Georgia Institute of Technology (2007)

Real Time Shadow of Transparent Casters Using Shadow Volume

Byungmoon Kim , Kihwan Kim , Greg Turk

College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0760 USA
{bmkim, kihwan23, turk}@cc.gatech.edu

Abstract

Shadow volume algorithms have often been used in real-time rendering applications, because they generate precise shadows on receivers and self-shadows on casters. We believe that processing of colors in shadow will enrich applications of the shadow volume. In this paper, we propose extensions to the shadow volume algorithm to handle colored transparent casters for the first time. We demonstrate how to count the number of shadow caster surfaces between a receiver and the light, and use that count to compute the map of light intensity at each pixel fragment of the receiver.

1. Introduction

In real-time graphics applications such as games, shadows of various objects add important visual realism and provide additional information about spatial relationships to the synthesized scene. For example, a shadow drawn at the foot of a game character makes the user believe that the character is standing on the ground. When the game character is jumping, the user can estimate the height of the character from location of the shadow. Similarly, in CAD or data visualization applications, shadows help the user understand the three-dimensional layout of various objects.

Each shadow-rendering application serves different demands. Some applications may require high quality but may not require computational efficiency, while others demand real-time performance at the price of reduced realism. Such sacrifice in realism includes restrictions on the light source type, the geometry of the shadow receiver, the resolution of the rendered shadow, and others. In general, real-time shadow algorithms have multiple restrictions. For example, shadow maps have aliasing artifacts. Shadow volume algorithms are limited to point or directional light sources and can be computationally inefficient when the depth complexity of the shadow volume is high, but they do not have restrictions on the geometry of the shadow receiver or the shadow resolution.

The shadow volume approach makes use of unseen polygons that are called *shadow caps* to create a shadow volume. We distinguish two kinds of shadow caps, the *side caps* that form the sides of the volume, and *light caps* that are just the original polygons that cast a shadow. All shadow volume algorithms use counting to determine whether a surface is inside a shadow volume, often using the stencil buffer.

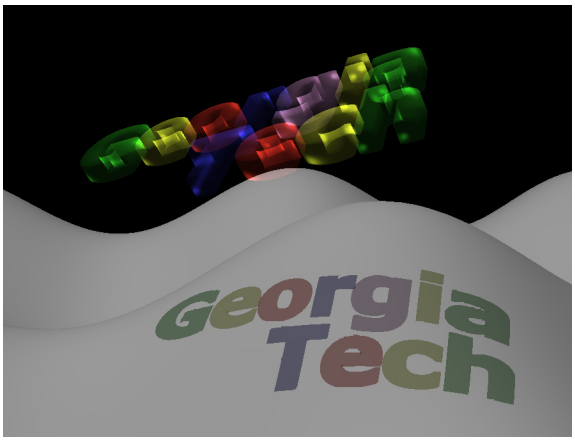

Figure 1: *Shadow of a colored transparent caster.*

In this paper, we extend the shadow volume algorithm to render shadows of colored transparent surfaces. To the best of our knowledge, no previous shadow algorithm (which include shadow volume algorithms) allow transparent casters in real-time settings. We achieve this by rendering a map that stores the intensity of light that travels through multiple transparent surfaces and arrives at the receiver fragment. This may be considered as an extension of the stencil buffer that is used in shadow volume algorithms. In addition, we

show that the proposed algorithm is also useful in visualizing the internal structure of a complex object.

Since Crow [Cro77] first developed this approach, the shadow volume algorithm has been extended in several ways. Heidman [Hei91] introduced a multi-pass rendering algorithm to compute the front or back-facing orientation of shadow caps on the CPU. Everitt and Kilgard [EK03] suggested a two pass method used in many stencil based algorithms. In order to robustly generate shadows when the camera moves into umbra, Batagelo [BJ99], Kilgard [Kil99], and McCool [McC01] suggested to cap off the shadow volume intersection with the near clip plane. To avoid the difficulty of clipping shadow side caps, Everitt and Kilgard [EK03] and Carmack[2000] devised the Z-fail algorithm. This approach makes use of a shadow cap that is beneath the receiver surface. Hornus [HHLH05] suggested a new Z-pass approach that provides additional robustness. Recently, real-time soft shadow algorithm using shadow volume were studied in [AAM03, BCS06]. Other real-time shadow algorithms includes shadow mapping [Wil78, RSC87, FFBG01, SD01, CD03] and shperical harmonics [RWS*06].

