Eurographics Symposium on Rendering 2003
Per Christensen and Daniel Cohen-Or (Editors)

Rendering Fake Soft Shadows with Smoothies

Eric Chan and Frédo Durand

Laboratory for Computer Science
Massachusetts Institute of Technology†

Abstract

We present a new method for real-time rendering of shadows in dynamic scenes. 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. 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 present results from several example scenes rendered in real-time.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors, I.3.3 [Computer Graphics]: Bitmap and framebuffer operations, I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

Keywords: soft shadow algorithms, projective texture mapping, programmable graphics hardware

1. Introduction

Shadows are important to image synthesis because they add realism and help the viewer identify spatial relationships[33]. Many of the 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. Existing methods are usually limited by sampling artifacts or scalability issues.

This paper describes a new real-time algorithm that combines shadow maps with geometric primitives that we call *smoothies* to render fake soft shadows (see Figure 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. automatically handles dynamic scenes.

We present results from several example scenes rendered by our algorithm in real-time.

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 nor orientation of the light source.

2. Related Work

Our method builds on the ideas of several existing shadow algorithms. We focus our discussion below on the most relevant methods and refer the reader to Woo et al.'s paper[35] for a broader survey of shadow algorithms.

A popular means for generating shadows is the shadow map, introduced by Williams in 1978[34]. The algorithm first renders a depth map of the scene from the light's viewpoint; the depth map is then used to determine which samples in the final image are visible to the light. Shadow maps are efficient and are accelerated in modern graphics hardware, but they are prone to sampling artifacts such as aliasing. Recent work attempts to reduce aliasing by increasing the effective shadow map resolution[12, 32].

Shadow maps have been extended to support antialiased shadows and soft shadows through a combination of filtering, stochastic sampling, and image warping techniques[29, 21, 1, 16, 13]. However, these methods require ex-

† email: {ericchan|fredo}@graphics.lcs.mit.edu

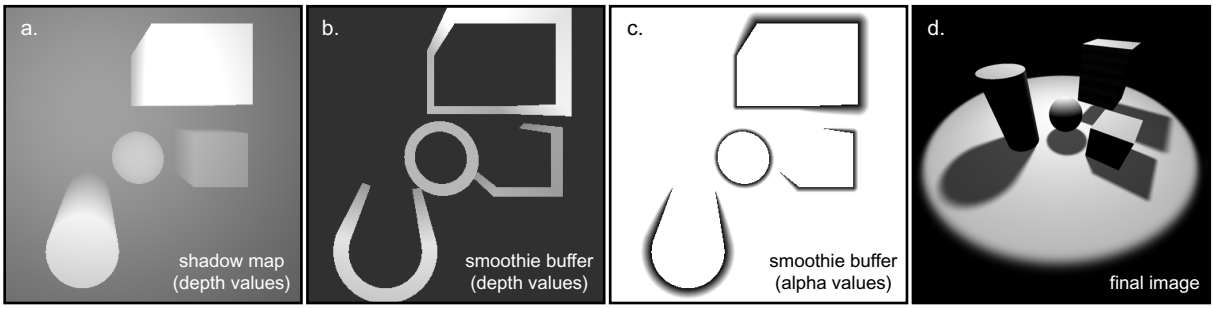

Figure 1: *Smoothie algorithm overview. (a) We first render a shadow map from the light's viewpoint. Next, we construct geometric primitives that we call* smoothies *at the objects' silhouettes. (b) We render the smoothies' depth values into the smoothie buffer. (c) For each pixel in the smoothie buffer, 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.*

amining 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[31] 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.[17] described how shadow maps can support linear light sources using only two samples per light. Brabec and Seidel[6] 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.[27] 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 penumbrae are supported. In other words, the computed shadow umbrae do not shrink properly as the size of the area light source increases.

