Vol. [VOL], No. [ISS]: 1–??

A Shadow Volume Algorithm for Opaque and Transparent Non-Manifold Casters

Byungmoon Kim[1], Kihwan Kim[2], Greg Turk[2]
[1]NVIDIA, [2]Georgia Institute of Technology
Sunday 20th July, 2008

Abstract. Since precise shadows can be generated in real-time, graphics applications often use the shadow volume algorithm. This algorithm was limited to manifold casters, but recently has been extended to general non-manifold casters with oriented triangles. We provide a further extension to general non-manifold meshes and an additional extension to shadows of transparent casters. To achieve these, we first introduce a generalization of an object's silhouette to non-manifold meshes. By using this generalization, we can compute the number of caster surfaces between the light and receiver, and furthermore, we can compute the light intensity arrived at the receiver fragments after the light has traveled through multiple colored transparent receiver surfaces. By using these extensions, shadows can be generated from transparent casters that have constant color and opacity.

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

Rendering shadows is still a problem without a unified approach. Therefore,

© A K Peters, Ltd.

1086-7651/06 $0.50 per page

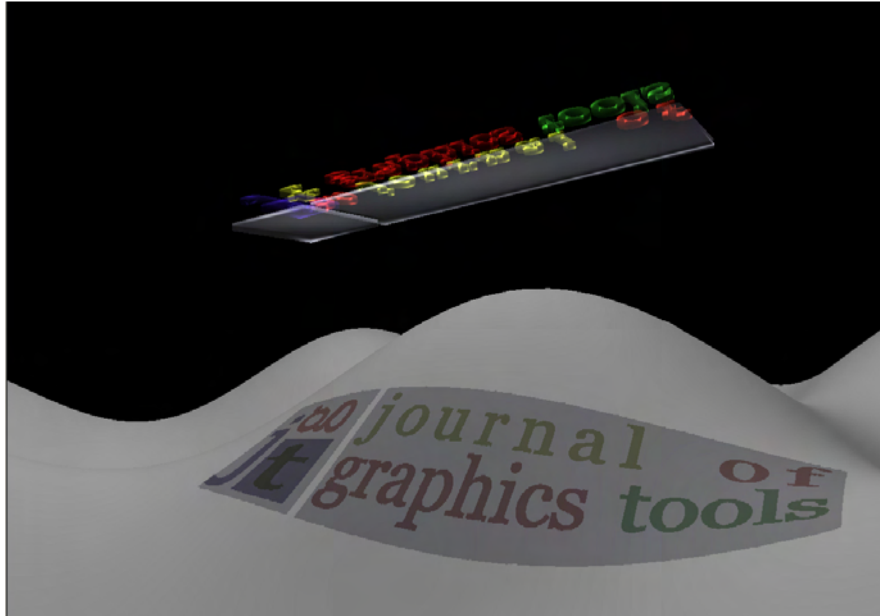

Figure 1. Shadows from a colored transparent caster.

different algorithms have been developed for various applications. Among such applications, games require real-time rendering of shadows, for which two commonly used methods are available: the shadow map and the shadow volume algorithms. The shadow map algorithms are fast and robust, but have aliasing artifacts since the resolution of the shadow map is limited. In contrast, shadow volume algorithms, introduced in [Crow 77], do not have aliasing artifacts, but have been limited to manifold meshes or non-manifold meshes made of one-sided polygons. We propose a method that extends the application of the shadow volume to general non-manifold mesh that can even become transparent as shown in Fig. 1.

Suppose that we have a screen-space mask that represents whether pixels are inside the shadow or not. Then, shadows are trivially obtained by rendering each pixel with or without specular and diffuse lighting depending on the mask. Therefore, computing this mask is the main idea in the shadow volume method. We briefly introduce this idea in the following paragraphs.

