use image-space occlusion queries to identify blockers that lie entirely in shadow; shadow-volume polygons for such blockers are redundant and may be culled. A similar method is used to cull blockers that cast shadows only on receivers lying outside the observer's view frustum.

Furthermore, Lloyd et al. limit the screen-space extent of shadow-volume polygons in two ways. In the first method, they compute overlaps of the blockers' axis-aligned bounding boxes to estimate the depth intervals from the light source that contain shadow receivers; they clamp shadow polygons to non-empty depth intervals. In the second method, they partition the observer's view frustum into discrete slices and use occlusion queries to determine which slices contain receivers; as in the first approach, shadow polygons are clamped to non-empty slices. The second approach provides greater culling but incurs additional overhead. The key difference between all of the above culling strategies and our method is that Lloyd et al. reduce fillrate by reducing the number and size of the shadow-volume polygons, whereas we draw the

entire polygons and rely on the hardware to perform culling of non-silhouette pixels. These approaches are fully complementary.

2.1.3 Hybrid Approaches

McCool [McC00] was the first to propose a hybrid algorithm that combines shadow maps and shadow volumes. His method first renders a depth map and runs an edge detection algorithm to find the blockers' silhouette edges. Next, the method reconstructs shadow volumes from these edges and uses them to compute shadows in the final image. A strength of McCool's approach is that shadow volume polygons are generated only for silhouette edges that are visible to the light source. Unfortunately, an expensive depth buffer readback is required, shadow polygons are fully rasterized, and artifacts can occur due to aliasing in the shadow volume reconstruction.

Govindaraju et al. [GLY⁺03] propose a different hybrid shadow algorithm. First, they use level-of-detail and PVS culling techniques to reduce the number of potential blockers and receivers; these techniques are implemented using hardware occlusion queries [Ope02b]. Next, they compute shadows using a mix of object-space clipping and shadow maps. Exact clipping of receiver polygons against the blockers' shadow frusta is performed for receivers that would otherwise exhibit shadow-map aliasing artifacts. To identify these receivers, they use a formula derived by Stamminger and Drettakis [SD02] that relates the size of a pixel in shadow-map space to its size when projected into the observer's image space. Shadow maps are used for the remaining receiver polygons. This hybrid approach improves the accuracy of shadow silhouettes without requiring an excessive number of clipping operations.

The approach of Govindaraju et al. is similar to ours in that both methods limit the amount of computation required to render accurate shadow silhouettes. The two methods perform culling at different stages, however. Whereas their method minimizes the number of *objects* that are processed by their clipping algorithm, our method minimizes the number of *pixels* in the image that are treated by shadow volumes. The two methods could be combined by replacing their software-based polygon clipping with our optimized, hardware-accelerated shadow volume rasterization.

2.2 Soft Shadow Algorithms

Both shadow maps and shadow volumes have been extended to support soft shadows.

Shadow maps can be used to render both antialiased and soft shadows through a combination of filtering, stochastic sampling, and image warping techniques [RSC87, LV00, ARHM00, HH97, GCHHar]. However, these methods require examining many samples per pixel to minimize noise and banding artifacts. Dense sampling is costly, even on modern graphics hardware which supports multiple texture accesses per pixel. Consequently, these techniques are mostly used today in high-quality offline rendering systems.

To avoid the cost of dense sampling, some methods use convolution to ensure a continuous variation of light intensity at the shadow edges. For example, Soler and Sillion [SS98] showed how to approximate soft shadows using projective texture mapping and convolution in image space; they convolve the image of the blockers with the inverse image of the light source. Unfortunately, their method cannot easily handle self-shadowing.

Researchers have developed simpler approximations that take advantage of shadow-mapping hardware. For instance, Heidrich et al. [HBS00] described how shadow maps can support linear light sources using only two samples per light. Brabec and Seidel [BS02] extended this idea to handle area light sources by searching over regions of the shadow map. Although their method uses only one depth sample per pixel, it requires a search procedure and an expensive readback from the hardware depth buffer to the host processor. Thus their method is practical only for low-resolution shadow maps.

Parker et al. [PSS98] proposed creating an “outer surface” around each object in the context of a ray tracer. Their technique can be seen as convolution in object space: a sample point whose shadow ray intersects the volume between a real surface and its outer surface is considered partially in shadow. The sample’s light intensity is determined by interpolating alpha values from 0 to 1 between the two surfaces. Only one shadow ray is needed to provide visually smooth results, but because of the way outer surfaces are constructed, only outer shadow penumbræ are supported. In other

words, the computed shadow umbrae do not shrink properly as the size of the area light source increases. Our smoothie algorithm is closely related to Parker et al.’s method in that we extend extra geometry outwards from the blockers’ silhouettes. Consequently, our method is also limited to outer shadow penumbrae.

Assarsson and Akenine-Möller have published several papers describing soft shadow algorithms based on shadow volumes [AMA02, AAM03, ADMAM03]. They replace each shadow volume polygon with a *penumbra wedge*, a set of polygons that encloses the penumbra region for a given silhouette edge. Their original approach achieves visual smoothness by linearly interpolating light intensity within the wedge, and the wedges are constructed in a manner that supports inner shadow penumbrae.

Subsequent publications describe an improved wedge construction technique that increases robustness and can handle multiple wedges independently of each other [AAM03, ADMAM03]. When rendering a wedge, the shadow polygon for the corresponding silhouette edge is clipped against the image of the light source to estimate partial visibility. The clipping calculation is performed for each wedge at every pixel to accumulate light into a visibility texture. This geometry-based approach yields an accurate approximation, supports inner shadow penumbrae, and can handle animated light sources with different shapes.

Unfortunately, as we discussed in Section 2.1, shadow volumes do not scale well to complex scenes because of their large fillrate requirements. The penumbra wedge algorithms consume even more fillrate than ordinary shadow volumes, because each silhouette edge gives rise to multiple wedge polygons. Furthermore, a long pixel shader that performs visibility calculations must be executed for all rasterized wedge pixels.

Other techniques rely on projective texturing to avoid the cost of shadow volumes. For instance, Haines [Hai01] describes a method for rendering a hard drop shadow into a texture map and approximating a penumbra along the shadow silhouettes. The method works by drawing cones at the silhouette vertices and drawing sheets that connect the cones at their outer tangents. The cones and sheets are smoothly shaded and projected onto the ground plane to produce soft planar shadows. Like the work

of Parker et al. [PSS98], this construction only grows penumbra regions outward from the umbra and does not support inner shadow penumbras.

Recently, Wyman and Hansen [WH03] independently developed a soft shadow algorithm similar to our smoothie algorithm. They extend Haines’s work by drawing cones and sheets at the objects’ silhouettes and storing intensity values into a *penumbra map*, which is then applied as a projective texture to create soft shadow edges. We will compare their approach to ours in more detail in Chapter 4.

2.3 Additional Methods

So far we have discussed many hard and soft shadow rendering techniques based on the shadow map and shadow volume algorithms. There are many other classes of techniques, however. For instance, distributed ray tracing [CPC84] is a natural way of generating accurate soft shadows, but it is too expensive for interactive applications.

Researchers have recently developed a number of techniques based on precomputed radiance transfer. These techniques use a preprocessing step to compress the incoming lighting information (taking into account occlusions) with a set of basis functions such as spherical harmonics [RH01, SKS02] or Haar wavelets [NRH03]. These techniques allow relighting of the scene with high-quality soft shadows at interactive rates. Unfortunately, the long precomputation times preclude support for dynamic environments.

2.4 Summary

This chapter has examined several real-time shadow algorithms and their strengths and limitations. Most of these algorithms are based on the classic shadow map and shadow volume methods. Shadow maps work in image space and are efficient and general, but they suffer from undersampling artifacts and support only direction light sources. Shadow volumes work in object space and are robust and accurate, and they handle omnidirectional light sources. However, they support only a limited class of

geometry and do not scale well to complex scenes because of high fillrate requirements. Soft shadow algorithms that extend these basic approaches inherit the corresponding limitations.

While discussing these various techniques, we have seen examples of algorithms that are difficult to map to hardware, either because they require an expensive read-back [McC00, BS02] or because they require complicated data structures that are hard to update for dynamic environments [FFBG01]. We have also discussed algorithms, such as ray tracing and precomputed radiance transfer, that can lead to high-quality shadows, but they do not support dynamic environments efficiently.

In comparison, the algorithms we propose in this thesis are designed to work for dynamic scenes, to map directly to existing graphics hardware, and to avoid expensive readbacks. Our algorithms attempt to balance the quality and performance requirements using a mix of image-space and object-space approaches. For instance, the hybrid algorithm uses the image-space shadow map approach to determine shadows for all parts of the image except at the shadow edges, then uses the object-space shadow volume approach to compute accurate shadow edges. In the smoothie algorithm, we find silhouette edges and construct extra geometric primitives in object space, but ultimately we perform the visibility calculations in image space using texture lookups. In the next two chapters, we'll discuss these algorithms in detail and see how they achieve both efficiency and good image quality.

Chapter 3

Hybrid Shadow Rendering Using Computation Masks

As we discussed in the previous chapter, shadow maps and shadow volumes are commonly-used techniques for the real-time rendering of shadows. Shadow maps are efficient and flexible, but they are prone to aliasing. Shadow volumes are accurate, but they have large fillrate requirements and thus do not scale well to complex scenes. This chapter describes a way of combining these algorithms to achieve both accuracy and scalability.

Sen et al. [SCH03] observed that shadow-map aliasing is only noticeable at the discontinuities between shadowed and lit regions, i.e. at the shadow silhouettes. On the other hand, shadow volumes compute shadows accurately at every pixel, but this accuracy is needed only at the silhouettes. This observation suggests a hybrid algorithm that uses a slower but accurate algorithm near the shadow discontinuities and a faster, less exact algorithm everywhere else.

This chapter presents a hybrid algorithm for rendering hard shadows from point light sources (see Figure 3-1). We first use a shadow map to find quickly pixels in the image that lie near shadow silhouettes, then apply the shadow volume algorithm only at these pixels; the shadow map determines shadows for the remaining non-silhouette pixels. This approach greatly reduces the fillrate needed for drawing shadow volumes, because the number of silhouette pixels is often a small fraction of the shadow

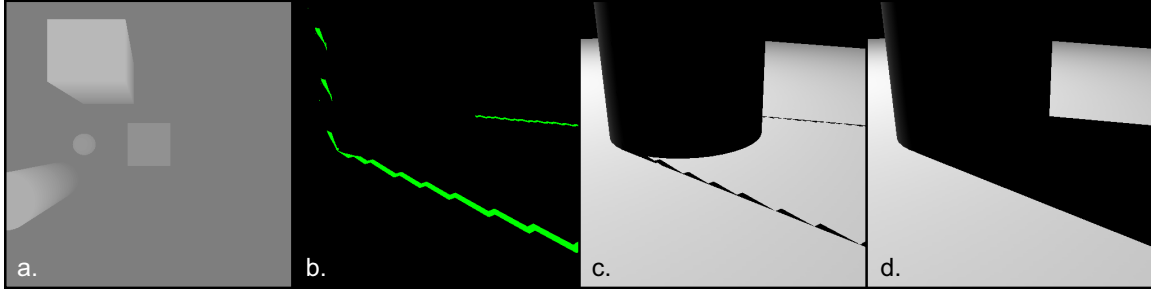


Figure 3-1: Overview. We first use a shadow map (a) to identify pixels in the image that lie close to shadow silhouettes. These pixels, seen from the observer’s view, are shaded green in (b). Next, we render shadow volumes only at these pixels to obtain accurate shadow edges (c). We use the shadow map to compute shadows everywhere else, and the final result appears in (d).

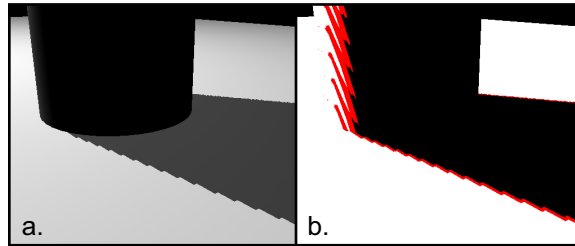


Figure 3-2: (a) The cylinder’s shadow exhibits aliasing due to shadow map under-sampling. However, aliasing is only apparent at the shadow edges. (b) Pixels that lie near shadow edges (shown in red) account for only a small fraction of the total image size.

polygons’ total screen area (see Figure 3-2). We show that our method produces accurate hard shadows and has substantially lower fillrate requirements than the original shadow volume algorithm.

To avoid processing non-silhouette pixels during shadow volume rasterization, we propose an extension to graphics hardware called a *computation mask*. Computation masks are useful in general for accelerating multipass rendering algorithms. Although they are not directly exposed in current hardware, we show how to simulate them efficiently using available features related to early z occlusion culling. Since computation masks exploit existing culling hardware, adding native hardware support requires minimal changes to modern graphics chips.

3.1 Algorithm

We assume that blockers are polygonal, well-behaved, closed, and manifold; these properties ensure a robust implementation of shadow volumes [EK02].

An overview of our approach is shown in Figure 3-1. We first create an ordinary shadow map, which serves to identify shadow silhouette pixels and compute shadows for non-silhouette pixels. Then, we use shadow volumes to compute accurate shadows only at silhouette pixels. The underlying assumptions are that the shadows' silhouette pixels account for a small fraction of the total number of shadow-polygon pixels, and that the hardware supports a mechanism for efficiently discarding pixels that do not lie on the silhouette.

We now explain these concepts in more detail; implementation and hardware issues are discussed in the next section. The algorithm's steps are:

1. Create a shadow map. We place the camera at the light source and render the nearest depth values to a buffer, as shown in Figure 3-1a. Since we only need the shadow map to approximate the shadow silhouette, we can use a low-resolution shadow map to conserve texture memory and speed up shadow-map rendering. The tradeoff is that low-resolution shadow maps can miss small features and usually increase the number of pixels classified as silhouette pixels. We will discuss this issue further in Section 3.3.1.