Recently, researchers have pushed this idea further with an emphasis on real-time rendering of soft shadows using graphics hardware. For instance, Akenine-Möller and Assarsson[3] developed a soft shadow algorithm based on shadow volumes[8]. They replace each shadow volume polygon with a *penumbra wedge*, a set of polygons that encloses the penumbra region for a given silhouette edge. The method achieves visual smoothness by linearly interpolating light intensity within the wedge, and the wedges are constructed in a manner that supports inner shadow penumbrae.

Assarsson and Akenine-Möller[4] subsequently generalized their penumbra wedge algorithm. They describe an improved wedge construction technique that increases robustness and can handle multiple wedges independently of each other. 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, shadow volumes are expensive, even when accelerated using a hardware stencil buffer. The reason is that each rendered pixel of every shadow volume polygon requires a stencil buffer update, resulting in significant overdraw and large fillrate requirements. Optimizations using per-object scissor rectangles and depth bounds clipping can reduce the fillrate consumption[11], but in many cases shadow volumes are too expensive for complex scenes. 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[14] describes a method for rendering a hard drop shadow into a texture map and approximating a penumbra along the shadow silhou-

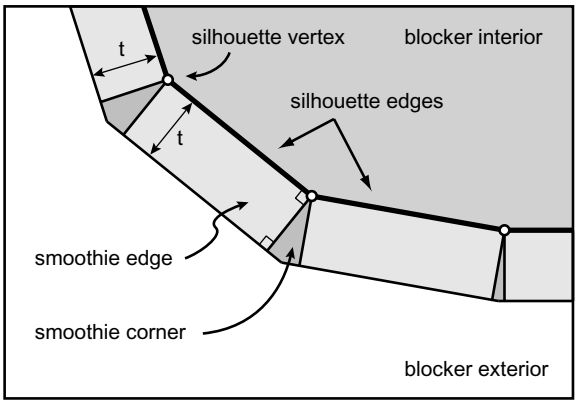

Figure 2: *Smoothie construction. A smoothie edge is obtained by extending a blocker's silhouette edge outwards to form a rectangle in screen space. A smoothie corner connects adjacent smoothie edges.*

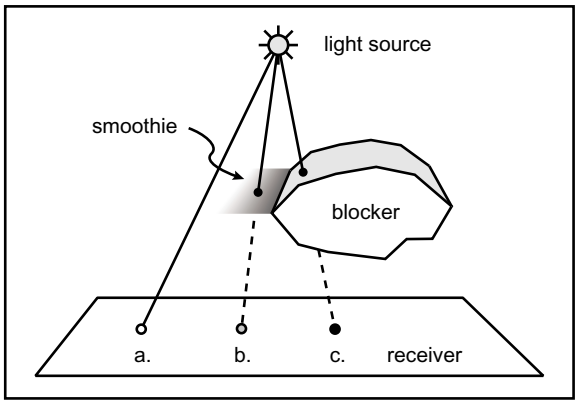

Figure 3: *Three possible scenarios: an image sample is either (a) illuminated, (b) partially in shadow, or (c) completely in shadow.*

ettes. 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.[27], this construction only grows penumbra regions outward from the umbra and does not support inner shadow penumbrae.

Recently, Wyman and Hansen[36] independently developed a soft shadow algorithm similar to ours. 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 compare their approach to ours in Section 6.1.

2.1. Overview

An overview of our approach is shown in Figure 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 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 3). A limitation of this approach is that computed shadow umbrae do not diminish as the area of the light source increases.

Our proposed algorithm is inspired by the different classes of techniques discussed above. 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 at al.[27] and the convolution approach of Soler and Sillion[31]. 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.[20], who showed how texture-mapped fins can improve the quality of fur rendering at the silhouettes.

We make the following assumptions in our approach:

1. Blockers are opaque. We do not handle shadows due to semi-transparent surfaces.
2. Blockers are represented as closed triangle meshes. This representation simplifies the task of finding object-space silhouette edges.