We first assume a point light source. The mask is obtained by rendering the boundary of the umbra. The boundary of the umbra consists of the caster surface and special set of polygons obtained by extruding the silhouette edges of the caster along the light direction. To facilitate future discussions, we will use the term *shadow polygon* to refer to the polygon that bound the shadow volume. In fig. 2, where the umbra of a triangle $V_0V_1V_2$ is shown, shadow polygons are the triangle $V_0V_1V_2$, and all the polygons obtained by extruding edges. Furthermore, we distinguish two types of shadow polygons: the *extrusion polygons* that form the sides of the volume,

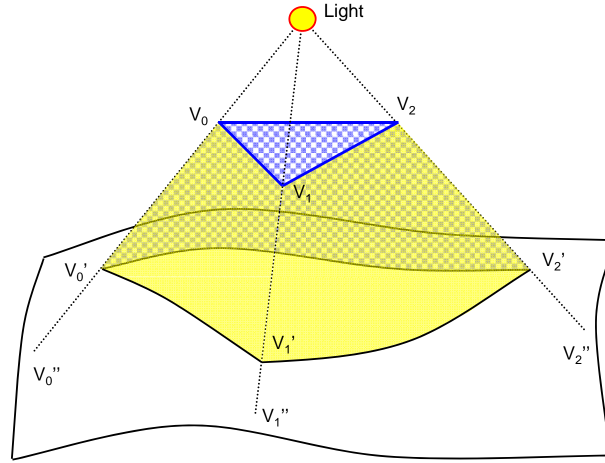

Figure 2. The shadow volume of a triangle.

and *light caps* that are polygons of the caster surface. For example, Fig. 2 contains one light cap that is the caster triangle $V_0V_1V_2$, and three extrusion polygons that are extruded from edges V_0V_1, V_1V_2, and V_2V_0.

Whether a visible fragment P at a pixel is in shadow is determined by checking how many front-facing and back-facing shadow polygons are between P and the viewer. This is determined by rasterizing each of the shadow polygons and incrementing or decrementing per-pixel counters in the case of front-facing or back-facing shadow polygons, respectively. A given pixel's counter is only incremented or decremented, however, if the shadow polygon in question is in front of the fragment P, as determined by a depth test during shadow polygon rasterization. The fragment P is in shadow if the final value of the counter is greater than zero, otherwise it is not in shadow. In practice, the stencil buffer is often used as the per-pixel counter, and the final shadow mask is obtained by testing whether the counter is greater than zero. In the example shown in Fig. 2, the shadow cap $V_0V_1V_2$ and the extruded polygons from V_0V_1 and V_1V_2 are front-facing, and thus would increment the per-pixel counters when they are in front of the visible surface at a given pixel. The extruded polygon from V_2V_0 is back-facing, and thus would decrement the per-pixel counters.

This algorithm described for the single triangle example in Fig. 2 needs to be extended for meshes. For triangle meshes, the light caps are still the mesh itself, but extrusion polygons are created by extruding the silhouette edges only (not all edges), as illustrated in Fig. 3(a). This approach is efficient, but requires a silhouette that is only well-defined for two-manifold meshes. Solutions are given for the manifold with edge case in [Bergeron 86], but not given for general non-manifold meshes. In [Aldridge and Woods 04], a method to handle general non-manifold mesh is given, but it required the mesh to be orientable. Therefore, the algorithm does not work for surface types (b) and (c) in Fig. 3, in which polygons in (b) and (c)