Early shadow volume algorithms were limited to manifold polygon meshes, and Bergeron [Ber86] extended the method to manifolds with boundaries. Shadow volume methods were further extended by Aldridge and Woods [AW04] to general polygon soup models (arbitrary connectivity including non-manifold edges). Their approach is fairly complex, however, and we introduce a simplified approach in our own work. Furthermore, we extend our approach to render transparent casters.

2. Counting Caster Surfaces Between the Light and the Receiver

The algorithm proposed in [AW04] makes use of a stencil buffer to count the number of blockers. When the buffer entry is zero, the fragment corresponding to the buffer entry receives light. Otherwise, the fragment does not receive light. In fact, the value in the stencil buffer is the number of caster surfaces between the light and receiver fragment. In this section, in order to show that the stencil contains the number of shadow caster surfaces, we provide another presentation of the shadow volume algorithm proposed in [AW04].

Consider a triangle soup shadow caster. Edges of the caster mesh can be adjacent to an arbitrary number of triangles. With this mesh, the shadow volume algorithm for triangle soup runs as follows. First, we loop over all edges of the triangle soup. On each edge, we test all triangles attached to the edge. Consider the edge E_4 depicted inside the balloon in Fig. 2. There exist four triangles T_0, T_1, T_2 and T_3 that share E_4. Note that the shadow side cap S_4 is the extrusion of the E_4 along the light direction. We test whether each of T_0, T_1, T_2 and T_3 is in front of S_4 or not. As shown in the figure, E_4 has three triangles in front of S_4, and one triangle

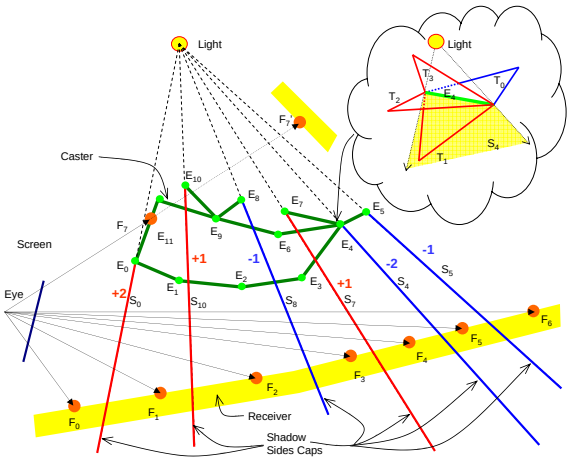

Figure 2: *The stencil built by the shadow volume algorithm contains number of caster surfaces between the receiver fragment and the light. For example, F_0 has 0, F_1 has 2, F_2 has 3.*

in the back of S_4. We compute the stencil increments for S_4 as the number of triangles in back minus the number of triangles in the front. To facilitate further discussions, we call this the *cap count*. The cap count for S_4 is $+3-1=+2$. We perform this operation for all edges and compute the cap counts for all edges. This is a linear time operation and it is a slight generalization of the silhouette computation. Therefore, the asymptotic computation time remains the same.

In Fig. 2, each side cap S_i has a different cap count. For example, E_0 will have +2 because two surfaces are behind it, E_7 and E_{10} will have +1, E_5 and E_8 will have -1. The non-manifold edge E_9 will have a zero cap count, and $E_{1,2,3,6,11}$ will also have a zero cap count. Notice that practical triangle soup models found in computer game are made of several patches that may be non-manifold. This is because game artists prepare several patches and then create the final model by performing operations such as welding vertices. Therefore, a large number of edges will have zero increments similar to $E_{1,2,3,6,11}$. The stencil is created by rendering side caps S_i while increasing or decreasing the stencil by their associated stencil increments. Because of this, the side caps that have zero increments do not need to be rendered. We render only $S_{9,7,10}$ and $S_{4,5,8}$.