In addition, our method does not take into account the shape and orientation of light sources, so we assume that light sources are roughly spherical.

3. 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 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[18, 28]. We use a simple brute-force algorithm, looping over all the edges and performing the above test for each edge. For static models, Sander at al.[30] describe a more efficient algorithm using hierarchical search trees. Hartner et al.[15] compare the performance of a number of object-space silhouette extraction algorithms.

3. Construct smoothies. We construct a smoothie edge

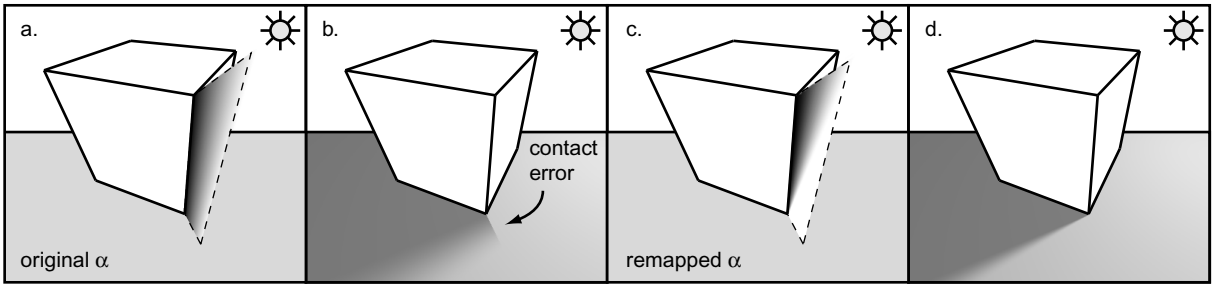

Figure 4: *Computing smoothie alpha values. Diagram (a) shows a smoothie with the original, linearly interpolated alpha. (b) The resulting soft shadow edge has a fixed thickness and also has an obvious discontinuity at the contact point between the box and the floor. In diagram (c) we have remapped the alpha values as described in Equation 1. (d) The remapping fixes the contact problem and also creates a soft shadow edge that more closely resembles a penumbra.*

for each silhouette edge and a smoothie corner for each silhouette vertex, as shown in Figure 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 1b and 1c). We discuss how to compute alpha in Section 3.1.

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 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[29] 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.

3.1. 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 4a).

There are two problems with this approach, both shown in Figure 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 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 shadow map rendered in Step 1.

Unfortunately, using the shadow map to resize the smoothies on graphics hardware requires texture accesses

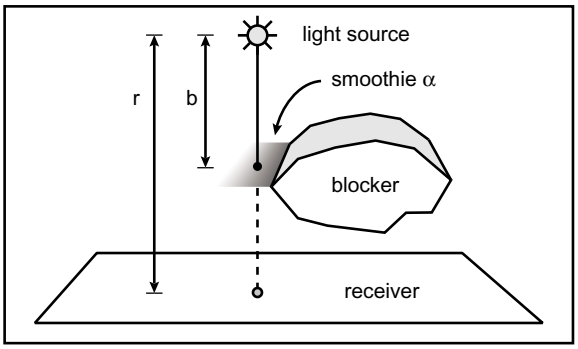

Figure 5: *Alpha computation. The penumbra size depends on the ratio b/r, where b is the distance between the light and the blocker, and r is the distance between the light and the receiver.*