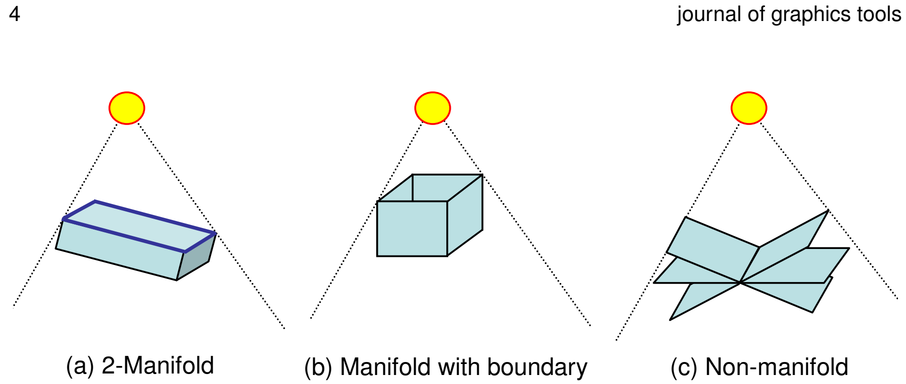

Figure 3. Shadow volumes applied to various mesh types.

are two-sided. In fact, meshes in (b) and (c) can be handled trivially by applying their algorithm to two sided triangles after orienting all triangles towards the light. However, this extension is neither discussed nor is the correctness proven in [Aldridge and Woods 04]. Although the extension to two-sided meshes is trivial, the proof of correctness is not an easy consequence. Therefore, we provide an alternative approach that makes the algorithm intuitive, leads to the proof, and provides a further extension to transparent casters.

In this paper, we extend the shadow volume algorithm to render shadows of colored transparent surfaces. 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 traditional shadow volume algorithms. In addition, we show that the proposed algorithm is also useful in visualizing the internal structure of a complex mesh.

All shadow volume algorithms use counting in order to determine whether a surface is inside a shadow volume. Often the stencil buffer is used, but stencils cannot be increased or decreased by more than one in current GPUs. The work-around is to render the extruded polygons multiple times, which reduces the performance. Moreover, the transparent geometry will require more complex operations than increment or decrement, making the use of the stencil buffer impossible. Therefore, we use a floating point render target instead of the stencil buffer.

Our first contribution is extending the algorithm presented in [Aldridge and Woods 04] to non-oriented triangle meshes. Our second contribution is the computation of the number of caster surfaces between the light and receiver, and a proof of its correctness. Our third contribution is the computation of the light intensity that arrives at a fragment with various methods and rendering the shadow of transparent casters.