The correctness of this algorithm can be seen from the equivalence of this algorithm to a naive algorithm that loops over all triangles while rendering the three side caps for each triangle. Suppose that the three side caps of the first triangle of the caster are rendered. Then, we can obtain the stencil map that contains the shadow of the first triangle. If we process one more triangle, the fragments under the shadow of the second triangle will have their stencil values increased by one. Therefore, the fragments that are occluded from the light by both of the triangles will have the stencil value of two. If we continue adding triangles until we process all the

triangles, we obtain the stencil map that contains number of caster surfaces between the fragment and the light. Notice that this naive algorithm is equivalent to looping over all edges of the caster mesh, and for each edge, drawing all side caps. Since drawing side caps of edge multiple times is equivalent to incrementing the stencil by the cap count, the naive algorithm is equivalent to the algorithm in [AW04]. The equivalence of [AW04] to the naive algorithm shows that the number in the buffer represents the number of caster surfaces between fragments of the receiver and the light. This is already mentioned in [Ber86] for the caster that is a manifold with boundary.

In Fig. 2, consider the Fragments F_4. The side caps drawn on the fragment F_4 are $S_{0,10,8,7}$. Therefore, the cap count is added to yield $2+1-1+1=3$, which is indeed the number of green caster surfaces between F_4 and the light.

3. Computing The Light Intensity Map

In this section, we describe how to extend the algorithm discussed in section 2 in order to compute the light intensity that arrives at each fragment. First, note that we process each group of triangles that have the same transparency and colors. Second, since the stencil buffer allows only a limited set of operations, we do not use it anymore. In addition, the stencil buffer does not even allow us to increase or decrease the stencil count by more than one. Therefore, [AW04] rendered side cap multiple times to increase or decrease the stencil by more than one. Moreover, the stencil buffer has limited resolution. In current graphics hardware, the stencil buffer only has 8 bits per pixel. Thus, we use a buffer that contains four 16-bit floating point numbers per each pixel.

Typically, shadow volume algorithms first render receivers to build the depth buffer, and render side caps of casters to build the stencil buffer. We modify this step by rendering directly to the 16-bit floating point buffer. We render side caps in a modified way so that when all the side caps are rendered, this buffer contains the light intensity arrived at each fragment. This floating point buffer is provided to the final rendering step as a texture map. During the final step, all receivers are rendered using the lighting information contained in the provided floating point buffer.

We now describe how to build the light intensity map. Suppose that a caster surface has opacity α. Then, the light fraction passing though this caster surface will be $(1-\alpha)(r,g,b)$, where (r,g,b) and α are color and the opacity of the caster, respectively. If n surfaces exists between the light and the fragment, the light fraction will be $(r_\alpha^n, g_\alpha^n, b_\alpha^n)$, where $r_\alpha=(1-\alpha)r$, $g_\alpha=(1-\alpha)g$, and $b_\alpha=(1-\alpha)b$. Finally, the light arrived at the receiver surface will be $(r_\alpha^n I_r, g_\alpha^n I_g, b_\alpha^n I_b)$, where (I_r,I_g,I_b) is the light intensity in three color channels. The light intensity can be computed using the extended shadow volume algorithm. The exponent n is the sum of the cap counts of all side caps as shown in Fig. 2.

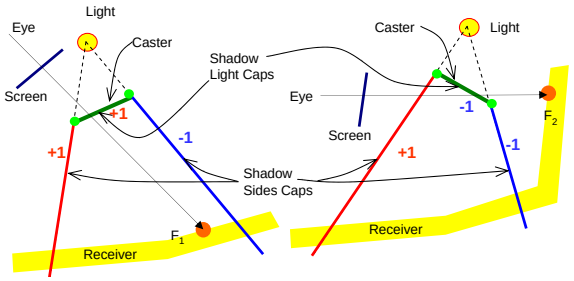

Figure 3: *If the light and the eye are in the same side of the caster triangle, the triangle has a count of +1 (left), and otherwise -1 (right)*

Rendering side caps is not sufficient when the caster surface is transparent since transparent caster reveals information that an opaque caster occludes. In Fig. 2, the fragment F_7 no longer exists, but we have the fragment F_7'. However, if we render the side caps only, F_7' will be in the shadow since it is only behind the side cap of E_{10}. This problem can be resolved by rendering the caster triangles in a way that is similar to side caps. We call these caster triangles *light caps*. To compute cap counts for the light caps, we first consider a single triangle caster.