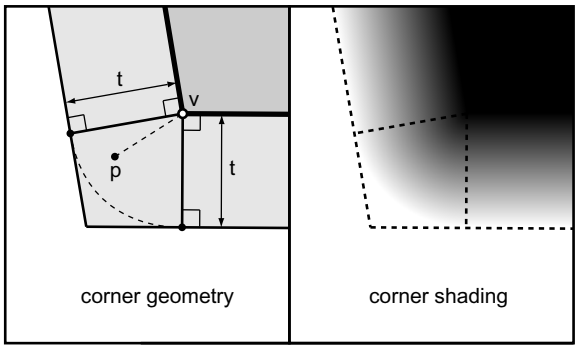

Figure 6: *Smoothie corner geometry and shading. A sample point $\vec{p}$ within the corner is assigned $\alpha = |\vec{v} - \vec{p}|/t$, which is then clamped to $[0, 1]$ and remapped using Equation 1.*

within the programmable *vertex* stage, a feature that is not yet available. Based on the DirectX vertex shader 3.0 proposals[25], however, we expect vertex textures to be implemented within one or two generations of graphics hardware.

In the meantime, 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

$$\alpha' = \frac{\alpha}{1 - b/r} \tag{1}$$

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 4c and 4d.

Smoothie corners must ensure a continuous transition in alpha between adjacent smoothie edges. Figure 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 1. The resulting shaded corner is shown in the right image of Figure 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[7] or backprojection[5, 10] techniques, both of which are expensive. Instead, we use minimum blending, a simple yet effective approximation. Minimum blending, which Parker et al.[27] 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 of light intensity within the penumbra. Parker at al.[27] note, however, that the falloff due to a diffuse spherical light source is sinusoidal, not linear. In our implementation, we follow Haines's suggestion[14] 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. Implementation

We implemented the smoothie algorithm using OpenGL and the Cg shading language[23]. 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[26] 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 3.1. 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.

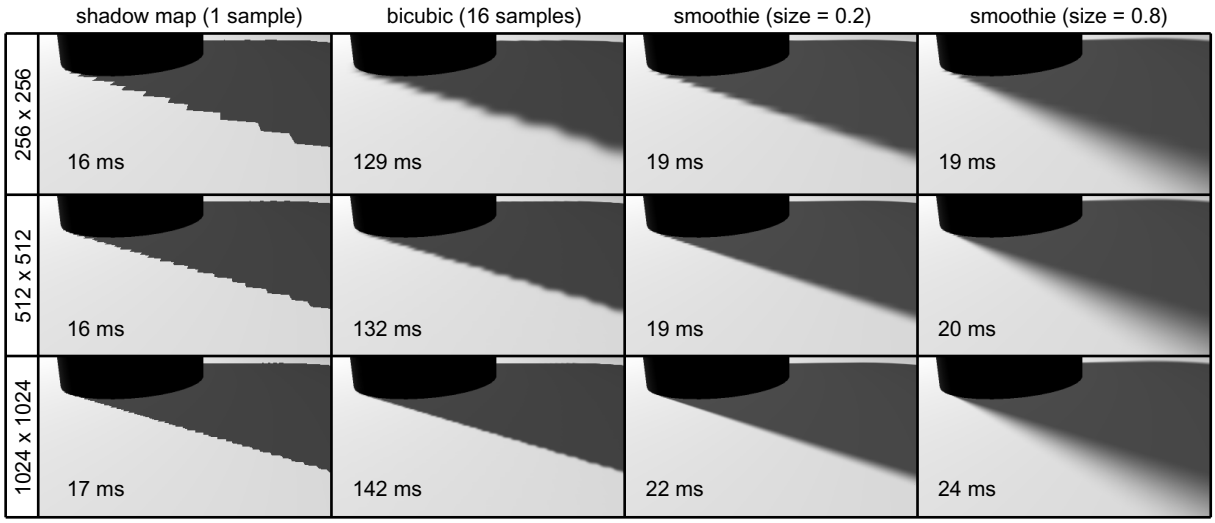

Figure 7: *Shadow edge comparison using different methods and shadow map resolutions. The shadow is cast by the cylinder from Figure 1. The total rendering time per frame is shown at the lower-left of each image.*

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 11. We first compiled the shaders to the NVIDIA vertex and fragment OpenGL extensions[19], 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 11 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.

5. 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.