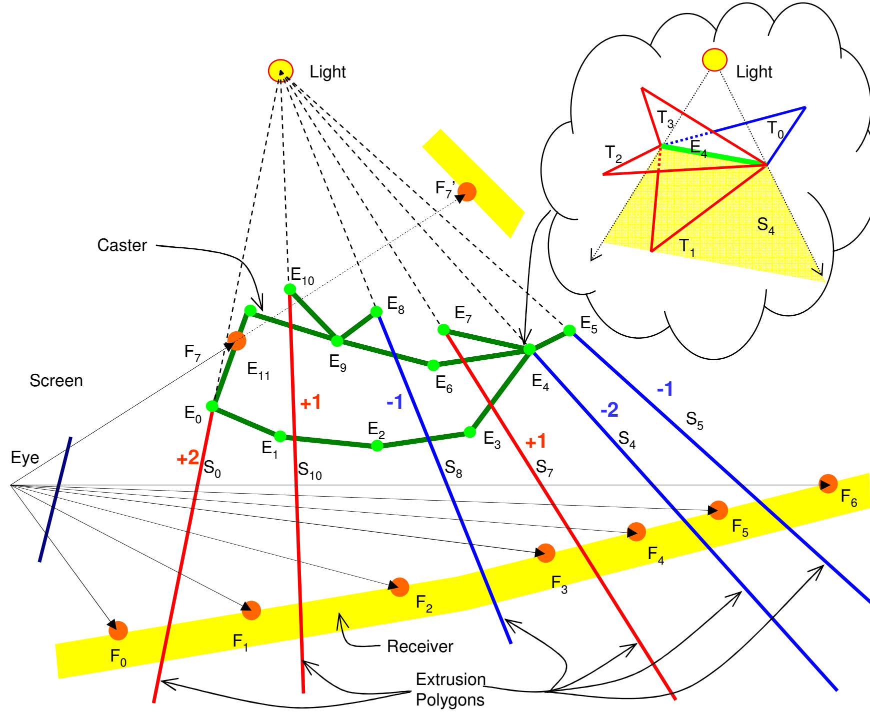

Figure 4. The stencil built by the proposed shadow volume algorithm contains number of caster surfaces between the receiver fragment and the light. For example, F_0 has 0, F_1 has 2, F_2 has 3. Note that the bright green dots $E_0, E_1, ...,$ and E_{11} are illustrations of edges observed from a viewer whose viewing direction is parallel to the edges. Therefore, they are observed as points. This is to facilitate the discussion of the algorithm, and in practice, the edges do not have to be parallel.

1. Counting Caster Surfaces Between the Light and the Receiver

1.1. A Generalization of the Silhouette to Non-manifold Meshes

Consider a shadow caster with a non-manifold triangle mesh, where the edges of the caster mesh can be adjacent to an arbitrary number of triangles. On this mesh, suppose that we loop over all edges of the triangle mesh. On each edge, we test all triangles that are adjacent to the edge: Consider the edge E_4 shown inside the balloon in Fig. 4. Since the edge is parallel to the viewing direction, E_4 is depicted as a bright green dot in Fig. 4. We extrude the edge E_4 along the light direction to obtain the extrusion polygon S_4. Note that E_4 is shared by four triangles T_0, T_1, T_2 and T_3. 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 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. Therefore, the final sum for S_4 is $+3-1=+2$. To facilitate further discussions, we call this the *multiplicity* of a silhouette edge. We compute multiplicities for all edges. If we remove the view-dependency by taking the absolute value of the multiplicity, the set of edges with non-zero multiplicity is a generalization of the silhouette to non-manifold.

In Fig. 4, each extrusion polygon S_i has a different multiplicity. For example, E_0 will have multiplicity $+2$ because two surfaces are behind it, E_7 and E_{10} will have multiplicity $+1$, E_5 and E_8 will have -1. The non-manifold edge E_9 will have zero multiplicity, and $E_{1,2,3,6,11}$ will also have a zero multiplicity. Notice that in practice non-manifold triangle meshes found in computer game are often 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 such as $E_{1,2,3,6,11}$ will have zero multiplicity.

1.2. Algorithm to Count Caster Surfaces Between Light and Receiver

We render the extrusion polygons and light caps so that the resulting render buffer contains the number of caster surfaces between light and receiver.

The stencil is created by rendering extrusion polygon S_i while increasing or decreasing the stencil by their associated stencil increments. Because of this, the extrusion polygons that have zero increments do not need to be rendered. For Fig. 4, for example, we would render only $S_{9,7,10}$ and $S_{4,5,8}$.

Rendering extrusion polygons is not sufficient when the caster surface is transparent since a transparent caster reveals information that an opaque caster occludes. In Fig. 4, the fragment F_7 no longer exists, but we have the fragment F_7'. However, if we render extrusion polygons only, F_7' will be in the shadow since it is only behind the extrusion polygon of E_{10}. This problem can be resolved by rendering the caster triangles in a way that is similar to extrusion polygons. We call these caster triangles *light caps*. To compute multiplicities for the light caps, we first consider a single triangle caster.

In Fig. 5, on the fragment F_1, no extrusion polygon 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 on the same side of the caster triangle, it is a front face of the shadow volume. Therefore, the multiplicity is $+1$. In contrast, when the light and the eye are on different sides 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 1, this single triangle caster can be generalized to non-manifold casters. First, the rendering of extrusion polygons remain the same. We now render the shadow caster triangles as light caps with the multiplicity computed by the method discussed above.