In Fig. 3, on the fragment F_1, no side cap is rendered. Therefore, F_1 is not in the shadow, which is incorrect. To fix this, the caster surface should be rendered as a light cap with a count of $+1$. Similarly, for F_2, the light cap should be rendered with the count -1. In general, if the light and the eye are in the same side of the caster triangle, it is a front face of the shadow volume. Therefore, the cap count is $+1$. In contrast, when the light and the eye are in different side of the caster triangle, it is a back face of the shadow volume. Therefore, we let the effective count be -1. Similarly to the discussions in section 2, this single triangle caster can be generalized to triangle soup casters. First, the rendering side caps remain the same. We now render the shadow caster triangles as light caps with the cap count computed by the method discussed above.

Let c be the cap count of a side cap. Then, if we initialize the light intensity map by (I_r,I_g,I_b), and then multiply $(r_\alpha^c, g_\alpha^c, b_\alpha^c)$ for each side cap, the exponent will be summed. Since the sum of effective triangles of all side caps is the number of caster surfaces, we can compute the light intensity that arrives at the receiver surface.

When there exist surfaces with different colors and opacities, we perform this operation repeatedly. Suppose that there exists n_c caster surfaces. Let their color and opacity be $(r_i,g_i,b_i,\alpha_i), i=1,2,\ldots,m$. Then the light fraction becomes

$$((1-\alpha_i)r_i,(1-\alpha_i)g_i,(1-\alpha_i)b_i)\equiv(r_{\alpha_i},g_{\alpha_i},b_{\alpha_i}) \quad (1)$$

In addition, assume that n_{S_i} is the number of side caps of i^{th} caster rendered on a receiver fragment. Also assume that j^{th} side cap of i^{th} caster has cap count $c_{i,j}$. Then the red chan-

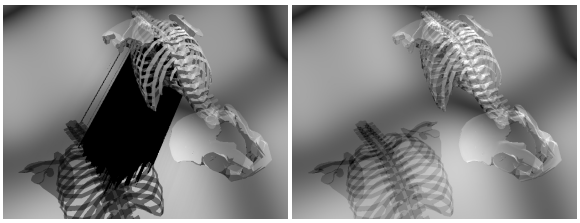

Figure 4: *The multiplicative rendering of side caps produces error when the depth complexity is high (left). When the logarithm is used, the shadow is rendered properly (right).*

nel of the light intensity that arrives at a fragment is

$$\left(\prod_{i=1}^{n_c}\prod_{j=1}^{n_{S_i}}(r_{\alpha_i})^{c_{i,j}}\right)I_r. \tag{2}$$

The green and blue channel intensities can be expressed similarly. As we can see from (2), the intensity computation can be performed in any order. Therefore, the side caps can be drawn in any order.

4. Implementation

When we implement (2), using the alpha blending operation, the resolution of 16-bit floating point buffer does not seem to be enough. When the depth complexity of the side caps is high, we observed severe artifact. In a relatively simple model, we can observe weak but noticeable artifacts. We show an example in the left image of Fig. 4. This is due to the fact that when r_{α_i} is not an integer power of 2, for some number x, $(xr_{\alpha_i})/r_{\alpha_i} \neq x$ due to a numerical error. A solution to this would be using the logarithm of those numbers since $[\log(x)+log(r_{\alpha_i})]-log(r_{\alpha_i})$ is exactly the same as $log(x)$ as far as the logarithms remain in the range of 16 bit floating point number that is approximately $\pm 6\times 10^5$.

Another immediate improvement can be made when a 32-bit floating point buffer is used. Unfortunately, alpha blending on a 32-bit floating point buffer is only supported by the latest GPU such as NVIDIA Geforce 8800. Therefore, we propose to compute the logarithm of the light intensity map. Taking the logarithm of (2), we compute

$$\left(\sum_{i=1}^{n_c}\sum_{j=1}^{n_{S_i}}c_{i,j}\log_a(r_{\alpha_i})\right)+\log_a I_r. \tag{3}$$

Now, since we use a summation, the truncation error does not exist as long as the logarithms stay within a reasonable range. After computing (3), we recover the light map by taking exponential of (3). As shown in the right image of Fig. 4, the shadow is rendered properly.