Figure 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 7 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 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[29] 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 7 indicate that the smoothie algorithm is more expensive than the regular shadow map, but still much faster than using the bicubic filter.

Figure 8 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[14] and Parker at al.[27], this construction produces shadow umbrae that do not shrink as the size of the light 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 9 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

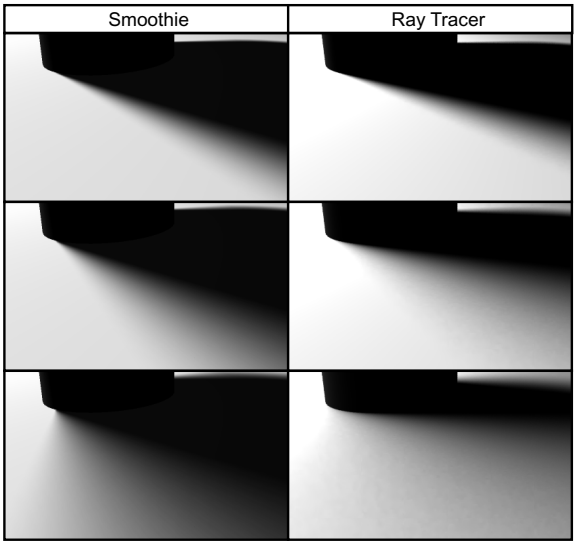

Figure 8: *Comparison of soft shadow edges. The images in the left column are rendered using the smoothie algorithm with a buffer resolution of* 1024×1024*. The images in the right column are rendered using a Monte Carlo ray tracer and show the geometrically-correct soft shadows. Each successive row shows the effect of simulating a larger area light source.*

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 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 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 10 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 12 (see color plates) 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 1 summarizes the rendering performance for the images shown in Figures 1d and 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 12, we found that enlarging the smoothies from $t = 0.02$ to $t = 0.2$ increases the rendering times by about 15%.

6. 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[3, 4]. 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 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 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.

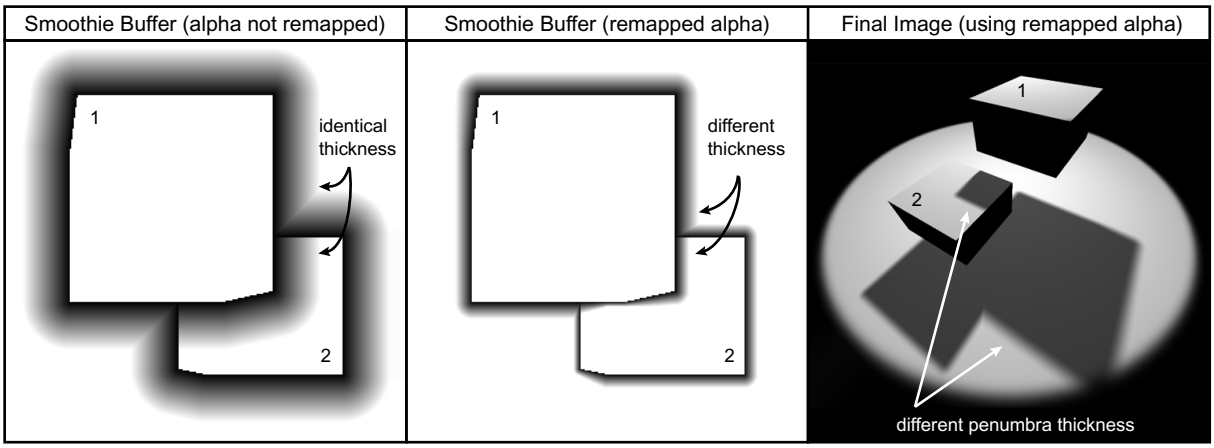

Figure 9: *Scene configuration with multiple blockers and receivers. Remapping the alpha values (shown in the left and middle images) causes the penumbra due to a single blocker to vary in thickness across multiple receivers. As a result, box 1 casts a shadow onto box 2 and the ground plane, but the shadow edges on the ground plane are softer.*