The stencil obtained will contain the number of caster surfaces between the light

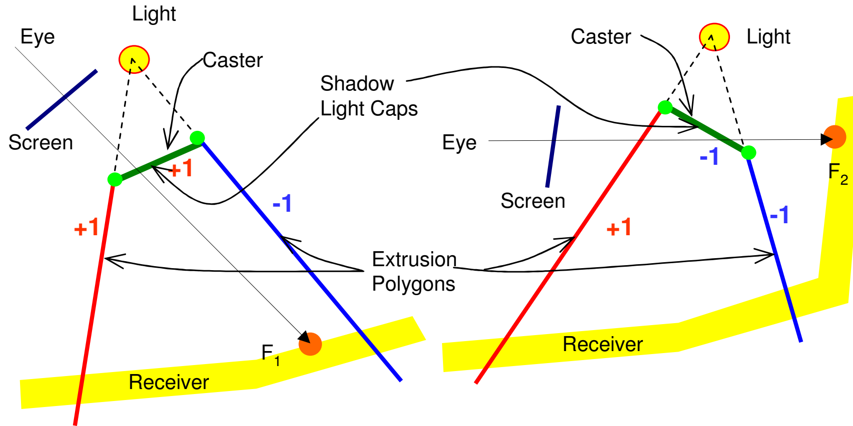

Figure 5. If the light and the eye are on the same side of the caster triangle, the triangle has a count of +1 (left), and otherwise -1 (right)

and the receiver. Before we discuss its proof, we can observe this approach by a few examples. In Fig. 4, consider the fragment F_4. The extrusion polygons drawn at the fragment F_4 are $S_{0,10,8,7}$. Therefore, the multiplicity is added to yield $2 + 1 - 1 + 1 = 3$, which is indeed the number of green caster surfaces between F_4 and the light.

We note that the algorithm described above requires us to know which triangles are adjacent to a given edge, and triangles are processed multiple times. Alternatively, we can achieve the same result by looping over triangles while increasing or decreasing multiplicities. Thus we do not require knowledge of which triangles are adjacent to an edge, just which edges make up a given triangle. This way, triangles do not have to be processed multiple times.

The correctness of the algorithm in the previous section 1.2 can be seen from the equivalence of this algorithm to a naive algorithm that loops over all triangles while rendering each triangle and its three extrusion polygons. Suppose that the three extrusion polygons 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 a stencil value of two. If we continue adding triangles until we process all the triangles, we obtain a stencil map that contains the number of caster surfaces between the fragment and the light. Notice that this naive algorithm is equivalent to looping over all the edges of the caster mesh, and for each edge, drawing all extrusion polygons. Since drawing extrusion polygons of edge multiple times is equivalent to incrementing the stencil by the multiplicity, the naive algorithm is equivalent to the algorithm described above. This equivalence shows that the number in the buffer represents the number of caster surfaces between fragments of the receiver and the light. Note that this

surface count is already mentioned in [Bergeron 86] for the caster that is a manifold with boundary. We generalize this to non-manifold meshes.

In this paper, we use the z-pass shadow volume algorithm only. Therefore, in order to allow the camera to be placed under the shadow, the extrusion polygons must be clipped [Batagelo and Junior 99], [Kilgard 99], and [McCool 01], or the z-fail algorithm [Everitt and Kilgard 03] must be used. Therefore, the applicability of the proposed surface counting approach to the z-fail shadow volume algorithm is an interesting question, the answer to which is rather trivially found by again starting from a single triangle, adding more triangles similar to the z-pass case. Observing that a shared edge has no contribution to the count and hence is skipped, we can see that rendering of the polygons extruded from the generalized silhouettes is sufficient for the z-fail case as well.

2. Computing The Light Intensity Map

In this section, we describe how to extend the algorithm discussed in section 1 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 particular, the stencil buffer does not even allow us to increase or decrease the stencil count by more than one. Therefore, [Aldridge and Woods 04] rendered extrusion polygon multiple times to increase or decrease the stencil by more than one. This consumes memory bandwidth. 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 pixel.