Note that when r_{α_i} is very small, the logarithm will be a large negative number. In our test, (3) works well if $r_{\alpha_i} > 0.05$. Since 0.05 is a barely noticeable intensity, we believe that (3) is a viable option. In addition, we notice that since 32-bit floating point buffer supports alpha blending in the latest GPU, our light intensity map computation will be more easy to use in the near future.

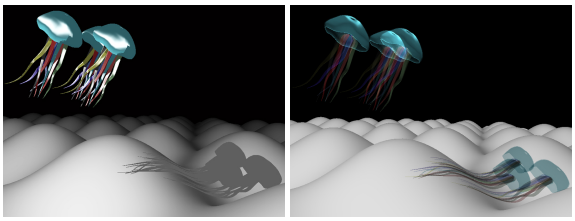

Figure 5: *Images rendered by a traditional shadow volume algorithm (left) and by the new shadow volume algorithm (right).*

5. Limitations of Self-Shadowing

For opaque casters, the typical shadow volume algorithm renders both the receiver and the caster to obtain the depth buffer. This makes casters into receivers, and therefore, produces self-shadows. Unfortunately, this is no longer true in the proposed shadow volume algorithm for transparent casters. Note that we build the light intensity map that provides lighting per fragment. If there are transparent casters, and if we want to render the casters under the shadow, the light transferred to each of these transparent casters must be computed for each fragment of these casters. This would require a buffer that can store values at multiple depths. Since current graphics hardware allows only a single depth per pixel, self-shadows cannot be fully resolved, although a partially correct self-shadows are possible by computing self-shadows in a separate step, as shown in section 6.

Note that self-shadows are often visually distracting due to the noisy self-shadow boundary. For this reason, commercial computer games, such as Doom III, do not recommend turning self-shadows on.

6. Algorithm Summary

The proposed transparent shadow algorithm performs the following two pass rendering.

1. Render receivers to build the depth buffer.
2. Compute and render shadow caps to build the light intensity map.
3. Using the light intensity map, render the receiver
4. Render transparent shadow casters.

Although the correct self-shadowing is not possible, partially correct self-shadows can be obtained by modifying the above step 4 by

4. Render shadow casters to build the depth buffer.
5. Render caps to obtain the intensity map on casters.
6. Using the intensity map on casters, render transparent shadow casters.

Although the self shadowing is limited, this method increases the visual realism as shown in Fig. 7.

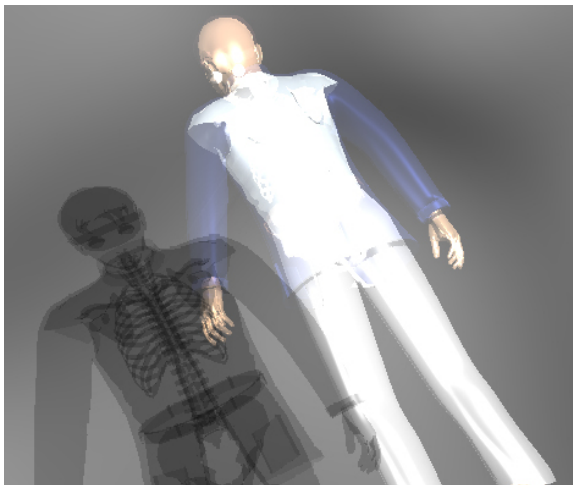

Figure 6: *By making object transparent and by rendering its shadow, the internal structure of a mesh can be visualized.*

7. Results and Discussions

As shown in figures 5 and 1, our algorithm produces shadows of transparent objects. In Fig. 7, a partially correct self-shadow is shown. In Fig. 6, we show that the internal mesh is revealed in the shadow, and we believe that this helps the user to understand the mesh. In Table 1, we compare the rendering times of the traditional stencil-buffer-based shadow volume for opaque models (See the left image of Fig. 5.) and the proposed algorithm for transparent models.

Model	# triangles	opaque	no s.s.	s.s.
Ball&cubes	1120	703.4	512.5	288.4
gh2007	13576	193.1	98.7	56.0
Jellyfish	27736	35.0	24.9	14.9
Skeleton&man	62724	41.4	21.7	12.9

Table 1: *Comparison of frame rates (frames/sec.) for conventional shadow volume using a stencil buffer (opaque), transparent shadows without self-shadowing (no s.s.) and with partial self-shadowing (s.s.) on a PentiumD 3.0GHz PC with NVIDIA GeForce 6800.*

8. Conclusion

We have presented an extension to the shadow volume algorithm that renders shadow of the transparent objects. We show that the shadow volume algorithm can compute the light intensity arrived at each receiver fragment. In addition, we have demonstrated that a low resolution floating point buffer can be used to compute the light intensity map by taking the logarithm of the light fractions. The proposed algorithm can also be used to visualize internal mesh structures, and the light intensity map may be used to accelerate the shadow ray computation in ray tracing applications.