2. Identify shadow silhouette pixels in the final image. We render the scene from the observer's viewpoint and use a technique suggested by Sen et al. [SCH03] to find silhouette pixels. We transform each sample to light space and compare its depth against the four nearest depth samples from the shadow map. If the comparison results disagree, then we classify the sample as a silhouette pixel (shown in green in Figure 3-1b). Otherwise, the sample is a non-silhouette pixel and is shaded according to the depth comparison result.

Reducing the number of silhouette pixels is desirable because it limits the amount of stencil fillrate consumed when drawing shadow volumes. For example, pixels that are back-facing with respect to the light are always in shadow, so we never tag them

```

// Find discontinuities: shadow silhouette pixels.
void main (out half4 color      : COLOR,
           half diffuse        : COL0,
           float4 uvProj       : TEXCOORD0,
           uniform sampler2D shadowMap)
{
    // Use hardware's 2x2 filter: 0 <= v <= 1.
    fixed v = tex2Dproj(shadowMap, uvProj).x;

    // Requirements for sil pixel: front-facing and
    // depth comparison results disagree.
    color = (v > 0 && v < 1 && diffuse > 0) ? 1 : 0;
}

```

Figure 3-3: Cg pixel shader for finding shadow silhouettes.

as silhouette pixels. Additional methods for reducing the number of silhouette pixels are discussed in Section 3.4.3.

During this step, we also perform standard z -buffering, which leaves the nearest depth values seen from the observer's viewpoint in the depth buffer. This prepares the depth buffer for drawing shadow volumes in the next step.

3. Draw shadow volumes. The stencil shadow volume algorithm works by incrementing or decrementing the stencil buffer based on whether pixels of shadow-volume polygons pass or fail the depth test. We follow the z -fail setup described by Everitt and Kilgard [EK02] because of its robustness. The key difference in our approach is that we rasterize shadow-polygon pixels and update the stencil buffer only at framebuffer addresses containing silhouette pixels.

At the end of this step, the stencil buffer contains non-zero for pixels that lie in shadow; it contains zero for pixels that either are not shadowed or are not silhouette pixels. For example, the black shadow edges in Figure 3-1c show the regions where the stencil buffer contains non-zero.

4. Compute shadows. We draw and shade the scene only at pixels with stencil values equal to zero, thereby avoiding the shadowed regions of the image.

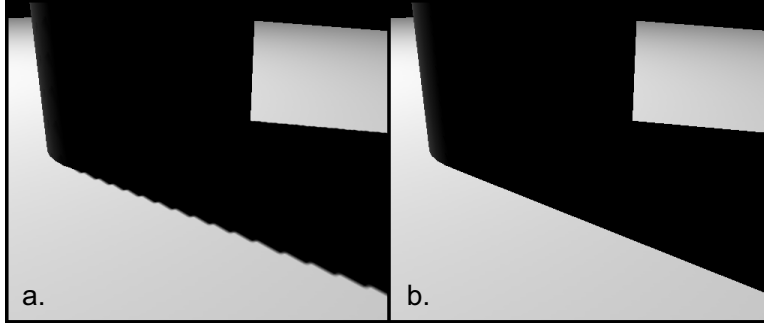


Figure 3-4: Comparison: a cylinder’s shadow is rendered using (a) a 512×512 shadow map and (b) our hybrid algorithm with a 256×256 shadow map. Using a lower-resolution shadow map in our case is acceptable because shadow volumes are responsible for reconstructing the shadow silhouette.

3.2 Implementation Issues

Our shadow algorithm performs rasterization and stencil updates of the shadow-volume polygons only at the silhouette pixels. To find these silhouette pixels, we compare the depth of each image sample with the four nearest depth samples in the shadow map and check if the results agree. We use hardware-assisted percentage closer filtering [RSC87] to accelerate this step. If the shadow query returns exactly 0 or 1, the depth comparison results agree and the pixel is not a silhouette pixel; otherwise the results disagree and the pixel lies on a silhouette. This optimization allows us to issue a single texture-fetch instruction in a pixel shader, shown in Figure 3-3. We tag silhouette pixels by writing 1 to the color buffer.

Rasterization and stencil updates of shadow-volume polygons are limited to silhouette pixels. We accomplish this task using a *computation mask*, a device that lets us pick specific framebuffer addresses to mask off so that the hardware can avoid processing pixels at those locations. Computation masks are useful for accelerating multipass rendering algorithms. For instance, Purcell et al. [PDC⁺03, Pur04] found that a computation mask with coarse granularity improved the performance of their hardware-based photon-mapping algorithm by factors of two to ten.

Current graphics hardware does not directly expose computation masks, but it turns out that the `EXT_depth_bounds_test` OpenGL extension [Ope02a] can be treated as one; see the appendix for a brief explanation of this extension. The idea is to use a

pixel shader to mask off pixels by setting their depth values to a constant *outside* the depth bounds. Then we enable depth-bounds testing so that in subsequent rendering passes, rasterized pixels at these masked-off framebuffer addresses can be discarded early in the pipeline. Similar culling mechanisms are commonly used for early z occlusion culling [Mor00].

In our implementation, we set up a computation mask as follows. We draw a screen-aligned quad and use a pixel shader to set the depth values of all non-silhouette pixels to $z = 0$; depth values of silhouette pixels are unmodified. Next, we enable depth-bounds testing and set the depth bounds to $[\epsilon, 1]$ for some small constant $\epsilon \approx 0.001$. Finally, we apply the robust z -fail variation of stencil shadow volumes [EK02]. Since the hardware discards all rasterized pixels whose depth values in the framebuffer are equal to $z = 0$, only silhouette pixels will be rendered.

Note that our implementation depends on the hardware's ability to preserve early culling behavior when using a pixel shader to compute depth values. This feature is available on the NVIDIA GeForce 6 (NV40) but not on earlier NVIDIA architectures such as the GeForce FX (NV30). ATI's Radeon 9700 (R300) and newer architectures also support this feature, but unfortunately those chips do not support the `EXT_depth_bounds_test` extension.

3.3 Results

All of the images presented in this section were generated at a resolution of 1024×768 on a 2.6 GHz Pentium 4 system with a NVIDIA GeForce 6800 (NV40) graphics card.

Examples. Figure 3-4 shows a cylinder casting a shadow onto the ground plane. We used an ordinary 512×512 shadow map and 2×2 bilinear percentage closer filtering [RSC87] for the image in Figure 3-4a. We used our hybrid method with a 256×256 shadow map for the image in Figure 3-4b. The lower-resolution shadow map is acceptable because the shadow-volume portion of our algorithm reconstructs the shadow edges accurately.

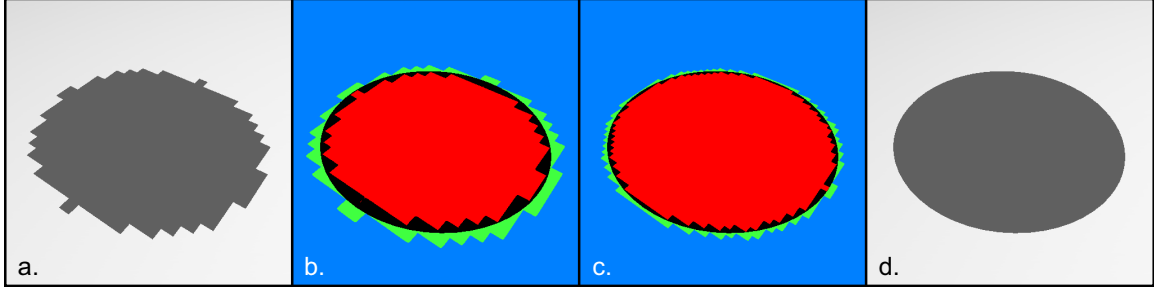


Figure 3-5: Visualization of mixing shadow maps and shadow volumes. We see the shadow of a ball cast onto the ground plane. (a) Aliasing is evident when the ball’s shadow is rendered using a 256×256 shadow map. The rest of the images illustrate how our method minimizes aliasing. In (b) and (c), non-silhouette pixels are shaded red and blue; for these pixels, the shadow map determines which ones are in shadow (red) and which ones are lit (blue). Silhouette pixels are shaded black and green; shadow volumes determine which ones are in shadow (black) and which ones are lit (green). Shadow map resolutions of 256×256 and 512×512 were used for (b) and (c), respectively. The final shadow is shown in (d).

Figure 3-5 shows a closeup of a ball’s shadow and illustrates how our method operates near shadow silhouettes. Figure 3-5a shows the aliasing artifacts that result from using an ordinary 256×256 shadow map. In the middle two images, non-silhouette pixels are shown in red and blue; red pixels are fully shadowed, and dark gray pixels are fully lit. Visibility for these pixels is determined using the shadow map. Silhouette pixels are shown in black and green; black pixels are in shadow and green pixels are lit. Shadow determination in this case is performed by shadow volumes. Figures 3-5b and 3-5c use 256×256 and 512×512 shadow maps, respectively. Figure 3-5d shows the final shadow computed by our method.

Methodology. We evaluated our method using the scenes shown in Figure 3-6. The Cubes and Dragon Cage scenes contain 12,000 triangles each, and the Tree scene has 40,000 triangles. We chose these scenes and viewpoints for several reasons. First, they have high shadow complexity and require enormous fillrate when using ordinary shadow volumes. Second, these scenes have many overlapping shadow edges, which can lead to temporal artifacts when using shadow silhouette maps [SCH03]. Third, we have chosen some camera views (see View C) that are on the opposite side of the blockers as the light source; these cases are difficult to handle using perspective

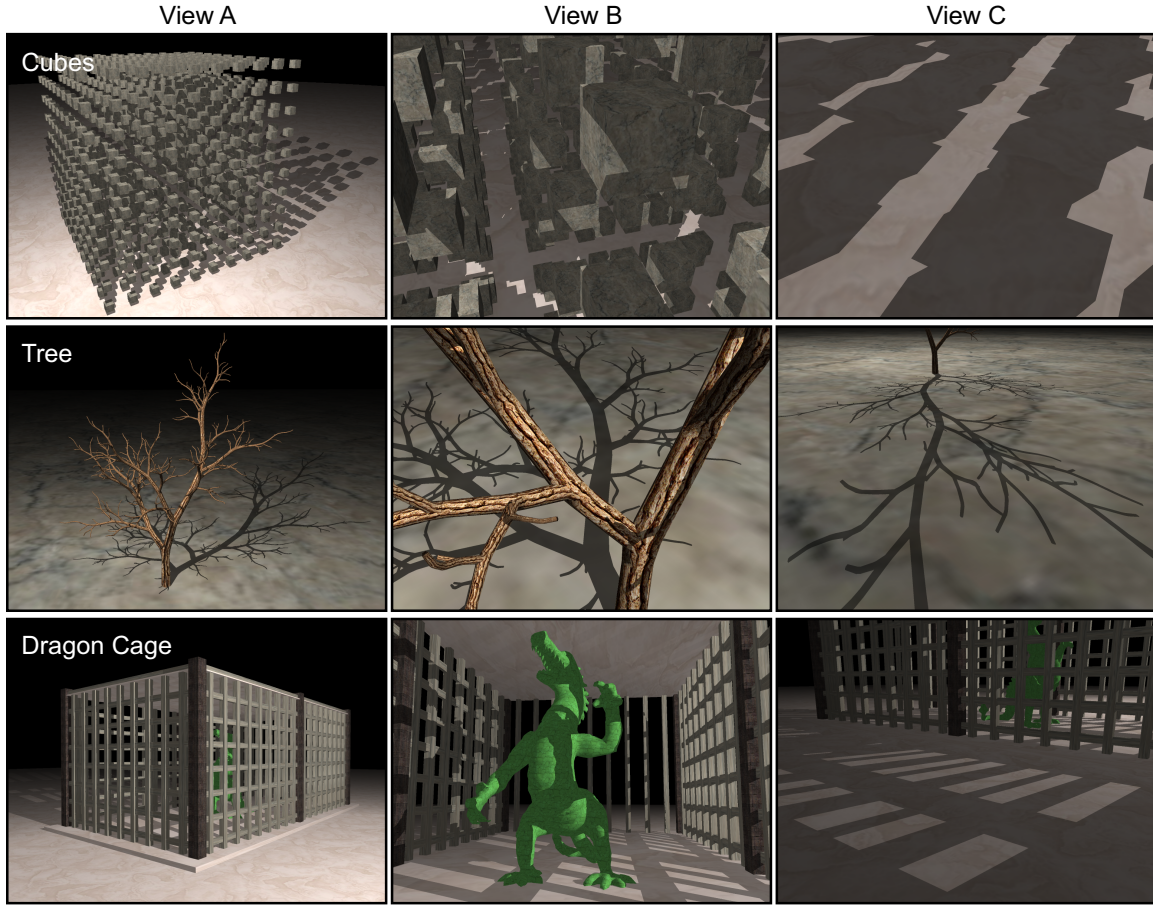


Figure 3-6: Three test scenes with high shadow complexity. Rows contain different scenes, and columns show different views. Each scene is illuminated using a single point light source.

shadow maps [SD02]. Finally, these scenes are heavily-shadowed, and the depth range of shadow-volume polygons is large, making it difficult to apply the scissor and depth-bounds optimizations described in Section 2 [Len02, EK03, MHE⁺03]. In summary, real-time shadow rendering is a challenging task in all of these scenes.

3.3.1 Image Quality

Figure 3-7 compares the image quality of shadow maps, our hybrid method, and shadow volumes; we used a 1024×1024 shadow map for the first two techniques. These images show that our method minimizes the edge-aliasing artifacts of shadow maps.