Typically, shadow volume algorithms first render receivers to build the depth buffer, and then render extrusion polygons of casters to build the stencil buffer. We modify this step by rendering directly to the 16-bit floating point buffer. We render extrusion polygons in a modified way so that when all extrusion polygons are rendered, this buffer contains the light intensity arriving 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 that is 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 the 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 arriving 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. This can be computed using the extended shadow volume algorithm. The exponent n is the sum of the multiplicities of all of the extrusion polygons as shown in Fig. 4.

Let c be the multiplicity of a extrusion polygon. 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 extrusion polygon,

the exponent will be summed. Since the sum of effective triangles of all extrusion polygons 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 $(r_i, g_i, b_i, \alpha_i), i = 1, 2, ..., m$. 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 extrusion polygons of i $^{\text{th}}$ caster rendered at a receiver fragment. Also assume that j $^{\text{th}}$ extrusion polygon of i $^{\text{th}}$ caster has multiplicity $c_{i,j}$. Then the red channel of the light intensity that arrives at a fragment is

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

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 extrusion polygons can be drawn in any order. Note that the extrusion polygons are drawn with the depth test enabled.

3. Implementation

When we implement (2), using the alpha blending operation, the resolution of a 16-bit floating point buffer is not sufficient. When the depth complexity of the extrusion polygons is high, we observed severe artifacts. In relatively simple models, we can observe weak but noticeable artifacts. We show an example of the problem in the left image of Fig. 6. 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 large numerical error. One solution to this would be using the logarithm of those numbers since $[\log(x) + log(r_{\alpha_i})] - log(r_{\alpha_i})$ is close to $log(x)$ with very small error so long as the logarithms remain in the expressible range of 16-bit floating point numbers, which 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 often slower than when using a 16-bit buffer. Therefore, we propose to compute the logarithm of the light intensity map. Taking the logarithm of (2), we compute

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

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 the exponential of (3). As shown in the right image of Fig. 6, 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

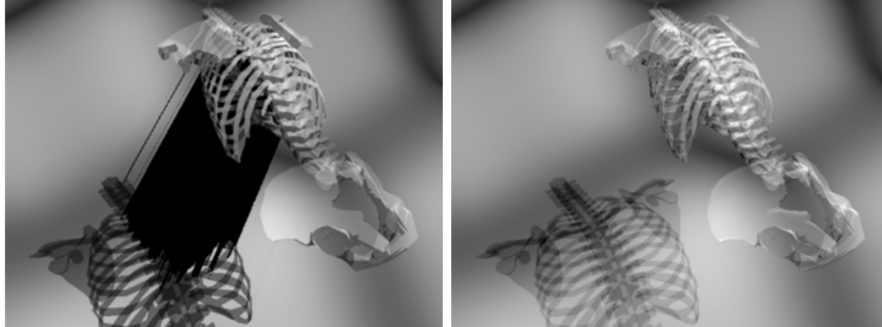

Figure 6. The multiplicative rendering of extrusion polygons produces errors when the depth complexity is high (left). When the logarithm is used, the shadow is rendered properly (right).

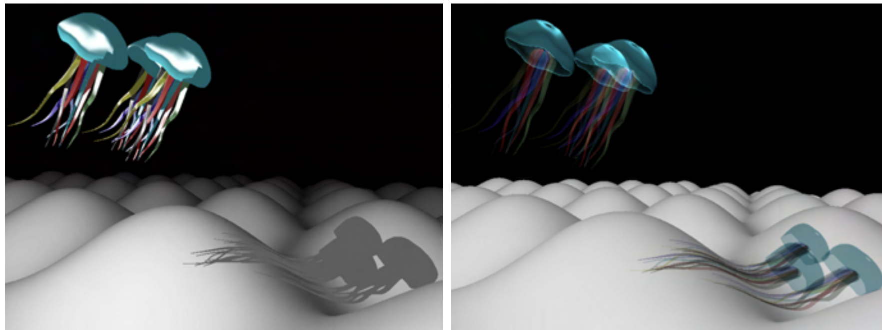

Figure 7. Images rendered by a traditional shadow volume algorithm (left) and by the new shadow volume algorithm (right).

believe that (3) is a viable option. In practice, when the opacity range of transparent objects are narrow, the image quality can be improved by scaling (3) so that the the logarithm of the transparency is more accurately represented in 16-bit floating point numbers. When we have objects with wide range of transparencies, a 32-bit floating points buffer can be used without the need of using logarithms.

