Volume 0 (1981), Number 0 pp. 1–14

Rendering Soft Shadows using Multi-layered Shadow Fins

Xiao-Hua Cai[1], Yun-Tao Jia[1], Xi Wang[1], Shi-Min Hu[1] and Ralph R. Martin[2]

[1] Tsinghua University, Beijing, China
[2] Cardiff University, United Kingdom

Abstract

Generating soft shadows in real-time is difficult. Exact methods (such as ray tracing, and multiple light source simulation) are too slow, while approximate methods often over-estimate the umbra regions. In this paper, we introduce a new algorithm based on the shadow map method to quickly and accurately render soft shadows produced by a light source. Our method builds inner and outer translucent fins on objects to represent the penumbra area inside and outside hard shadows respectively. The fins are traced into multi-layered light space maps to store illuminance adjustment to shadows. The viewing space illuminance buffer is then calculated using those maps. Finally, by blending illuminance and shading, a scene with highly accurate soft shadow effects is produced. Our method does not suffer from umbra over-estimation. Physical relations between light, objects, and shadows demonstrate the soundness of our approach.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation: Bitmap and Frame Buffer Operations I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Shading, Shadowing

1. Introduction

Shadows provide important cues to spatial relationships in images, improving viewers' understanding [SUC95], so their computation has been widely studied. For interactive requirements, two widely used methods are those of *shadow volumes* [Cro77] and *shadow maps* [Wil78]. The basic ideas used by these approaches depend on point light sources only, and thus they generate *hard* shadows. Further work has been done to extend them to the generation of soft shadows, as we will discuss shortly.

On the other hand, for *soft* shadow rendering, various offline methods have been devised, such as *distributed ray tracing* [CPC84], *lazy visibility evaluation* [HDG99], and *light source sampling* [HH97]. However, computing accurate soft shadows is time consuming, and is still hard to do in real time.

Compared to the shadow volume method, the shadow map method is simpler and faster, and is not so sensitive to scene complexity. Thus, many methods based on this technology have been devised to approximate soft shadows, such as *plateaus* [Hai01], *penumbra maps* [WH03], and *smoothies* [CD03]. Although these methods have high performance, their results suffer from an umbra over-estimation problem, and can produce results far removed from reality. This problem becomes particularly evident when the light sources are of large extent. In this paper, we introduce a new algorithm based on the shadow map method to properly render soft shadows quickly. Our approach does not suffer from the umbra over-estimation problem. Furthermore, its computations are based on realistic physical relations between light, objects and shadows. Our approach assumes that the light source lies within a plane.

In Figure 1 compare images of the same scene generated using several methods. From left to right, the methods used are *hard* shadows, the *smoothies* method [CD03], our new method, and a *sampling* method (which combines hard shadows from 1024 point samples of the light source); the latter is used as a reference image.

As in the approaches of [WH03] and [CD03], our main idea is to initially construct additional external planar geometric primitives (called *shadow fins*, see Figure 4) attached to silhouette edges, and then generate shadows by considering the scene including the information from these fins.

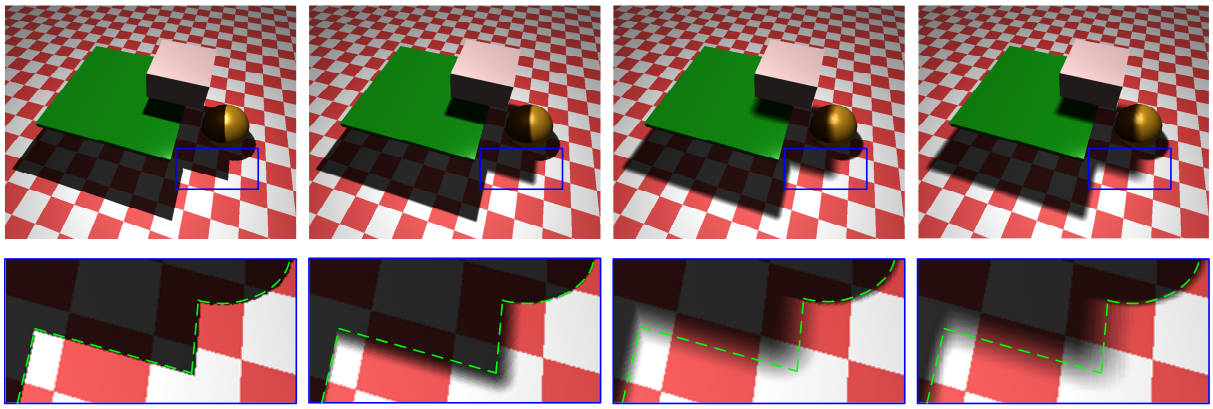

Figure 1: *Shadows produced by: hard shadows, smoothies, our new method, 1024 point sampling method.*