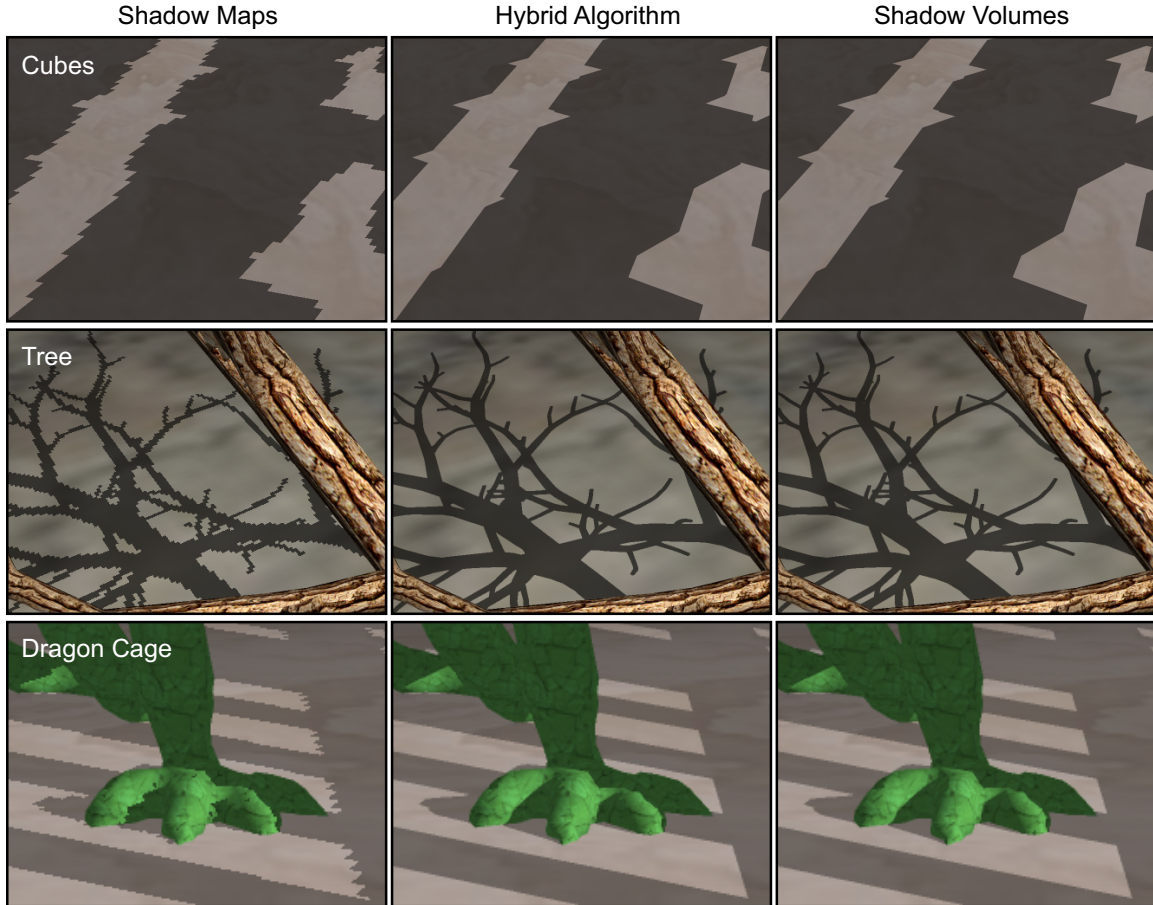


Figure 3-7: Comparison of image quality using shadow maps (left column), our hybrid algorithm (center column), and shadow volumes (right column). A shadow-map resolution of 1024×1024 was used for the shadow map and hybrid algorithms.

A more subtle improvement is that our method reduces self-shadowing artifacts. With regular shadow maps, incorrect self-shadowing may occur due to limited depth-buffer resolution and precision. These artifacts are visible, for example, in the Dragon Cage scene in Figure 3-7 (see the lower-left image). The problem is usually addressed by adding a small bias to the depth values when rendering the shadow map [AMH02, RSC87, Wil78]. Unfortunately, the amount of bias required depends on the scene configuration and is hard to set automatically for dynamic scenes.

In our approach, however, incorrectly-shadowed pixels are often classified as silhouette pixels and thus are rendered correctly by the shadow-volume portion of the algorithm. The reason is that the depth value of an affected pixel usually lies between the depth values of two adjacent samples in the shadow map. As a result, the depth

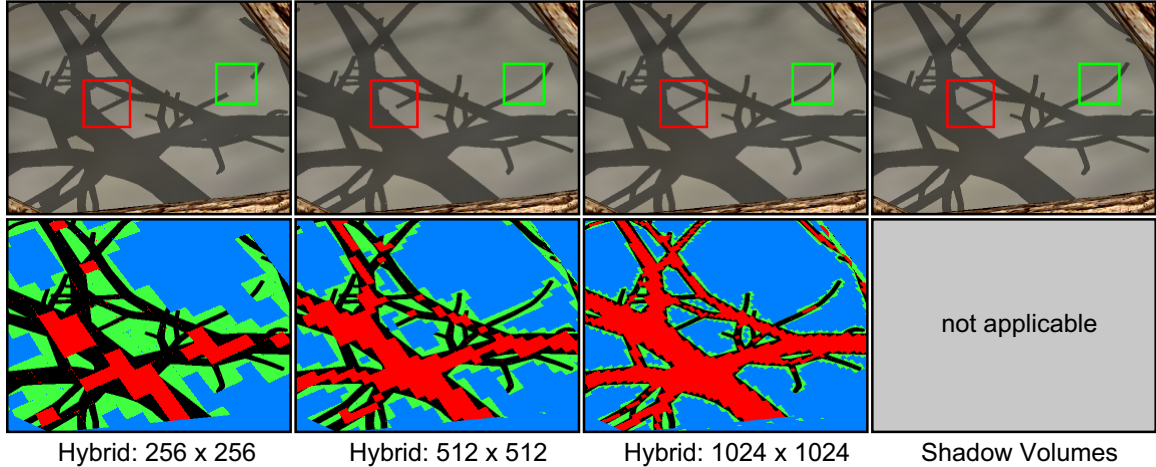


Figure 3-8: Artifacts. These images are crops from the Tree scene, View B. The images in the left three columns were generated using our hybrid algorithm with varying shadow-map resolutions. In the top row, the regions indicated in red and green show missing or incorrect shadows due to undersampling in the shadow map. The corresponding images in the bottom row visualize the reconstruction errors using the same color scheme as in Figures 3-5b and 3-5c. The reference image in the far-right column was obtained using shadow volumes.

comparisons disagree and the pixel is tagged as a silhouette pixel. One implication is that we can choose the shadow bias conservatively, erring on the side of applying too little bias and relying on the shadow volumes to avoid self-shadowing artifacts. If we apply too little bias, however, then most of the pixels in the image will be classified as silhouette pixels.

Although shadows computed using our approach are often similar to those computed using shadow volumes, small differences may occur due to sampling errors in the shadow map. To understand these differences better, we studied several images produced using the hybrid algorithm at different shadow-map resolutions. Figure 3-8 shows an example of one such set of images; we chose the Tree scene as our example because its thin branches are difficult to represent accurately in a discrete buffer. The indicated regions in red and green show missing or incorrect shadows due to undersampling.

More generally, a limitation of using lower shadow-map resolutions is that small blockers may be poorly represented in the shadow map. This form of aliasing manifests itself in vanishing and popping shadows. Existing shadow algorithms that rely

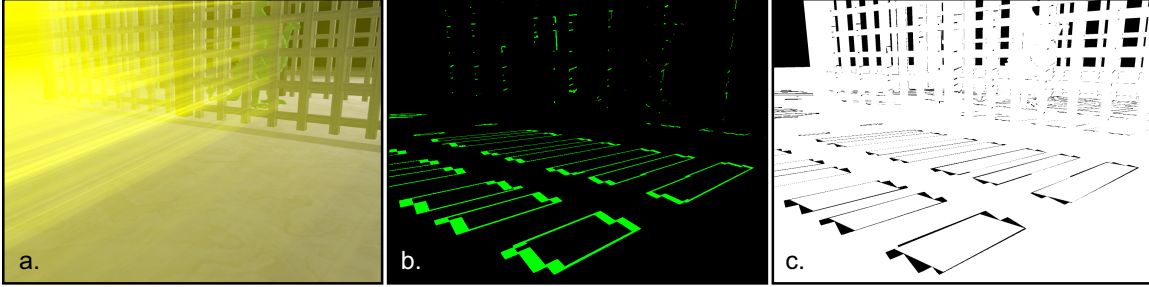


Figure 3-9: Fillrate consumption and overflow in the Dragon Cage scene. The shadow volume polygons, shaded yellow in (a), cover the entire image and have an overflow factor of 79; brighter yellow corresponds to higher overflow. Our hybrid method restricts drawing shadow polygons to the silhouette pixels, shaded green in (b); these pixels cover just 5% of the image. The image on the right (c) illustrates the resulting stencil buffer: black pixels on the floor and walls are in shadow and represent non-zero stencil values.

on discrete buffers for visibility queries, such as the work of McCool [McC00], Govindaraju et al. [GLY⁺03], and Sen et al. [SCH03], also exhibit similar artifacts. The difficulty is that arbitrarily small objects may cast arbitrarily large shadows, depending on the scene configuration, and low-resolution shadow maps are more likely to miss such objects. This problem could be addressed by combining our method with a perspective shadow map [Koz04, SD02], which optimizes the depth buffer’s sample distribution to maximize resolution near the viewer.

3.3.2 Performance

Fillrate consumption is significant for the shadow volume algorithm in all of our test scenes. Figure 3-9a shows an example in which shadow-volume polygons cover the entire image and have an overflow factor of 79, meaning that every pixel is processed 79 times on average. We reduce this huge fillrate consumption by limiting shadow-polygon rasterization to silhouette pixels, shaded green in Figure 3-9b; silhouette pixels in this scene cover just 5% of the image. Performing rasterization and stencil updates only at these pixels leads to the stencil buffer shown in Figure 3-9c.

Figure 3-10 compares the number of pixels rasterized by shadow volumes and our hybrid method. It also shows the percentage of pixels classified as silhouette pixels as

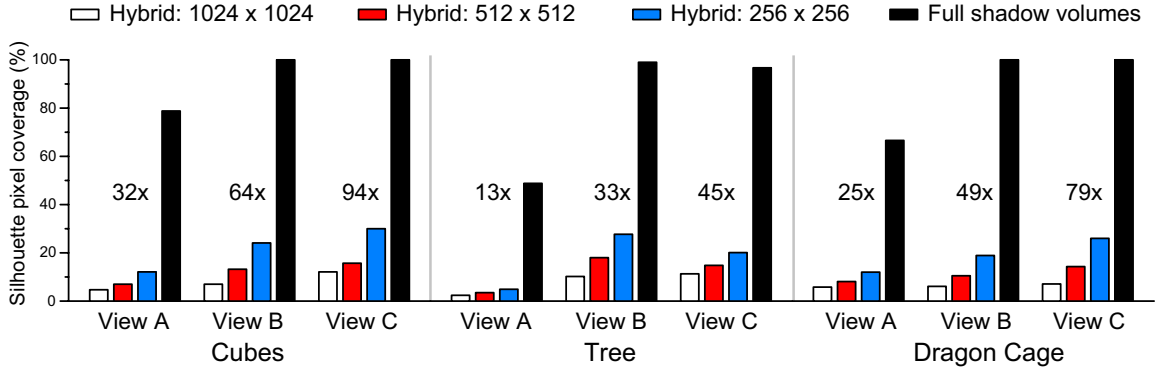


Figure 3-10: Shadow volume overdraw and silhouette pixel coverage. The black bars show the percentage of pixels in the image covered by shadow volumes, and the number next to each bar is the overdraw factor (the ratio of shadow-polygon pixels to the total image size). The white, red, and blue bars show the percentage of pixels in the image that are classified as shadow silhouette pixels by our hybrid algorithm; colors correspond to different shadow-map resolutions.

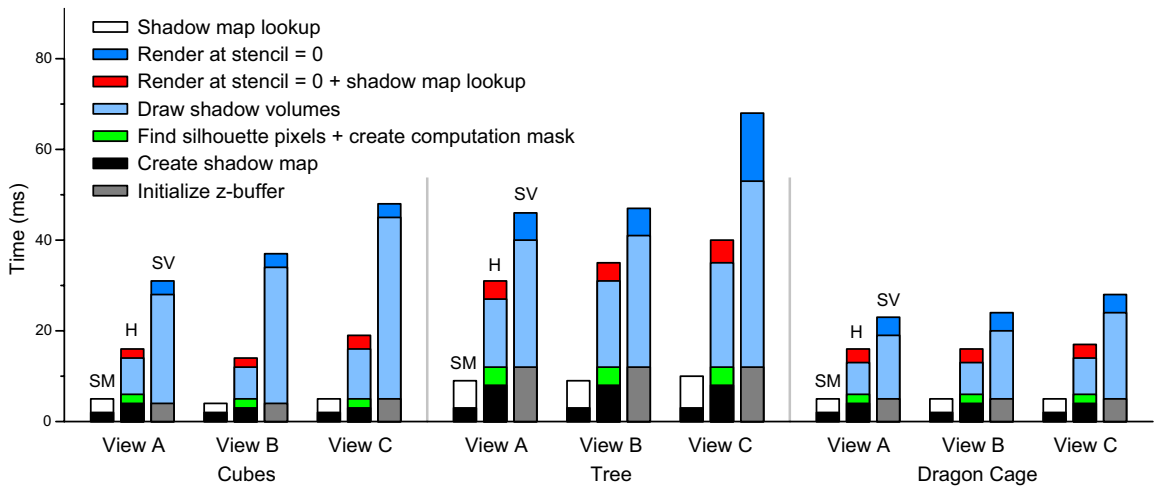


Figure 3-11: Performance comparison. The vertical bars measure the per-frame time in milliseconds (ms) for the shadow map algorithm (SM), our hybrid algorithm (H), and the shadow volume algorithm (SV). Colored sections of a bar indicate how much time is spent in each part of the algorithm.

a function of shadow-map resolution. Even with a 256×256 shadow map, the fraction of silhouette pixels is much smaller than the fraction of all shadow-volume pixels.