4. Limitations on 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 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

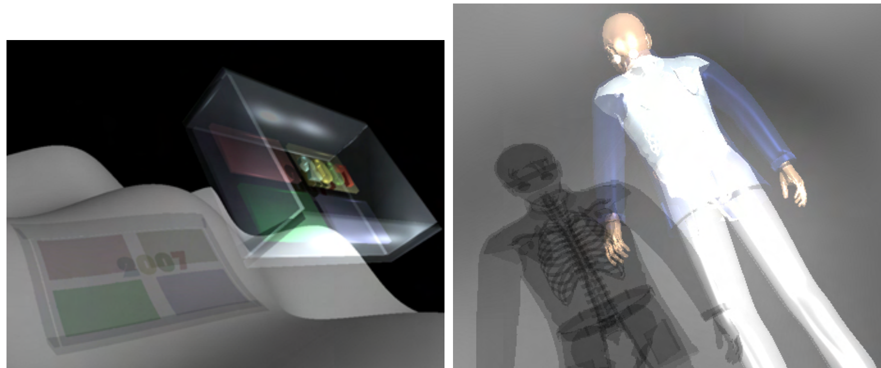

Figure 8. Left: by rendering the model once, we obtain partial self shadowing. See the shadow of letters at the bottom of the glass box. Right: by making object transparent and by rendering its shadow, the internal structure of a mesh can be visualized.

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 partially correct self-shadows are possible by computing self-shadows in a separate step, as shown in section 1.2.

5. 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 light caps and extruded polygons to build the light intensity map.
3. Using the light intensity map, render the receiver.
4. Render the transparent shadow casters.

Although 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 light caps and extruded polygons to obtain the intensity map on casters.
6. Using the intensity map on casters, render the transparent shadow casters.

Although the self shadowing is limited, this method increases the visual realism as shown in the left image of Fig. 8.

6. Results and Discussions

As shown in figures 1 and 7, our algorithm produces shadows of transparent objects. In the left image of Fig. 8, a partially correct self-shadow is shown. In the right image of Fig. 8, 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. 7) 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 frames/second for shadows of opaque casters (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.

References

[Aldridge and Woods 04] Graham Aldridge and Eric Woods. “Robust, geometry-independent shadow volumes.” In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pp. 250–253, 2004.

[Batagelo and Junior 99] Harlen Costa Batagelo and Ilaim Costa Junior. “Real-Time Shadow Generation Using BSP Trees and Stencil Buffers.” In *XII Brazilian Symposium on Computer Graphics and Image Processing*, pp. 93–102, 1999.

[Bergeron 86] Philippe Bergeron. “A General Version of Crow’s Shadow Volumes.” *IEEE Computer Graphics and Application* 1:1 (1986), 17–28.

[Crow 77] Frank Crow. “Shadow Algorithms for Computer Graphics.” In *Proceedings of SIGGRAPH*, 11, 11, pp. 242–248. New York, NY, USA: ACM Press, 1977.

[Everitt and Kilgard 03] Cass Everitt and Mark J. Kilgard. “Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering.” *http://developer.nvidia.com/object/robust_shadow_volumes.html*.

[Kilgard 99] Mark Kilgard. “Improving Shadows and Reflections via the Stencil Buffer.” In *Advanced OpenGL Game Development course notes Game developer Conference*, pp. 204–253, 1999.

[McCool 01] Michael D. McCool. “Shadow volume reconstruction from depth maps.” *ACM Trans. Graph.*, pp. 1–25.