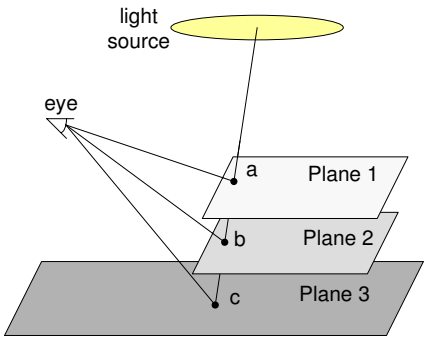

Figure 2: *Multiple objects along one beam of light*

Our method differs in that previous methods only used outward projecting fins, allowing them to take the outer penumbra into account, whereas we use fins which extend both inwards and outwards from the silhouette edges to simulate the penumbra area inside and outside hard shadows respectively.

The size of these primitives is based on geometric relations between the light source, the shadow casters, and the receivers. We accumulate the contributions due to these fins, per-pixel, into illuminance intensities in screen space, and then blend them with the corresponding shading values to obtain a final image with highly accurate soft shadows.

However, taking the inner penumbra into account is much more complicated here than in the cases handled by existing soft shadow methods based on the shadow map approach. This is because *several* different shadow states may need to be taken into account corresponding to a single beam of light, while a shadow map only stores a *single* value. Consider Figure 2. Points *a*, *b* and *c* on receiver planes 1, 2 and 3 correspond to one ray of light. A single stored value can only represent information from one of them, preventing soft shadows for such points from being rendered correctly.

To avoid this problem, we use a *multiple layer* approach, which partitions the constructed fins into multiple planar layers. These layers are parallel to the plane of the light source, and the extruded fins are then rendered into a penumbra map for each layer, which includes both depth and weight information (see Figure 3, left). Because each fin affects all layers below it, a given fin may need to be rendered several times during the whole algorithm. In order to avoid doing so explicitly, for efficiency, we implement this process in an accumulative manner—the penumbra map for each layer is calculated by combining information from the previous layer with the penumbra effects of any fins between these two layers. In this way, the complexity can be dramatically reduced.

In summary, the main advantages of our new soft-shadow approach are as follows:

1. It is a fast method based on shadow maps; its performance varies nearly linearly with scene complexity.
2. It generates shadows based on realistic physical relations, providing highly accurate sizes and shading for umbra and penumbra areas.
3. It uses an incremental method for penumbra map calculation, leading to real-time performance.

The next Section discusses related work, and overviews our approach. Section 3 explains the concepts in the context of a single-layered algorithm, then Section 4 gives our multi-layered algorithm. Section 5 discusses implementation issues, Section 6 gives some results, and Section 7 discusses them. Finally, we give our conclusions.

2. Previous work

Computation of shadows has a rich literature. Surveys such as those by Woo et al. [WPF90] and more lately Hansefratz et al. [HLHS03] discuss a wide range of methods.

Real-time computation of hard shadows is normally done using one of two basic methods: *shadow volumes* [Cro77]

or *shadow maps* [Wil78]. The shadow volume method constructs faces perpendicular to the light direction on silhouette edges. By counting the numbers of front and back faces which are traversed by any line of sight from the light source, one can decide whether the final receiver of the light is or is not inside the shadow region. Heidmann [Hei91] presented a hardware-based version of the method in 1991. The main drawback of the shadow volume method is its high sensitivity to geometric complexity of the scene.