Figure 3-11 compares the performance of the shadow map, shadow volume, and hybrid algorithms. Performance is measured using the time required for each algorithm to render one frame; all times are reported in milliseconds. The plot gives a performance breakdown for each part of each algorithm. Not surprisingly, the cost

of the hybrid and shadow volume algorithms is dominated by the rasterization and stencil updates of the shadow-volume polygons. Our method is significantly faster, however: we observed speedups of 30% to over 100%, measured in frames per second. Keep in mind that these performance numbers are highly scene-dependent and view-dependent; hardware culling implementations (see Section 3.4.2) also play a large role in determining the actual speedup. We provide performance numbers simply to demonstrate that substantial acceleration is attainable across a number of different scenes and viewpoints.

3.4 Discussion

3.4.1 Algorithm Tradeoffs

The key performance tradeoff in our work is the reduction of fillrate at the cost of an extra rendering pass. We emphasize that our method is designed to handle dynamic scenes with high shadow complexity, which would ordinarily give rise to many overlapping shadow volumes and quickly saturate the hardware’s fillrate. Thus our method is most relevant to fillrate-limited applications, such as many of the current real-time game engines. Ordinary shadow volumes will clearly run faster for scenes of sufficiently low shadow complexity.

Since our method combines both shadow maps and shadow volumes, it inherits some of the limitations from both. In contrast to the shadow map algorithm, which handles any geometry that can be represented in a depth buffer, our method requires watertight polygonal models for robust shadow volume rendering. In contrast to the shadow volume algorithm, our method is restricted to directional light sources because we use shadow maps to find silhouette pixels; omnidirectional lights require additional rendering passes. Finally, our method requires one more rendering pass than ordinary shadow volumes because we must first create the shadow map. Fortunately, this extra pass is inexpensive because it can be done at lower resolutions and requires no shading.

3.4.2 Computation Masks

We described at length in Section 3.2 how to treat the depth bounds test as a computation mask, but this trick is only necessary because current hardware lacks a dedicated computation mask. We believe that adding a true computation mask to graphics hardware is worthwhile for several reasons. First, as we pointed out earlier, computation masks are closely related to early z occlusion culling, and thus most of the required technology is already present in current hardware. In particular, computation masks can take advantage of the early tile-based rejection mechanisms already used for occlusion culling and depth bounds testing. Furthermore, the representation for a computation mask is much more compact: only a single bit per pixel is needed, as compared to 16 or 24 bits per pixel for an uncompressed depth buffer.

Early pixel rejection (due to either computation masks, occlusion culling, or depth bounds testing) is unlikely to occur with single-pixel granularity. It is more likely that such culling takes place on a per-tile basis, such as a 16-pixel or 8-pixel tile. Fortunately, the images in Figures 3-1b and 3-9b suggest that non-silhouette pixels tend to occupy large contiguous regions of screen space and can benefit from conservative tile-based culling.

3.4.3 Additional Optimizations

One way to reduce further the number of classified silhouette pixels is to consider only the pixels that undersample the shadow map. Stamminger and Drettakis [SD02] derive a simple formula for estimating the ratio r of image resolution to shadow-map resolution; silhouette pixels with $r < 1$ can be omitted because they won't exhibit aliasing artifacts. This culling strategy could be added to the shader in Figure 3-3 at the cost of additional per-pixel floating-point arithmetic. In our test cases, however, we found that this technique reduced the number of classified silhouette pixels by only 5%, not enough to justify the computational overhead.

We have also considered (but not implemented) an optimization inspired by the work of Lloyd et al. [LWGMar] and the hybrid algorithms of McCool [McC00] and

Govindaraju et al. [GLY⁺03]. As mentioned earlier, an advantage of McCool’s work is that shadow volumes are not needed for blockers that are entirely in shadow, because these blockers are absent from the shadow map. Similarly, Lloyd et al. and Govindaraju et al. use image-space occlusion queries to reduce the number of blockers and receivers considered for shadow computation.

These occlusion queries could be combined with our algorithm in the following way. After rendering a shadow map in the first step of the algorithm, render a bounding volume for each blocker from the light’s viewpoint and use an occlusion query to check if any of the bounding volume’s pixels pass the depth test. If no pixels pass, then drawing the shadow volumes for that blocker may be skipped. Note that this culling strategy must be applied on a per-blocker basis (as opposed to a per-silhouette-edge basis) to ensure the consistency of the stencil-based shadow volume algorithm.

This optimization is useful in situations where complex blockers are often shadowed; a common scenario is a computer game in which monsters hide in the shadows. In these cases, it may be possible to avoid drawing the shadow polygons entirely for a large model. The cost of the optimization includes the occlusion query, which adds latency to the rendering pipeline, and the drawing of bounding boxes, which consumes additional fillrate. On the other hand, most of the latency can be hidden by issuing many queries but only checking the result of the first query after several bounding volumes have been drawn. Drawing a bounding box is also fast because there are no writes to the framebuffer, no shading is needed, and hardware-accelerated occlusion culling is applicable.

3.5 Conclusions and Future Work

We have presented a hybrid shadow rendering approach that combines the strengths of shadow maps and shadow volumes. The key idea is to use shadow maps to identify which regions of the image will exhibit edge aliasing, then apply shadow volumes only in those regions to obtain better accuracy.

We have shown that our algorithm benefits from using a computation mask that

discards pixels early in the pipeline. More generally, a computation mask allows the programmer to identify a small set of pixels in the scene that are “interesting”; all other pixels are masked off. Expensive per-pixel algorithms can then operate on this subset of pixels without incurring the cost over the entire framebuffer. This strategy of decomposing a problem into two parts — a large part that relies on a fast but inexact technique, and a small part that uses a slower but accurate technique — is applicable to a number of multipass rendering algorithms.

We believe that the benefits of hardware computation masks will become more significant as pixel shaders increase in complexity. Current hardware already performs aggressive occlusion culling to avoid unnecessary shading. Computation masks represent a logical step in this direction.

Notes

The `EXT_depth_bounds_test` OpenGL extension [Ope02a] states that a rasterized fragment is discarded if the current depth value stored in the framebuffer at that fragment’s address lies outside user-specified depth bounds. The key is that an architecture’s implementation may discard fragments early in the pipeline without performing buffer updates, such as stencil writes. Thus modern graphics architectures, which often rasterize and shade tiles of fragments in parallel, can aggressively discard an entire tile during rasterization if all the fragments in the tile lie outside of the depth bounds. Similar mechanisms are already used for early z occlusion culling [Mor00].

Chapter 4

Rendering Fake Soft Shadows

Using Smoothies

Whereas the previous chapter proposed an algorithm for rendering hard shadows, this chapter presents an approximate soft shadow algorithm. Specifically, we describe an algorithm that combines shadow maps with geometric primitives that we call *smoothies* to render fake soft shadows (see Figure 4-1). Although our method is not geometrically accurate, the resulting shadows appear qualitatively like soft shadows. Specifically, the algorithm:

1. hides undersampling artifacts such as aliasing,
2. generates soft shadow edges that resemble penumbrae,
3. performs shadow calculations efficiently in image space,
4. maps well to programmable graphics hardware, and
5. naturally handles dynamic scenes.

We emphasize that our method does *not* compute geometrically-correct shadows. Instead, we focus on the qualitative aspects of penumbrae without modeling an area light source. For example, we compute the penumbra size using the ratio of distances between the light source, blocker, and receiver, but we consider neither the shape

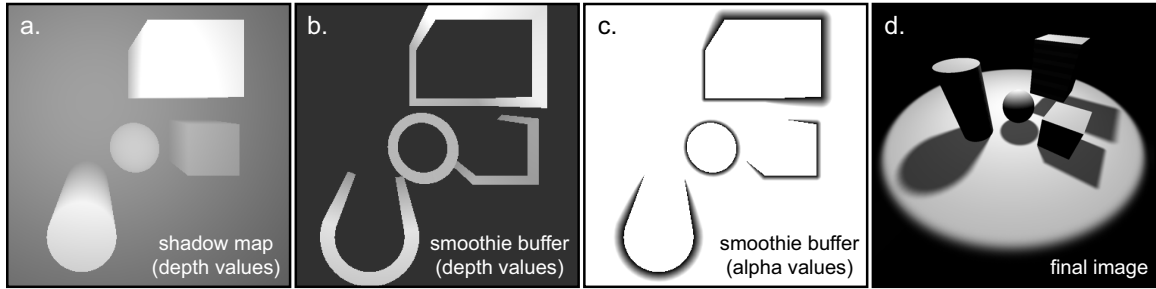


Figure 4-1: Smoothie algorithm overview. (a) We first render a shadow map from the light’s viewpoint. Next, we construct geometric primitives that we call smoothies at the objects’ silhouettes. (b) We render the smoothies’ depth values into the smoothie buffer. (c) For each pixel in the smoothie buffer, we also store an alpha value that depends on the ratio of distances between the light source, blocker, and receiver. (d) Finally, we render the scene from the observer’s viewpoint. We perform depth comparisons against the values stored in both the shadow map and the smoothie buffer, then filter the results. The smoothies produce soft shadow edges that resemble penumbrae.

nor orientation of the light source. In this sense, we have chosen a phenomenological approach.

This work was originally published in the *Proceedings of the Eurographics Symposium on Rendering 2003* [CD03].

4.1 Overview

An overview of our approach is shown in Figure 4-1. We first render an ordinary shadow map from the light’s viewpoint. Next, we construct geometric primitives that we call *smoothies* at the objects’ silhouettes (see Figure 4-2). We render the smoothies into a *smoothie buffer* and store a depth and alpha value at each pixel. The alpha value depends on the ratio of distances between the light source, blocker, and receiver. Finally, we render the scene from the observer’s viewpoint. We combine the depth and alpha information from the shadow map and smoothie buffer to compute soft shadows that resemble penumbrae (see Figure 4-3). A limitation of this approach is that computed shadow umbrae do not diminish as the area of the light source increases.

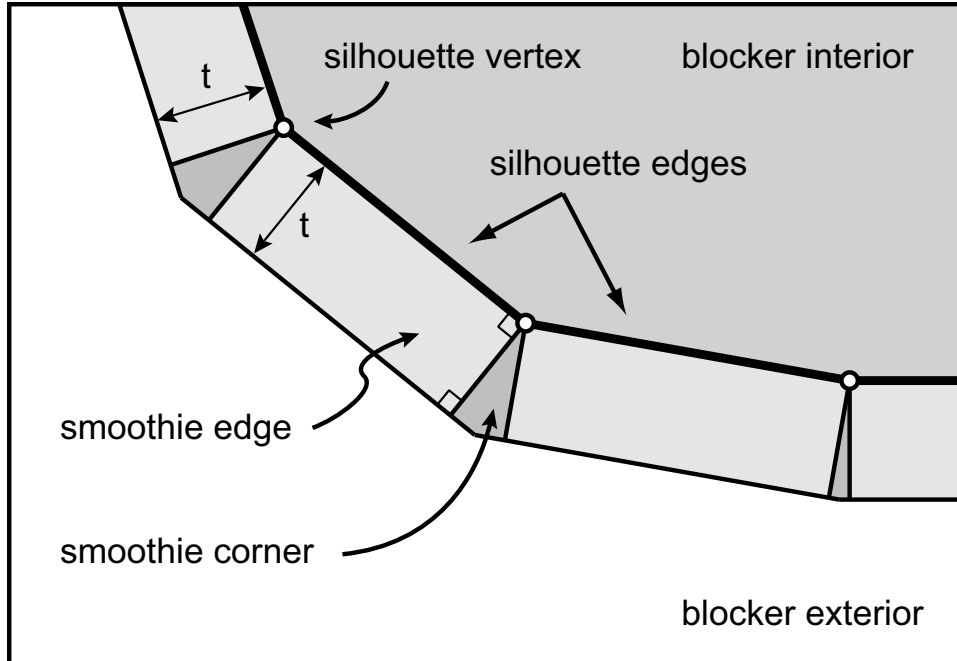


Figure 4-2: Smoothie construction. A smoothie edge is obtained by extending a blocker’s silhouette edge outwards to form a rectangle in screen space. A smoothie corner connects adjacent smoothie edges.

Our proposed algorithm is inspired by many different classes of techniques. For example, we identify silhouettes with respect to the light source in object space, an idea borrowed from shadow volumes. In some sense, our method is a hybrid between the “outer surface” approach of Parker et al. [PSS98] and the convolution approach of Soler and Sillion [SS98]. Specifically, smoothies are defined in object space and are related to the outer volume, but they can also be seen as pre-convolved shadow edges. Smoothies are also related to the work of Lengyel et al. [LPFH01], who showed how texture-mapped fins can improve the quality of fur rendering at the silhouettes.

We make the following assumptions in our approach. First, blockers are opaque; we do not handle shadows due to semi-transparent surfaces. Second, blockers are represented as closed triangle meshes; this representation simplifies the task of finding object-space silhouette edges. Finally, our method does not take into account the shape and orientation of light sources, so we assume that light sources are roughly spherical.