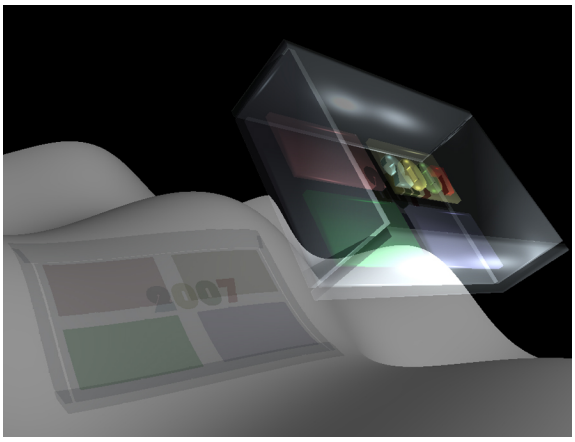

Figure 7: *By rendering the model once, we obtain partial self shadowing. See the shadow of letters at the bottom of the glass box.*

References

[AAM03] ASSARSSON U., AKENINE-MOLLER T.: A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Trans. Graph. 22*, 3 (2003), 511–520.

[AW04] ALDRIDGE G., WOODS E.: Robust, geometry-independent shadow volumes. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (2004), pp. 250–253.

[BCS06] BAVOIL L., CALLAHAN S. P., SILVA C. T.: Robust soft shadow mapping with depth peeling. In *SCI Institute Technical Report, No. UUSCI-2006-028, University of Utah* (2006).

[Ber86] BERGERON P.: A general version of Crow's shadow volumes. *IEEE Computer Graphics and Application 1*, 1 (1986), 17–28.

[BJ99] BATAGELO H. C., JUNIOR I. C.: Real-time shadow generation using bsp trees and stencil buffers. In *XII Brazilian Symposium on Computer Graphics and Image Processing* (1999), pp. 93–102.

[CD03] CHAN E., DURAND F.: Rendering fake soft shadows with smoothies. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 208–218.

[Cro77] CROW F.: Shadow algorithms for computer graphics. In *Proceedings of SIGGRAPH* (New York, NY, USA, 1977), vol. 11, ACM Press, pp. 242–248.

[EK03] EVERITT C., KILGARD M. J.: Practical and robust stenciled shadow volumes for hardware-accelerated rendering. *http://developer.nvidia.com/object/robust_shadow_volumes.html* (2003).

[FFBG01] FERNANDO R., FERNANDEZ S., BALA K., GREENBERG D. P.: Adaptive shadow maps. In *In Proceedings of ACM SIGGRAPH* (New York, NY, USA, 2001), ACM Press, pp. 387–390.

[Hei91] HEIDMANN T.: Realshadows real time. *IRIS Universe*, 18 (1991), 28–31.

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

[Kil99] KILGARD M.: Improving shadows and reflections via the stencil buffer. In *Advanced OpenGL Game Development course notes Game developer Conference* (1999), pp. 204–253.

[McC01] MCCOOL M. D.: Shadow volume reconstruction from depth maps. *ACM Trans. Graph.*, 1 (2001), 1–25.

[RSC87] REEVES W. T., SALESIN D. H., COOK R. L.: Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM Press, pp. 283–291.

[RWS*06] REN Z., WANG R., SNYDER J., ZHOU K., LIU X., SUN B., SLOAN P.-P., BAO H., PENG Q., GUO B.: Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. In *ACM SIGGRAPH* (2006), pp. 977–986.

[SD01] STAMMINGER M., DRETTAKIS G.: Perspective shadow maps. In *In Proceedings of ACM SIGGRAPH* (New York, NY, USA, 2001), ACM Press, pp. 557–562.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. In *In Proceedings of ACM SIGGRAPH* (New York, NY, USA, 1978), ACM Press, pp. 270–274.