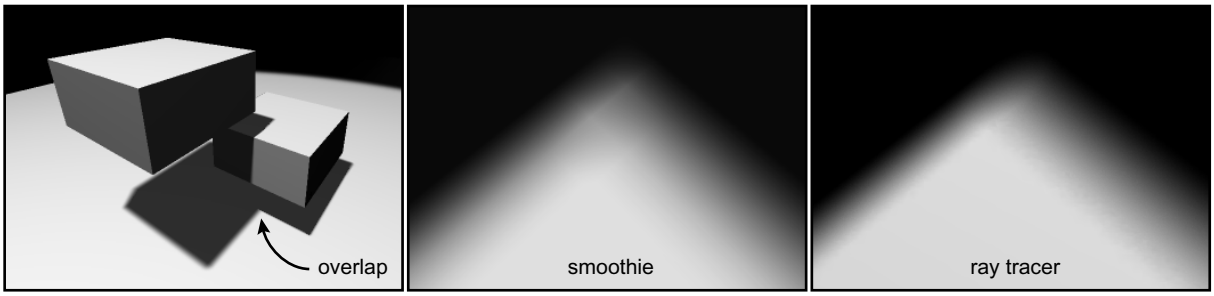

Figure 10: *Comparison of overlapping penumbrae. The left image shows how shadows may overlap due to multiple blockers. The next two images show a close-up of the region indicated by the arrow. The middle image is rendered using the smoothie algorithm. When multiple smoothies overlap, their alpha values 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.*

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[24], but it is unclear how to construct the smoothies directly without an expensive readback from the hardware framebuffer to the host processor. We believe additional hardware is needed to avoid this cost, such as a feedback path from the output of the edge detection pixel shader to a vertex array.

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 7, 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 techniques that increase the effective shadow map resolution, such as perspective shadow maps[32].

6.1. Comparison to Penumbra Maps

Wyman and Hansen's penumbra map algorithm[36] is similar to our method. Both techniques approximate soft shadows

```
// 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;
}
```

```
// 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));
}
```

```
// 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
}
```

Rendering Pass	Vertex	Fragment
Pass 1 (shadow map)	5	1
Pass 2 (smoothie depth)	n/a	n/a
Pass 3 (smoothie alpha)	5 5	11 (edges) 13 (corners)
Pass 4 (make shadows)	12	10

Figure 11: *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.*

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 tesselated 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 tesselate 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

Scene	Boat (Fig. 12, 1st row)	Primitives (Fig. 1d)	Flamingo (Fig. 12, 2nd row)	Elephant (Fig. 12, 3rd row)	Motorbike (Fig. 12, 4th row)
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 1: *Performance measurements for each scene. Timings per frame are given for different buffer resolutions.*

more than two adjacent silhouette edges, as described in Section 3.

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.

7. 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 the 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[9]. Similarly, it may be possible to extend our work to generate more accurate shadow penumbrae by storing extra silhouette data.

Ongoing OpenGL and DirectX 9 proposals[25, 22, 2] indicate that graphics hardware will soon 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 influence the design of shadow algorithms in the future.

Acknowledgments

We are grateful to Mark Kilgard, Cass Everitt, and David Kirk from NVIDIA and Michael Doggett, Evan Hart, and James Percy from ATI for providing hardware and driver assistance. 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. Sylvain Lefebvre, George Drettakis, Janet Chen, Bill Mark, and our anonymous reviewers provided valuable feedback on this paper. Special thanks to Smoothie for casting a very large shadow. This work is supported in part by an ASEE National Defense Science and Engineering Graduate fellowship.