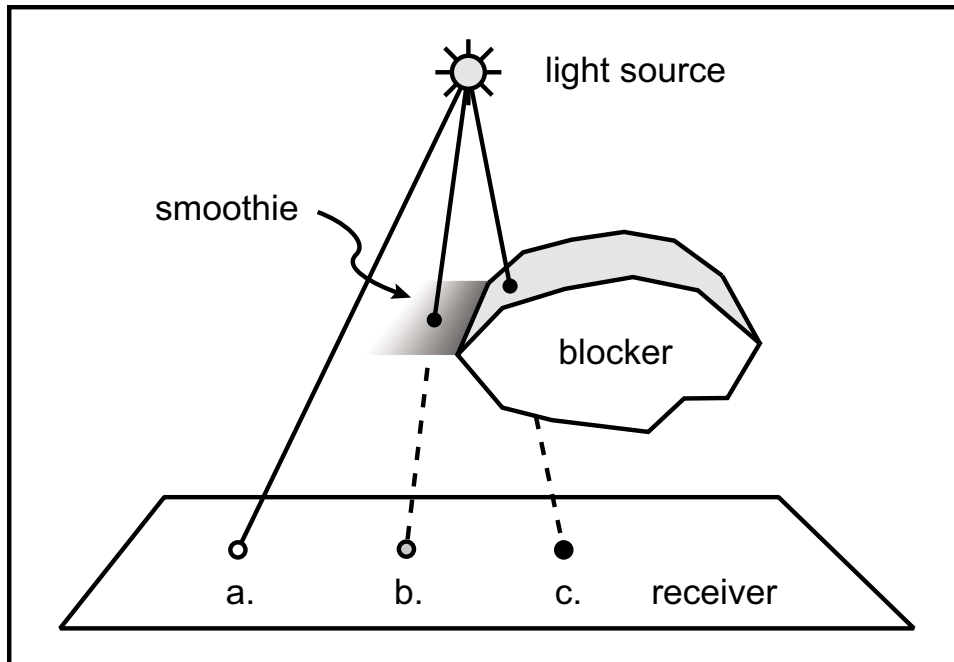


Figure 4-3: Three possible scenarios: an image sample is either (a) illuminated, (b) partially in shadow, or (c) completely in shadow.

4.2 Algorithm

The smoothie algorithm has five steps:

1. **Create a shadow map.** This is done by rendering the blockers from the light source and storing the nearest depth values to a buffer (see Figure 4-1a).

2. **Identify silhouette edges.** We identify the silhouette edges of the blockers with respect to the light source in object space. Since we assume that blockers are represented as closed triangle meshes, a silhouette edge is simply an edge such that one of its triangles faces towards the light and the other triangle faces away.

Silhouette detection algorithms have been developed for many applications in computer graphics, including non-photorealistic rendering and illustrations [IFH⁺03, Ras01]. We use a simple brute-force algorithm, looping over all the edges and performing the above test for each edge. For static models, Sander et al. [SGG⁺00] and Bala et al. [BWG03] describe more efficient hierarchical algorithms.

3. **Construct smoothies.** We construct a smoothie edge for each silhouette

edge and a smoothie corner for each silhouette vertex, as shown in Figure 4-2. A *smoothie edge* is obtained by extending the silhouette edge away from the blocker to form a rectangle of a fixed width t in the screen space of the light source. A *smoothie corner* connects adjacent smoothie edges. The variable t is a user parameter that controls the size of the smoothies. As we will see later, larger values of t can be used to simulate larger area light sources.

Arbitrary closed meshes may have silhouette vertices with more than two adjacent silhouette edges. We treat this situation as a special case. First, we average the adjacent face normals to obtain a shared vertex normal \vec{n} whose projection in the screen space of the light source has length t . Then we draw triangles that connect \vec{n} with each of the adjacent smoothie edges.

4. Render smoothies. We render each smoothie from the light’s viewpoint. This is similar to generating the shadow map in Step 1, but this time we draw only the smoothies, not the blockers. We compute two quantities for each rendered pixel, a depth value and an alpha value, and store them together in a *smoothie buffer* (see Figures 4-1b and 4-1c). We discuss how to compute alpha in Section 4.2.1.

5. Compute shadows with depth comparisons. We render the scene from the observer’s viewpoint and compute the shadows. We use one or two depth comparisons to determine the intensity value v at each image sample, as illustrated in Figure 4-3:

1. If the sample’s depth value is greater than the shadow map’s depth value, then the sample is completely in shadow ($v = 0$). This is the case when the sample is behind a blocker.
2. Otherwise, if the sample’s depth value is greater than the smoothie buffer’s depth value, then the sample is partially in shadow ($v = \alpha$, where α is the smoothie buffer’s alpha value). This case occurs when the sample is not behind a blocker, but is behind a smoothie.
3. Otherwise, the sample is completely illuminated ($v = 1$).

For better antialiasing, we perform these depth comparisons at the four nearest samples of the shadow map and smoothie buffer, then bilinearly interpolate the results. This is similar to percentage closer filtering [RSC87] using a 2×2 tent filter. The filtered visibility value can be used to modulate the surface illumination.

In summary, the smoothie algorithm involves three rendering passes. The first pass generates a standard shadow map; the second pass renders the smoothies' alpha and depth values into the smoothie buffer; and the final pass performs depth comparisons and filtering to generate the shadows. To handle dynamic scenes with multiple light sources, we follow the above steps for each light source per frame.

4.2.1 Computing Smoothie Alpha Values

We return to the discussion of computing alpha values when rendering smoothies in Step 4 of the algorithm. Some of the design choices described below are motivated by current graphics hardware capabilities.

The smoothie's alpha is intended to simulate the gradual variation in light intensity within the penumbra. Thus alpha should vary continuously from 0 at one edge to 1 at the opposite edge, which can be accomplished using linear interpolation (see Figure 4-4a).

There are two problems with this approach, both shown in Figure 4-4b. One problem occurs when two objects are close together, because the smoothie of one object casts a shadow that does not appear to originate from that object. The second problem is that computing alpha values in the manner described above with fixed-size smoothies will generate a penumbra whose thickness is also fixed, which does not model the behavior of true penumbrae well.

The shape of a penumbra depends on many factors and is expensive to model accurately, so we focus on its main qualitative feature: the ratio of distances between the light source, blocker, and receiver (see Figure 4-5). The size of a smoothie, rather than being fixed, should depend on this ratio. We can estimate the distance from the light to the blocker by computing the distance from the light to the smoothie. Furthermore, we can find the distance from the light source to the receiver using the

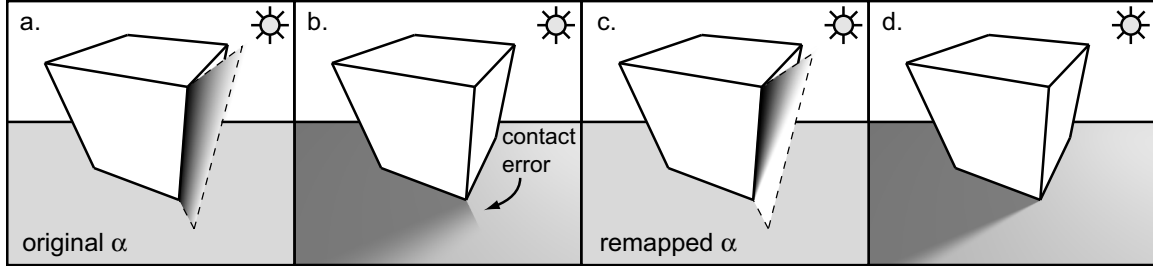


Figure 4-4: Computing smoothie alpha values. Diagram (a) shows a smoothie with the original, linearly interpolated alpha. (b) The resulting soft shadow edge has a fixed thickness and also has an obvious discontinuity at the contact point between the box and the floor. In diagram (c) we have remapped the alpha values as described in Equation 4.1. (d) The remapping fixes the contact problem and also creates a soft shadow edge that more closely resembles a penumbra.

shadow map rendered in Step 1.

Using the shadow map to resize the smoothies on graphics hardware requires texture accesses within the programmable *vertex* stage, a feature that has only recently appeared in the latest generation of graphics hardware, such as NVIDIA’s GeForce 6 (NV40) architecture [NVI04]. We have not yet experimented with this new architecture due to limited availability.

Alternatively, we can work around both of the problems discussed above by rendering the fixed-size smoothie edges using a pixel shader that remaps the alpha values appropriately. Let α be the linearly interpolated smoothie alpha at a pixel, b be the distance between the light source and smoothie, and r be the distance between the light source and receiver. We compute a new α' using

$$\alpha' = \frac{\alpha}{1 - b/r} \quad (4.1)$$

and clamp the result to $[0, 1]$. The remapping produces a new set of alpha values similar to alphas that *would* have been obtained by keeping the original α and adjusting the smoothie size. An example of remapping the alpha values is shown in Figures 4-4c and 4-4d.

Smoothie corners must ensure a continuous transition in alpha between adjacent

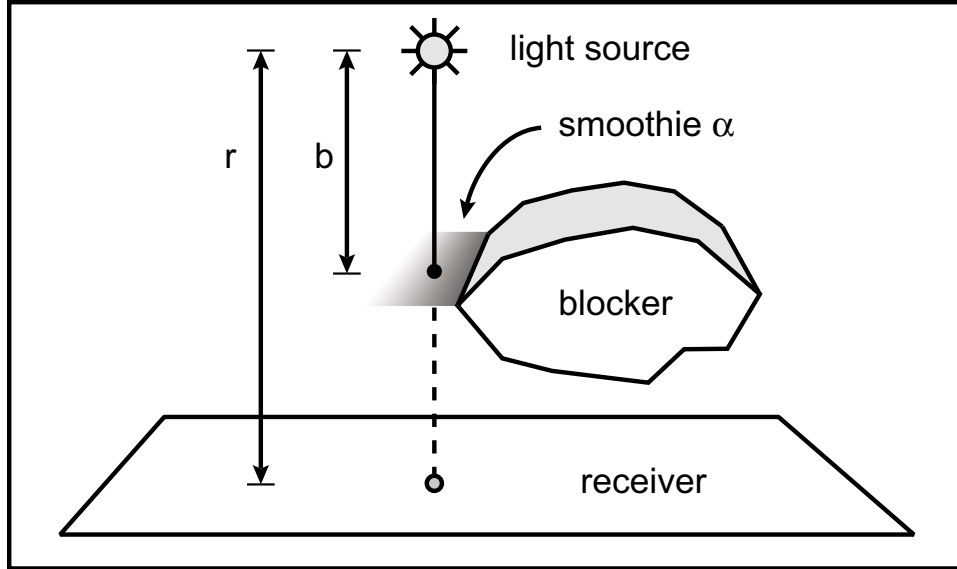


Figure 4-5: Alpha computation. The penumbra size depends on the ratio b/r , where b is the distance between the light and the blocker, and r is the distance between the light and the receiver.

smoothie edges. Figure 4-6 shows a smoothie corner rooted at the silhouette vertex \vec{v} . For a sample point \vec{p} within the corner, we compute $\alpha = |\vec{v} - \vec{p}|/t$, clamp α to $[0, 1]$, and remap the result using Equation 4.1. The resulting shaded corner is shown in the right image of Figure 4-6.

The case where multiple smoothies overlap in the screen space of the light source corresponds to the geometric situation where multiple blockers partially hide the extended light source. The accurate computation of visibility can be performed using Monte Carlo [CPC84] or backprojection [BRW89, DF94] techniques, both of which are expensive. Instead, we use minimum blending, a simple yet effective approximation. Minimum blending, which Parker et al. [PSS98] refer to as *thresholding*, just keeps the minimum of all the alpha values. This has the effect of ensuring continuous shadow transitions without making the overlapping region appear too dark. Blending in this manner is not geometrically accurate, but it is much simpler, more efficient, and still gives visually acceptable results. Note that we discard smoothie pixels that lie behind a blocker instead of blending them into the smoothie buffer.

The method for computing alpha values described above gives a linear falloff

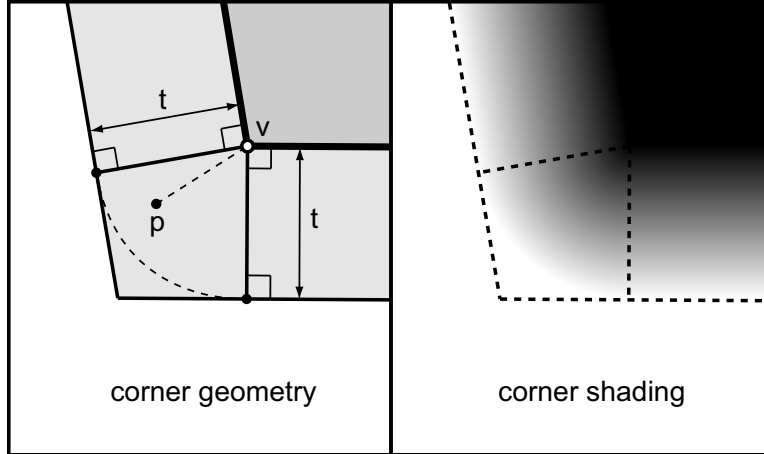


Figure 4-6: Smoothie corner geometry and shading. A sample point \vec{p} within the corner is assigned $\alpha = |\vec{v} - \vec{p}|/t$, which is then clamped to $[0, 1]$ and remapped using Equation 4.1.

of light intensity within the penumbra. Parker et al. [PSS98] note, however, that the falloff due to a diffuse spherical light source is sinusoidal, not linear. In our implementation, we follow Haines’s suggestion [Hai01] and precompute the sinusoidal falloff into a one-dimensional texture map. We then use the linear remapped alpha value to index into this texture map, which yields a more realistic penumbra.

4.3 Implementation

We implemented the smoothie algorithm using OpenGL and the Cg shading language [MGAKar]. The algorithm maps directly to the vertex and pixel shaders of programmable graphics hardware without requiring expensive readbacks across the AGP bus. Since the hardware does not retain vertex connectivity information, however, identifying silhouette edges must be done on the host processor. Our code is currently optimized for the NVIDIA GeForce FX [NVI02] and is split into four rendering passes:

Pass 1. We generate the shadow map by storing the blockers’ depth values into a 16-bit floating-point buffer. In the OpenGL graphics pipeline, depth values are stored in screen space and hence are non-linearly distributed over the view frustum.

We require linearly-distributed depth values, however, to find the ratio of distances between the light source, blocker, and receiver. Thus we compute depth values in world space using a vertex shader.

Pass 2. We draw the smoothies and store their depth values into a standard OpenGL depth buffer.

Pass 3. We redraw the smoothies and compute their alpha values in a pixel shader, as described in Section 4.2.1. We initialize a separate fixed-point buffer to $\alpha = 1$ and composite the smoothies' alpha values into this buffer using minimum blending. This blend mode is supported in hardware through the `EXT_blend_minmax` OpenGL extension.

Pass 4. We render the scene from the observer's viewpoint and compute the shadows in a pixel shader. We accelerate the depth comparisons and 2×2 filtering using the native hardware shadow map support.

The Cg code for these rendering passes is shown in Figure 4-7. We first compiled the shaders to the NVIDIA vertex and fragment OpenGL extensions [Kil02], then optimized the assembly code by hand to improve performance. Most of the per-pixel calculations use 12-bit fixed-point precision or 16-bit floating-point precision for faster arithmetic; this choice does not appear to affect image quality. The table at the bottom-right of Figure 3-3 shows the number of shader instructions for each rendering pass.

Additional optimizations are possible on different hardware. For example, the ATI Radeon 9700 supports multiple render targets per pass. Thus the second and third passes above could be combined into a single pass that stores the smoothies' depth and alpha values into separate buffers.

4.4 Results

All of the images presented in this section and in the accompanying video were generated at a resolution of 1024×1024 on a 2.6 GHz Pentium 4 system with an NVIDIA Geforce FX 5800 Ultra graphics card.

<pre> // Pass 1 (compute linear z): vertex program. void main (float4 pObj : POSITION, out float4 pClip : POSITION, out float zLinear : TEXCOORD0, uniform float4x4 mvp, uniform float4x4 mv) { pClip = mul(mvp, pObj); // obj -> clip space zLinear = mul(mv, pObj).z; // z in eye space } // Pass 1 (compute linear z): fragment program. void main (half zLinear : TEXCOORD0, out half4 z : COLOR) { z = zLinear; } </pre>	<pre> // Pass 4 (make shadows): vertex program. void main (float4 pObj : POSITION, out float4 pClip : POSITION, out float4 projRect : TEXCOORD0, uniform float4x4 mvp, uniform float4x4 mv, uniform float4x4 lightClip) { pClip = mul(mvp, pObj); // obj -> clip space float4 pEye = mul(mv, pObj); // obj -> eye space // Compute RECT projective texture coordinates. // lightClip maps camera's eye space to light's // clip space, and multiplies x and y by shadow // map resolution to yield texRECT coordinates. float4 projCoords = mul(lightClip, pEye); } // Pass 4 (make shadows): fragment program. void main (float4 projRect : TEXCOORD0, out half4 color : COLOR, uniform samplerRECT shadowMap : TEXUNIT0, uniform samplerRECT smDepthMap : TEXUNIT1, uniform samplerRECT smAlphaMap : TEXUNIT2, uniform sampler1D sinFalloff : TEXUNIT3) { // Use hardware shadow map depth comparison and // 2x2 filtering support (ARB_shadow). All // textures set to GL_LINEAR filter mode. fixed u = texRECTproj(shadowMap, projRect).x; fixed p = texRECTproj(smDepthMap, projRect).x; fixed a = texRECTproj(smAlphaMap, projRect).x; // Are we in shadow? fixed v = 1; // assume not in shadow if (p < 1) v = a; // if in penumbra, use alpha if (u < 1) v = 0; // if in umbra, all shadow // Add sinusoidal falloff. v = tex1D(sinFalloff, v); // ... optional shading calculations go here ... color = v; // in practice, modulate shading by v } </pre>																		
<pre> // Pass 3 (remap alpha): vertex program. void main (float4 pObj : POSITION, out float4 pClip : POSITION, out float zLinear : TEXCOORD0, uniform float4x4 mvp, uniform float4x4 mv) { pClip = mul(mvp, pObj); // obj -> clip space zLinear = mul(mv, pObj).z; // z in eye space } // Pass 3 (smoothie EDGE): fragment program. void main (half zLinear : TEXCOORD0, float4 wpos : WPOS, out half4 aRemap : COLOR, uniform half3 v, // silhouette vertex uniform half3 n, // silhouette normal uniform half invT, // 1/t (smoothie size) uniform samplerRECT shadowMap) { // Compute linear alpha in screen space. half alpha = dot(wpos.xyz - v, n) * invT; // Compute ratio f = b/r to remap alpha values. // If b/r > 1, smoothie pixel is occluded. // Otherwise, remap alpha and clamp to [0,1]. half f = zLinear / texRECT(shadowMap, wpos.xy).x; aRemap = (f > 1) ? 1 : saturate(alpha / (1 - f)); } // Pass 3 (smoothie CORNER): fragment program. void main (half zLinear : TEXCOORD0, float4 wpos : WPOS, out half4 aRemap : COLOR, uniform half3 v, // silhouette vertex uniform half invT, // 1/t (smoothie size) uniform samplerRECT shadowMap) { // Compute linear alpha in screen space and remap. half alpha = saturate(length(wpos.xyz - v) * invT); half f = zLinear / texRECT(shadowMap, wpos.xy).x; aRemap = (f > 1) ? 1 : saturate(alpha / (1 - f)); } </pre>	<table border="1"> <thead> <tr> <th>Rendering Pass</th> <th>Vertex</th> <th>Fragment</th> </tr> </thead> <tbody> <tr> <td>Pass 1 (shadow map)</td> <td>5</td> <td>1</td> </tr> <tr> <td>Pass 2 (smoothie depth)</td> <td>n/a</td> <td>n/a</td> </tr> <tr> <td>Pass 3 (smoothie alpha)</td> <td>5</td> <td>11 (edges)</td> </tr> <tr> <td></td> <td>5</td> <td>13 (corners)</td> </tr> <tr> <td>Pass 4 (make shadows)</td> <td>12</td> <td>10</td> </tr> </tbody> </table>	Rendering Pass	Vertex	Fragment	Pass 1 (shadow map)	5	1	Pass 2 (smoothie depth)	n/a	n/a	Pass 3 (smoothie alpha)	5	11 (edges)		5	13 (corners)	Pass 4 (make shadows)	12	10
Rendering Pass	Vertex	Fragment																	
Pass 1 (shadow map)	5	1																	
Pass 2 (smoothie depth)	n/a	n/a																	
Pass 3 (smoothie alpha)	5	11 (edges)																	
	5	13 (corners)																	
Pass 4 (make shadows)	12	10																	

Figure 4-7: Cg vertex and fragment shader code. We compiled the shaders and optimized the resulting assembly code by hand. The table at the bottom-right shows the number of assembly instructions for each rendering pass after optimization.

Figure 4-1d shows a simple scene with 10,000 triangles rendered using our method at 39 fps. The spotlight casts shadows with soft edges onto the ground plane.

Figure 4-8 compares the quality and performance of shadows generated using different methods. The images display a close-up of the shadow cast by the cylinder in Figure 4-1d. The first column of images is rendered using an ordinary shadow map; the second column is rendered using a version of percentage closer filtering [RSC87]


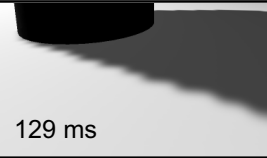
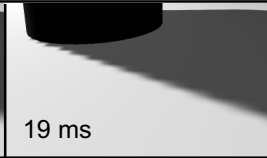
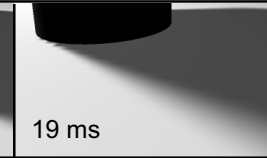


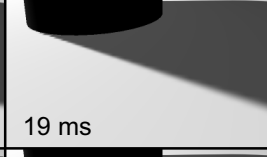
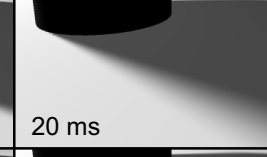
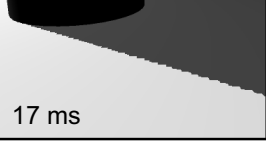

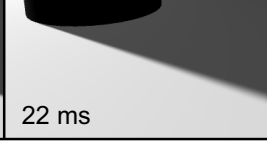
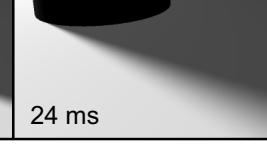
	shadow map (1 sample)	bicubic (16 samples)	smoothie (size = 0.2)	smoothie (size = 0.8)
256 x 256	 16 ms	 129 ms	 19 ms	 19 ms
512 x 512	 16 ms	 132 ms	 19 ms	 20 ms
1024 x 1024	 17 ms	 142 ms	 22 ms	 24 ms

Figure 4-8: Shadow edge comparison using different methods and shadow map resolutions. The shadow is cast by the cylinder from Figure 4-1. The total rendering time per frame is shown at the lower-left of each image.

with a bicubic filter; the last two columns are rendered using our method. The rows correspond to different shadow map resolutions.

These images illustrate the advantages of our approach. First, whereas aliasing is evident when using the other methods, the smoothie algorithm helps to hide the aliasing artifacts, even at low shadow map resolutions. Second, the smoothies give rise to soft shadow edges that resemble penumbrae; notice that the penumbra is larger in regions farther away from the cylinder. We can indirectly simulate a larger area light source by making the smoothies larger, as shown in the fourth column. Finally, the rendering times given in Figure 4-8 indicate that the smoothie algorithm is more expensive than the regular shadow map, but still much faster than using the bicubic filter.

Figure 4-9 compares the shadows generated using our method to the geometrically-correct shadows computed using a Monte Carlo ray tracer. These images show that our method works best when simulating area light sources that are small relative to the blockers (top row). Recall that the smoothies only extend *outwards* from the blockers' silhouette edges. Like the work of Haines [Hai01] and Parker et al. [PSS98], this construction produces shadow umbrae that do not shrink as the size of the light

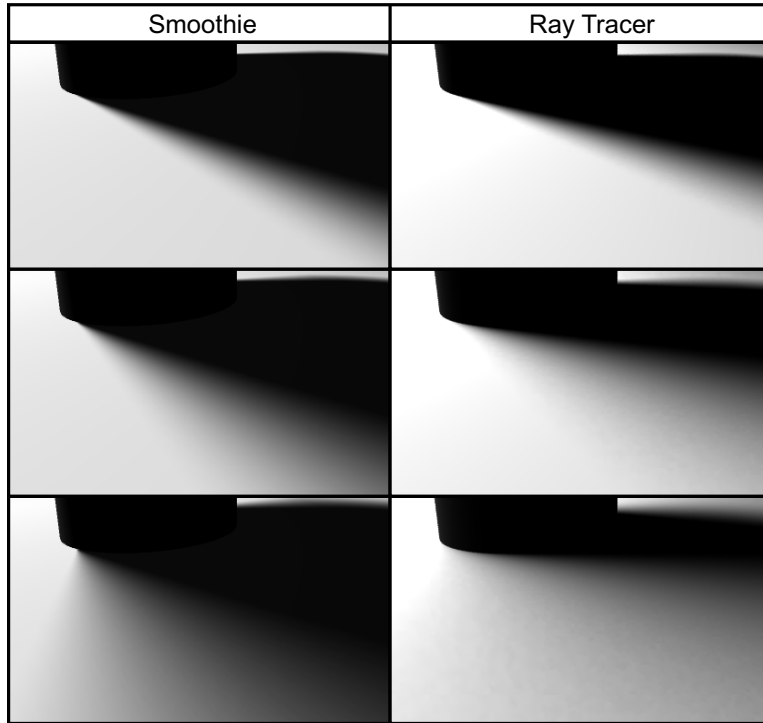


Figure 4-9: Comparison of soft shadow edges. The images in the left column are rendered using the smoothie algorithm with a buffer resolution of 1024×1024 . The images in the right column are rendered using a Monte Carlo ray tracer and show the geometrically-correct soft shadows. Each successive row shows the effect of simulating a larger area light source.

source increases. Furthermore, the construction relies on a single set of object-space silhouette edges computed with respect to a point light source. This approximation becomes worse as the size of the light source increases, since different points on the light source correspond to different sets of silhouette edges. The resulting differences in visual quality are noticeable at the contact point between the cylinder and the floor in the bottom row of images.

Figure 4-10 illustrates the importance of remapping the alpha values when a shadow falls on multiple receivers. In this scene, two boxes are arranged above a ground plane, and a spotlight illuminates the objects from above. The left and middle images show the smoothie buffer's alpha values in two different cases. In the left image, alpha is linearly interpolated from the silhouette edge of each smoothie to the opposite edge. This approach does not take into account the depth discontinuity

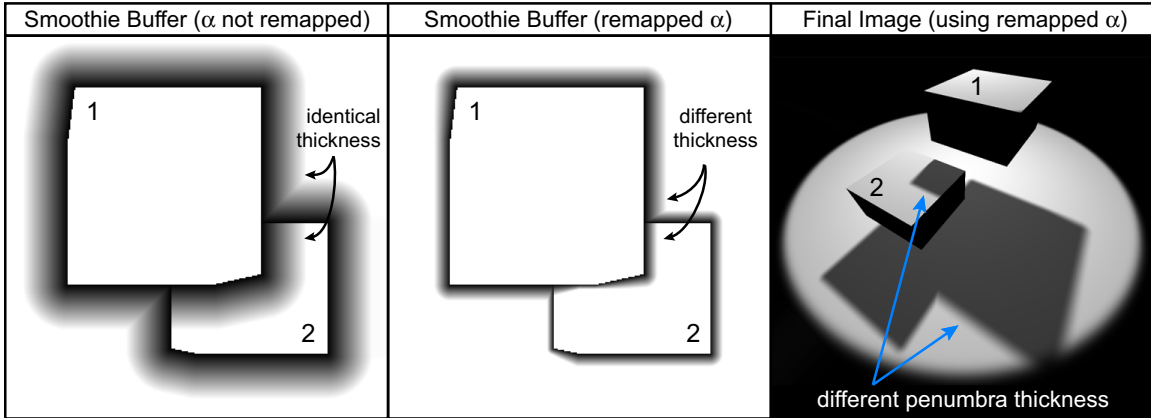


Figure 4-10: Scene configuration with multiple blockers and receivers. Remapping the alpha values (shown in the left and middle images) causes the penumbra due to a single blocker to vary in thickness across multiple receivers. As a result, box 1 casts a shadow onto box 2 and the ground plane, but the shadow edges on the ground plane are softer.