References

1. 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. 1

2. Kurt Akeley. Buffer objects, 2003. *http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_OGL_BufferObjects.ppt*. 10

3. 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. 2, 7

4. Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm on graphics hardware. In *Proceedings of ACM SIGGRAPH*. ACM Press, 2003 (to appear). 2, 7

5. 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. 5

6. Stefan Brabec and Hans-Peter Seidel. Single Sample Soft Shadows Using Depth Maps. In *Proceedings of Graphics Interface*, pages 219–228, May 2002. 2

7. Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of ACM SIGGRAPH*, pages 137–145, 1984. 5

8. Franklin C. Crow. Shadow algorithms for computer graphics. In *Proceedings of ACM SIGGRAPH*, pages 242–248. ACM Press, 1977. 2

9. Carsten Dachsbacher and Marc Stamminger. Translucent shadow maps. In *Proceedings of the Eurographics Symposium on Rendering*, 2003 (to appear). 10

10. 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. 5

11. Cass Everitt and Mark J. Kilgard. Optimized stencil shadow volumes, 2003. *http://developer.nvidia.com/docs/IO/4449/SUPP/ GDC2003_ShadowVolumes.pdf*. 2

12. Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Proceedings of ACM SIGGRAPH*, pages 387–390. ACM Press, 2001. 1

13. Simon Gibson, Jon Cook, Toby Howard, and Roger Hubbold. Rapid shadow generation in real-world lighting environments. In *Proceedings of the Eurographics Symposium on Rendering*, 2003 (to appear). 1

14. Eric Haines. Soft planar shadows using plateaus. In *Journal of Graphics Tools,* vol. 6, no. 1, pages 19–27, 2001. 2, 5, 6

15. Ashley Hartner, Mark Hartner, Elaine Cohen, and Bruce Gooch. Object space silhouette algorithms, 2003. Submitted for publication. 3

16. Paul Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, Computer Science Department, Carnegie Mellon University, 1997. 1

17. 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. 2

18. 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. 3

19. Mark Kilgard. NVIDIA OpenGL extension specifications for the CineFX architecture (NV30), 2002. *http://developer.nvidia.com/docs/IO/1174/ATT/nv30specs.pdf*. 6

20. 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. 3

21. Tom Lokovic and Eric Veach. Deep shadow maps. In *Proceedings of ACM SIGGRAPH*, pages 385–392. ACM Press, 2000. 1

22. Rob Mace. OpenGL ARB superbuffers, 2003. *http://developer.nvidia.com/docs/IO/4449/SUPP/ GDC2003_OGL_ARBSuperbuffers.ppt*. 10

23. 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). 5

24. Michael D. McCool. Shadow volume reconstruction from depth maps. *ACM Transactions on Graphics (TOG)*, 19(1):1–26, 2000. 8

25. Microsoft Corporation. DirectX 9 shader reference. *http://msdn.microsoft.com/library/en-us/directx9_c/directx/graphics/reference/Shaders/Shaders.asp*. 5, 10

26. NVIDIA Corporation. Geforce FX, 2002. http://www.nvidia.com/view.asp?PAGE=fx_desktop. 5

27. Steve Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, 1998. 2, 3, 5, 6

28. 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. 3

29. 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. 1, 4, 6

30. 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. 3

31. 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. 2, 3

32. Marc Stamminger and George Drettakis. Perspective shadow maps. In *Proceedings of ACM SIGGRAPH*, pages 557–562. ACM Press, 2002. 1, 8

33. 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. 1

34. Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of ACM SIGGRAPH*, pages 270–274. ACM Press, 1978. 1

35. Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, November 1990. 1

36. Chris Wyman and Charles Hansen. Penumbra maps. In *Proceedings of the Eurographics Symposium on Rendering*, 2003 (to appear). 3, 8