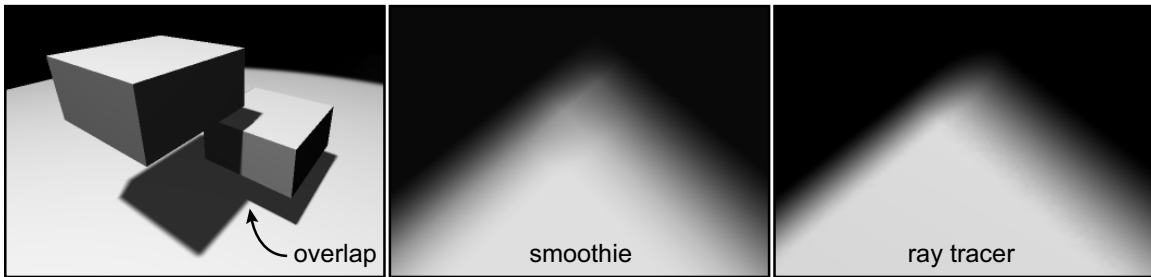


Figure 4-11: Comparison of overlapping penumbrae. The left image shows how shadows may overlap due to multiple blockers. The next two images show a close-up of the region indicated by the arrow. The middle image is rendered using the smoothie algorithm. When multiple smoothies overlap, their alpha values are composited using minimum blending. The right image is generated using a Monte Carlo ray tracer and shows the geometrically-correct soft shadow that results from multiple blockers.

between box 2 and the ground plane in the region indicated by arrows. In the middle image, the alpha values have been remapped at each pixel using Equation 4.1. The remapping has the desired effect (shown in the right image): the shadow cast by box 1 has a smaller, harder edge on box 2 and a thicker, softer edge on the ground plane.

Figure 4-11 shows a common case of two blockers overlapping in the screen space of the light source. Recall that when multiple smoothies overlap, we use minimum blending to composite their alpha values. Although this operation is not geometrically

accurate, the middle image shows that the resulting overlapping penumbrae appear smooth and reasonably similar to the image generated by the ray tracer.

Figure 4-12 shows four different scenes rendered using the smoothie algorithm. The models in these scenes vary in geometric complexity, from 5000 triangles for the boat to over 50,000 triangles for the motorbike. The images demonstrate the ability of our algorithm to handle both complex blockers and complex receivers. For instance, the right image in the second row shows the shadows cast by the flamingo’s head onto its neck, and by its neck onto its back. Similarly, the right image in the last row shows a difficult case where the spokes of the motorbike wheel cast shadows onto themselves and onto the ground plane. All of these scenes run at interactive framerates; even the intricate motorbike renders at 18 fps using buffer resolutions of 1024×1024 .

Table 4.1 summarizes the rendering performance for the images shown in Figures 4-1d and 4-12. For most scenes, the final rendering pass accounts for over 90% of the total running time. An exception is the motorbike scene, which contains many more overlapping silhouette edges than the other scenes; about 30% of its total rendering time is spent computing the smoothies’ alpha values. Even so, increasing the size of smoothies to simulate larger area light sources incurs little overhead. For the scenes shown in Figure 4-12, we found that enlarging the smoothies from $t = 0.02$ to $t = 0.2$ increases the rendering times by about 15%.

Scene	Boat (Fig. 4-12)	Primitives (Fig. 4-1d)	Flamingo (Fig. 4-12)	Elephant (Fig. 4-12)	Motorbike (Fig. 4-12)
Geometry					
Triangles	5664	9424	26,370	39,290	50,648
Edges	8488	14,124	39,460	59,039	76,287
Silhouette Edges	1677	150	1554	2861	10,600
Rendering times					
256×256	20 ms	20 ms	21 ms	26 ms	37 ms
512×512	22 ms	22 ms	22 ms	27 ms	50 ms
1024×1024	26 ms	26 ms	25 ms	32 ms	53 ms

Table 4.1: Performance measurements for each scene. Timings per frame are given for different buffer resolutions.

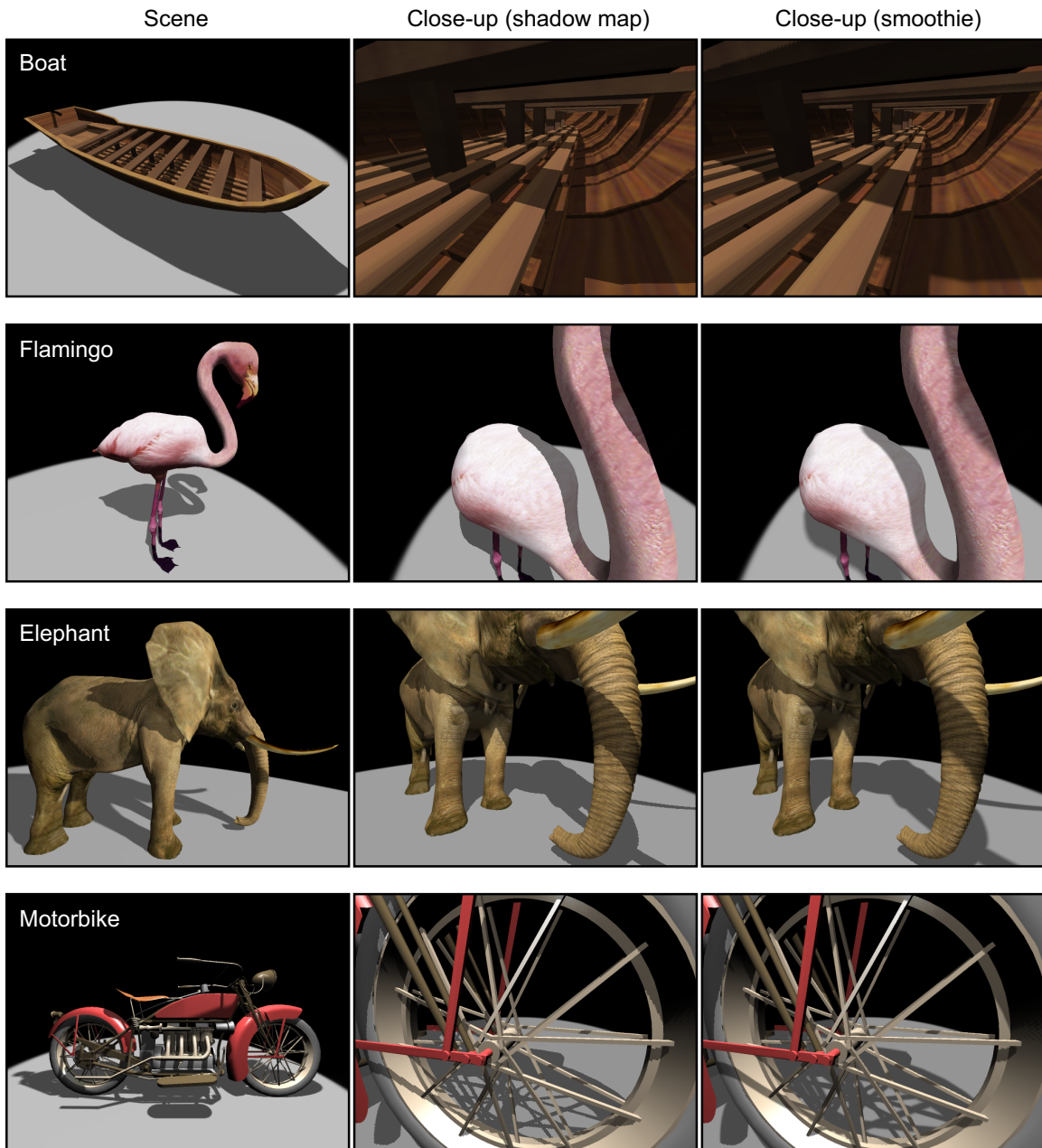


Figure 4-12: Scenes rendered using the smoothie algorithm, arranged in order of increasing geometric complexity. These images show the interaction between complex blockers and receivers. For instance, the elephant’s tusk casts a shadow onto its trunk, and the spokes of the motorbike wheel cast shadows onto themselves. The middle and right columns compare the shadow quality produced by ordinary shadow maps and our method, respectively.

4.5 Discussion

The above results illustrate the advantages of combining smoothies with shadow maps. In summary, constructing smoothies in object space with interpolated alpha values ensures smooth shadow edges, but performing the visibility calculations in image space limits the number of depth samples we need to examine. Thus we can achieve interactive framerates for detailed scenes.

The combination of object-space and image-space calculations is an important difference between our work and the penumbra wedge algorithms [AMA02, AAM03, ADMAM03]. Since penumbra wedges are based on shadow volumes, wedges are rendered from the observer's viewpoint. Thus rasterized wedges tend to occupy substantial screen area, especially in heavily shadowed scenes. In contrast, smoothies are rendered from the light's viewpoint. They occupy relatively little screen area (see Figure 4-1b) and therefore are cheaper to render.

The geometry cost of the smoothie algorithm lies somewhere between the cost of shadow maps and shadow volumes. Shadow maps require drawing the scene twice; our method adds to this cost by also drawing the smoothies twice, once to store the depth values and once to store the alpha values. However, only one smoothie is needed per silhouette edge, and for complex models the number of silhouette edges is small relative to the number of triangle faces. The shadow volume algorithm draws the same number of shadow polygons, but it also requires drawing all blocker triangles once more to cap the shadow volumes properly. Compared to shadow volumes, penumbra wedges require additional polygons to represent each wedge.

The size of the smoothies is defined in terms of an image-space user parameter t (see Figure 4-2). Fortunately, t is an intuitive parameter because it directly affects the size of the generated penumbra. Applications such as 3D game engines should choose t for a given environment depending on the spatial arrangement of objects and the desired shadow softness. We have found that for scenes of size $10 \times 10 \times 10$ units, t values in the range of 0.02 to 0.2 (measured in normalized screen space coordinates) give a reasonable approximation for small area lights and yield good image quality.

Our work attempts to combine the best qualities of existing shadow techniques, but it also inherits some of their limitations. For example, since our method uses shadow maps, we require the light source to be a directional light or a spotlight; additional buffers and rendering passes are needed to support omnidirectional lights. In contrast, shadow volumes and the penumbra wedge algorithms automatically handle omnidirectional lights.

A second limitation is that our method, like the shadow volume algorithm, relies on finding the blockers' silhouette edges in object space. Thus we assume that blockers are represented as closed triangle meshes to simplify the computations. In contrast, ordinary shadow maps trivially support any geometric representation that can be rendered into a depth buffer. It would be nice to identify silhouette edges and construct smoothies in image space instead of in object space. One approach might be to find the blockers' silhouettes using McCool's edge detection algorithm [McC00]. Constructing the smoothies directly from these detected silhouettes requires a feedback path from the output of the edge detection pixel shader to a vertex array. This feedback path has recently become available in graphics hardware, but we have not yet experimented with it.

Another limitation inherited from shadow maps is the use of discrete depth buffer samples, which leads to aliasing at the shadow edges. Although we have shown that smoothies can hide aliasing artifacts, this approach does not work as well when the smoothies are small or when the light source is far away (see Figure 4-8, top row, third column). The reason is that less shadow map resolution is available to capture the smooth variation in alpha values. This problem can be addressed by combining our algorithm with a perspective shadow map [SD02, Koz04], which effectively provides more resolution to depth samples located closer to the viewer.

In the previous section, we saw that the shadow umbrae computed by the smoothie algorithm do not shrink as the size of the light source increases. It seems logical to extend a smoothie not only outwards from a silhouette edge, but also *inwards* to model the inner penumbra. This approach, however, leads to "light leaks" when two blockers touch. In general, rendering the inner penumbra accurately requires examining the

interactions between multiple blockers. We have not yet found a robust and consistent way to handle these interactions.

4.5.1 Comparison to Penumbra Maps

The penumbra map algorithm [WH03] was developed independently by Wyman and Hansen and published in the same forum as our algorithm. Both techniques approximate soft shadows by attaching geometric primitives to the objects' silhouette edges and rendering alpha values into a buffer. There are two important differences, however.

The first difference lies in the geometric primitives themselves. The penumbra map algorithm builds cones at the silhouette vertices and draws sheets to connect the cones. These primitives must be tessellated finely enough to ensure smooth shadow corners and to avoid Gouraud shading artifacts. Thus each silhouette vertex introduces multiple cone vertices, and each silhouette edge introduces at least four new vertices.

In contrast, our method constructs smoothie edges as rectangles in screen space, which avoids Gouraud shading artifacts. Instead of drawing cones at the silhouette vertices to create round soft shadow corners, we draw quadrilaterals with sharp corners and rely on pixel shaders to interpolate the alpha values. Since we do not tessellate the smoothies, each silhouette vertex and edge introduces exactly four new vertices. Although more shading is required for the smoothie corners, the corner geometry occupies little screen area. Our approach, however, requires a special case for vertices with more than two adjacent silhouette edges, as described in Section 4.2.