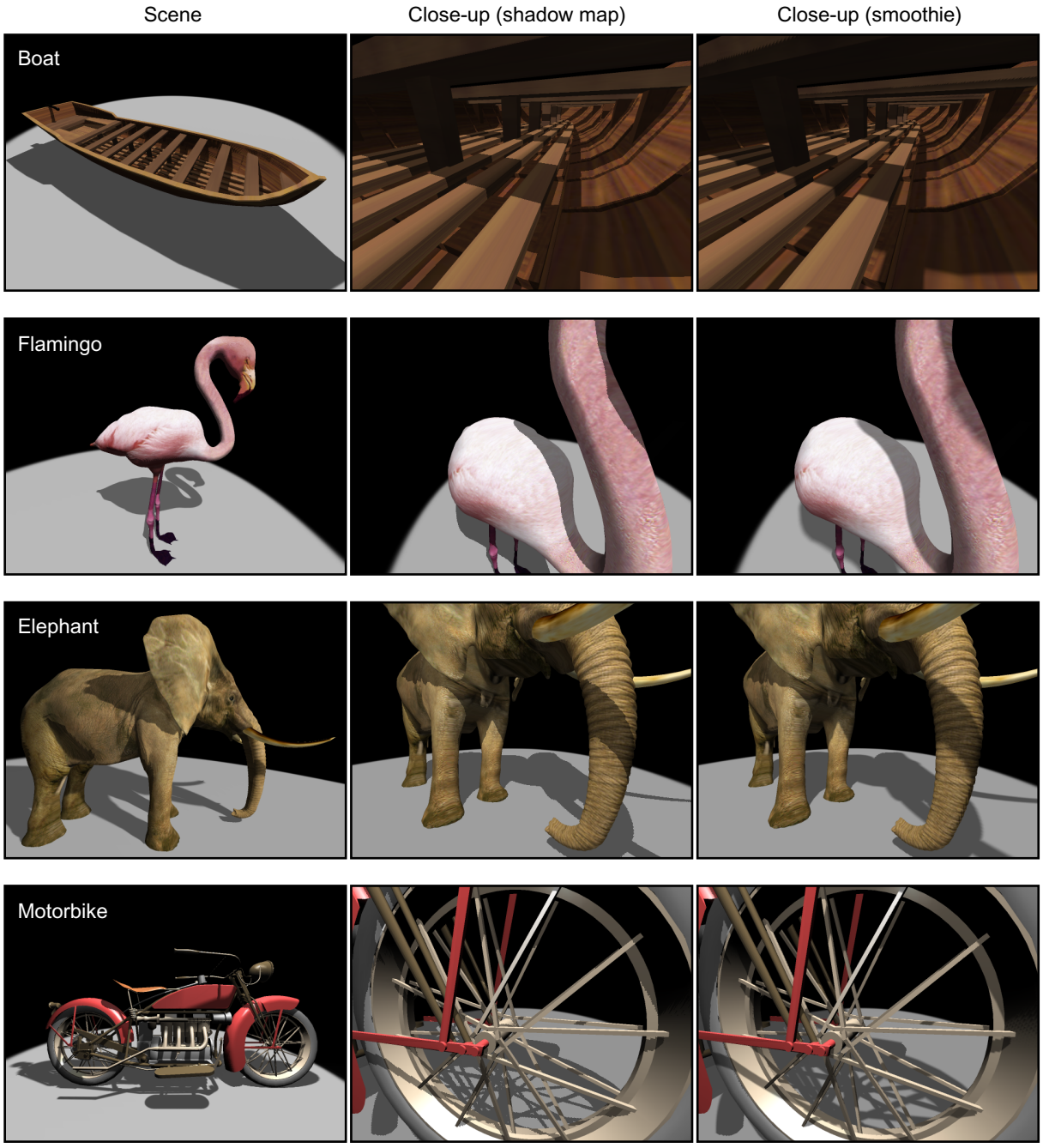

Figure 12: *Scenes rendered using the smoothie algorithm, arranged in order of increasing geometric complexity. These images show the interaction between complex blockers and receivers. For instance, the elephant's tusk casts a shadow onto its trunk, and the spokes of the motorbike wheel cast shadows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