The second difference is that the penumbra map algorithm stores the depth values of the blockers but not the depth values of the cones and sheets. In contrast, our method keeps track of the depth values of both the blockers and the smoothies; thus we require an additional depth comparison in the final rendering pass. The smoothie's depth value is useful, however, in cases where some objects in the scene are specified as shadow receivers but not shadow blockers. Applications such as 3D video games often restrict the number of blockers in a scene for performance reasons; for instance,

blockers may be limited to characters and other dynamic objects. The second depth value ensures that soft shadow edges are cast properly on all receivers, including non-blockers. Fortunately, the penumbra map algorithm can be extended easily to store depth values of the cones and sheets. Likewise, the smoothie algorithm can be simplified by omitting the smoothies' depth values.

4.6 Conclusions

We have described the smoothie algorithm, a simple extension to shadow maps for rendering soft shadows. Our experiments show that while the soft shadow edges are not geometrically accurate, they resemble penumbrae and help to hide aliasing artifacts. The algorithm is also efficient and can be implemented using programmable graphics hardware to achieve real-time performance.

Current research in real-time shadow algorithms is closely tied to programmable features of graphics hardware. In particular, many multipass algorithms exploit the hardware's ability to compute and store arbitrary data values per-pixel at high precision. For example, recent work has shown how to simulate subsurface scattering effects by keeping additional data in a shadow map [DS03]. Similarly, it may be possible to extend our work to generate more accurate shadow penumbrae by storing extra silhouette data.

Recently-announced graphics architectures such as the NVIDIA GeForce 6 [NVI04] provide per-vertex texture accesses, floating-point blending, and the ability to render directly to vertex attribute arrays. We expect that new hardware features such as these will continue to facilitate the design of real-time shadow algorithms in the future.

Chapter 5

Conclusions

Many shadow algorithms have been proposed over the past three decades for both real-time and offline rendering applications. In this thesis, we have focused on methods optimized for real-time shadows, but our hybrid and smoothie algorithms are both well-suited for offline rendering systems. On the other hand, a surprisingly high number of papers concerning real-time shadows have been published in just the last three years. One reason for this emphasis is that commodity graphics hardware has recently become programmable in both the vertex and fragment stages; this flexibility allows more sophisticated algorithms to be mapped to the hardware. The two algorithms that we have presented in this thesis are examples of techniques that take advantage of programmable graphics hardware.

With the recent developments in graphics hardware, shadows are rapidly becoming a key feature of hardware-accelerated 3D game engines. It is interesting to consider the shadow algorithms that developers currently choose for these engines, especially in light of the design criteria covered in Chapter 1. Many engines that draw only hard shadows, such as *Doom 3*, *Halo 2*, and *Neverwinter Nights*, use the shadow volume algorithm because of its robustness and artifact-free image quality. This choice of shadow algorithm underscores the importance of robustness when attempting to apply a shadow algorithm to real-world applications with dynamic environments. As we have discussed throughout this thesis, however, shadow volumes do not scale well to complex scenes. Indeed, John Carmack of id Software has stated that the *Doom*

3 engine spends half of its rendering time drawing shadows! The hybrid algorithm presented in Chapter 3 is designed to address these scalability problems while maintaining high image quality.

Whereas many game engines are beginning to support hard shadow algorithms such as shadow volumes, no current engine implements a general solution for soft shadows. Games such as *Max Payne 2* and *Half-Life 2* use blurred projective textures to simulate soft shadows. This technique is fast and simple and it masks aliasing artifacts, but it is not a general solution because it does not support self-shadowing. Instead, this technique is mostly used to draw shadows cast by characters onto the environment.

Our purpose in discussing the shadow techniques used by various game engines is to identify some areas for future work. For instance, how can we solve the under-sampling and robustness problems of shadow maps while preserving the generality of shadow maps? Although many solutions have been proposed in the past few years, some do not work for certain scene configurations, and others are limited to a certain class of geometric models. Second, how can we further optimize the shadow volume algorithm to make it practical for large, complex scenes? Third, can we develop a soft shadow algorithm that is efficient, artifact-free, and robust? Some of the current solutions, such as ours, extend the basic shadow map and shadow volume approaches, and therefore they also inherit the corresponding limitations. Other solutions, such as those based on precomputed radiance transfer, already achieve both high quality and high performance; the challenge in this case is to extend these techniques to support dynamic scenes. In summary, even though dozens of papers on real-time shadow computation have been published over the past three years, there are still many interesting and challenging problems to be solved.

We believe that three additional hardware features will help researchers find solutions to these problems. The first of these is data-dependent flow control (e.g. branching and looping) in fragment programs; data-dependent looping is helpful for many tasks, including efficient filtering with spatially-varying kernel sizes and local search algorithms. The second feature is the upcoming PCI Express architecture,

which will enable significantly faster readbacks from the graphics hardware to the host processor; fast readbacks will facilitate hybrid CPU/GPU algorithms. The third feature that will improve shadow rendering performance significantly is the computation mask, which we have discussed at length in Chapter 3. We have shown that a user-programmable mask is useful for shadow rendering, but it applies much more generally to multipass rendering algorithms. We hope that computation masks will be supported natively in future generations of graphics hardware.

Bibliography

- [AAM03] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics*, 22(3):511–520, 2003.
- [AAMar] Timo Aila and Tomas Akenine-Möller. A hierarchical shadow volume algorithm. In *Proceedings of the ACM SIGGRAPH / Eurographics Workshop on Graphics Hardware*. ACM Press, 2004 (to appear).
- [ADMAM03] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An optimized soft shadow volume algorithm with real-time performance. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 33–40. Eurographics Association, 2003.
- [AMA02] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 309–318, 2002.
- [AMH02] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering (2nd edition)*, pages 248–276. A. K. Peters Ltd., 2002.
- [AMN03] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics (TOG)*, 22(3):792–800, 2003.

- [ARHM00] Maneesh Agrawala, Ravi Ramamoorthi, Alan Heirich, and Laurent Moll. Efficient image-based methods for rendering soft shadows. In *Proceedings of ACM SIGGRAPH*, pages 375–384. ACM Press, 2000.
- [Ber86] Philippe Bergeron. A general version of Crow’s shadow volumes. In *IEEE Computer Graphics and Applications*, pages 17–28, September 1986.
- [BRW89] D. R. Baum, H. E. Rushmeire, and J. M. Winget. Improving radiosity solutions through the use of analytically determined form-factors. In *Proceedings of ACM SIGGRAPH*, pages 325–334. ACM Press, 1989.
- [BS02] Stefan Brabec and Hans-Peter Seidel. Single sample soft shadows using depth maps. In *Proceedings of Graphics Interface*, pages 219–228, May 2002.
- [BWG03] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. *ACM Transactions on Graphics (TOG)*, 22(3):631–640, 2003.
- [Car00] John Carmack. Private email, May 23, 2000.
<http://developer.nvidia.com/attach/3413>.
- [CD03] Eric Chan and Frédo Durand. Rendering fake soft shadows with smoothies. In *Proceedings of the Eurographics Symposium on Rendering*, pages 208–218. Eurographics Association, 2003.
- [CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of ACM SIGGRAPH*, pages 137–145, 1984.
- [Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of ACM SIGGRAPH*, pages 242–248. ACM Press, 1977.
- [DF94] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In *Proceedings of ACM SIGGRAPH*, pages 223–230. ACM Press, 1994.

- [Die96] Paul Diefenbach. Multi-pass pipeline rendering: Interaction and realism through hardware provisions. Ph. D. thesis MS-CIS-96-26, University of Pennsylvania, 1996.
- [DS03] Carsten Dachsbacher and Marc Stamminger. Translucent shadow maps. In *Proceedings of the Eurographics Symposium On Rendering 2003*, pages 197–201. Eurographics Association, 2003.
- [EK02] Cass Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering, 2002.
http://developer.nvidia.com/object/robust_shadow_volumes.html.
- [EK03] Cass Everitt and Mark J. Kilgard. Optimized stencil shadow volumes, 2003. http://developer.nvidia.com/docs/IO/8230/GDC2003_ShadowVolumes.pdf.
- [FFBG01] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Proceedings of ACM SIGGRAPH*, pages 387–390. ACM Press, 2001.
- [GCHHar] Simon Gibson, Jon Cook, Toby Howard, and Roger Hubbard. Rapid shadow generation in real-world lighting environments. In *Proceedings of the Eurographics Symposium on Rendering, 2003* (to appear).
- [GLY+03] Naga K. Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha. Interactive shadow generation in complex environments. *ACM Transactions on Graphics (TOG)*, 22(3):501–510, 2003.
- [Hai01] Eric Haines. Soft planar shadows using plateaus. In *Journal of Graphics Tools*, vol. 6, no. 1, pages 19–27, 2001.
- [HBS00] Wolfgang Heidrich, Stefan Brabec, and Hans-Peter Seidel. Soft shadow maps for linear lights. In *Proceedings of the 11th Eurographics Workshop on Rendering*, pages 269–280, 2000.

- [Hei91] Tim Heidmann. Real shadows, real time. In *Iris Universe, No. 18*, pages 23–31. Silicon Graphics Inc., November 1991.
<http://developer.nvidia.com/attach/3414>.
- [HH97] Paul Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, Computer Science Department, Carnegie Mellon University, 1997.
- [IFH⁺03] Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. A developer’s guide to silhouette algorithms for polygonal models. *IEEE Computer Graphics and Applications*, special issue on NPR, 2003.
- [Kil01] Mark J. Kilgard. Shadow mapping with today’s OpenGL hardware, 2001.
http://developer.nvidia.com/docs/IO/1446/ATT/ShadowMaps_CEDEC_E.pdf.
- [Kil02] Mark Kilgard. NVIDIA OpenGL extension specifications for the CineFX architecture (NV30), 2002.
<http://developer.nvidia.com/docs/IO/1174/ATT/nv30specs.pdf>.
- [Koz04] Simon Kozlov. Perspective shadow maps: Care and feeding. In *GPU Gems*, pages 217–244. Addison-Wesley, 2004.
- [Len02] Eric Lengyel. The mechanics of robust stencil shadows. In *Gamasutra*, October 11, 2002.
http://www.gamasutra.com/features/20021011/lengyel_01.htm.
- [LPFH01] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 227–232. ACM Press, 2001.
- [LV00] Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of ACM SIGGRAPH*, pages 385–392. ACM Press, 2000.

- [LWGMar] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. CC shadow volumes. In *Proceedings of the Eurographics Symposium on Rendering, 2004* (to appear).
- [McC00] Michael D. McCool. Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics (TOG)*, 19(1):1–26, 2000.
- [MGAKar] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Proceedings of ACM SIGGRAPH*. ACM Press, 2003 (to appear).
- [MHE⁺03] Morgan McGuire, John F. Hughes, Kevin Egan, Mark Kilgard, and Cass Everitt. Fast, practical and robust shadows. Technical Report CS03-19, Brown University, October 27, 2003.
- [Mor00] Steve Morein. ATI Radeon – HyperZ technology, August 22, 2000. http://www.ibiblio.org/hwsw/previous/www_2000/presentations/ATIHOT3D.pdf.
- [NRH03] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Transactions on Graphics*, 22(3):376–381, 2003.
- [NVI02] NVIDIA Corporation. GeForce FX, 2002. http://www.nvidia.com/view.asp?PAGE=fx_desktop.
- [NVI04] NVIDIA Corporation. GeForce 6800 product web site, 2004. http://www.nvidia.com/page/geforce_6800.html.
- [Ope02a] OpenGL Architectural Review Board. `EXT_depth_bounds_test` OpenGL extension specification, 2002. http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_depth_bounds_test.txt.

- [Ope02b] OpenGL Architectural Review Board. `NV_occlusion_query` OpenGL extension specification, 2002.
http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_occlusion_query_test.txt.
- [PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712. ACM Press, 2002.
- [PDC⁺03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.
- [PSS98] Steve Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, 1998.
- [Pur04] Timothy J. Purcell. Ray tracing on a stream processor. Ph. D. thesis, Stanford University, 2004.
- [Ras01] Ramesh Raskar. Hardware support for non-photorealistic rendering. In *Proceedings of the ACM SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 41–47. ACM Press, 2001.
- [RH01] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of ACM SIGGRAPH*, pages 497–500. ACM Press, 2001.
- [RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of ACM SIGGRAPH*, pages 283–291. ACM Press, 1987.

- [SCH03] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. *ACM Transactions on Graphics (TOG)*, 22(3):521–526, 2003.
- [SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proceedings of ACM SIGGRAPH*, pages 557–562. ACM Press, 2002.
- [SGG⁺00] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *Proceedings of ACM SIGGRAPH*, pages 327–334. ACM Press, 2000.
- [SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of ACM SIGGRAPH*, pages 527–536. ACM Press, 2002.
- [SKvW⁺92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *Proceedings of ACM SIGGRAPH*, pages 249–252. ACM Press, 1992.
- [SS98] Cyril Soler and François X. Sillion. Fast calculation of soft shadow textures using convolution. In *Proceedings of ACM SIGGRAPH*, pages 321–332. ACM Press, 1998.
- [WFG92] Leonard C. Wanger, James A. Ferwerda, and Donald P. Greenberg. Perceiving spatial relationships in computer-generated images. In *IEEE Computer Graphics and Applications*, vol. 12, no. 3, pages 44–58, 1992.
- [WH03] Chris Wyman and Charles Hansen. Penumbra maps: approximate soft shadows in real-time. In *Proceedings of the Eurographics Symposium on Rendering 2003*, pages 202–207. Eurographics Association, 2003.
- [Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of ACM SIGGRAPH*, pages 270–274. ACM Press, 1978.

- [WPF90] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, November 1990.