The shadow map method renders objects using the light source as a view point to initially generate a depth map (the shadow map). As it proceeds, it compares the depth value of each fragment of the scene to the stored value in the shadow map to decide whether it is in shadow. Segal et al. [SKvW*92] gave a hardware accelerated algorithm for this approach in 1992. The main problem when using shadow maps is sampling aliasing which occurs at the edges of shadows. *Adaptive shadow maps* [FFBG01], *perspective shadows* [SD02], *shadow silhouette maps* [SCH03] and *alias-free shadow maps* [AL04] help to overcome this aliasing problem, but they all still generate hard shadows.

To produce accurate *soft* shadows, the computation is very costly. To overcome this, much work has been done on approximate approaches, such as *layered depth images* [ARHM00], *vertex tracing* [SR00], various *back projection* techniques [DF94], and so on. Soler and Sillion [SS98] introduced a method based on convolution in image space. In an ideal situation for their method, the light, the occluders, and the receivers lie on parallel planes, in which case they obtain good results. For other arrangements, they try to control the error. However, this method renders the scene layer by layer in such a way that it cannot cast shadows from a single object onto itself. (Our method also uses layers, but in a different way, so we do not suffer from this problem.) Such methods are too slow for real-time applications. Sloan et al. [SKS02] gave a sampling method for a low-spatial-frequency approximation of the lighting environment. This method can render objects and generate soft shadows in real-time, but it needs a lengthy precomputation to determine radiance transfer between objects, and cannot handle shadows in dynamic scenes where objects are changing. Furthermore, when the light sources degenerate to a single point source, the resulting shadow looks too soft.

Akenine-Möller et al. [AMA02] extended the shadow volume method to handle soft shadows. They replaced the traditional shadow volume with a four-faced penumbra wedge on silhouette edges, and computed every covered fragment to generate soft shadows. However, this method can only handle simple cases of silhouette edges as occluders. Because neighbouring faces of wedges can share side planes, their method suffers from a robustness problem when constructing wedges. They improved this method in [AAM03], where the wedges for the silhouette edges are now independent and a precomputed four-dimensional texture is used to represent light visibility factors. This no longer suffers from the robustness problem, has enhanced performance, and works for any kind of light source. However, they need to construct wedges and compute every fragment covered by each wedge, so their method requires a high rate of data transfer between the CPU and graphics processor (GPU).

There are many methods based on the shadow map technique. Reeves et al. [RSC87] give a subsampling technique for blurring the shadow map. However, it only improves the blurring effect at the edges of the original hard shadows, and does not produce real soft shadows. Heidrich et al. [HBS00] handle linear light sources by considering their two end points, but their method does not achieve real-time performance. Braded and Seidel [BS02] extended the idea to handle area sources using simple sampling, but this method cannot produce self-shadows and seems only suitable for low-resolution shadow maps.

Combining the ideas of shadow maps and shadow volumes, Haines presented the *plateau method* [Hai01], which renders the soft shadow as a texture. It uses hard shadows to initially approximate the umbra, then constructs cones (from terminal points of silhouette edges down to the receivers) and sheets (connecting neighbouring cones). Attenuation is controlled by these primitives when light is cast to generate the penumbra. However, by only taking into account the distances from casters to the receiver, and not the size and the position of the light source, the penumbra generated is not physically accurate. Moreover, the receivers are restricted to being planar.

Wyman and Hansen [WH03] introduced the *penumbra map* method based on plateaus. They also construct cones and sheets, and render them as a texture. Because they take the light source size and spatial configuration of silhouette edges into account, as well as computing various refinements at the fragment determination stage, they do not have the planar receiver limitation. Furthermore, the penumbra they generate is more physically accurate.

Chan et al.'s method [CD03] is similar to the previous one. It uses hard shadows to approximate the umbra as above, and then attaches geometry primitives, called *smoothies*, comprising rectangles and corner pieces, to silhouette edges. By rendering such geometry, they generate the penumbra too. However, this method only considers the spatial configuration of silhouette edges, not the size of the light source and its direction, so it produces fake penumbras. Instead of taking into account details of the light source, it uses a single global parameter to control the size of the smoothies to simulate the effect of changes in size of the light source.

The last three methods: plateaus, penumbra maps, and smoothies, are all real-time approaches. However, because they use a hard shadow to approximate the umbra, then generate the penumbra only outside the umbra, all three overestimate the umbra, and cannot generate the shrinking effect

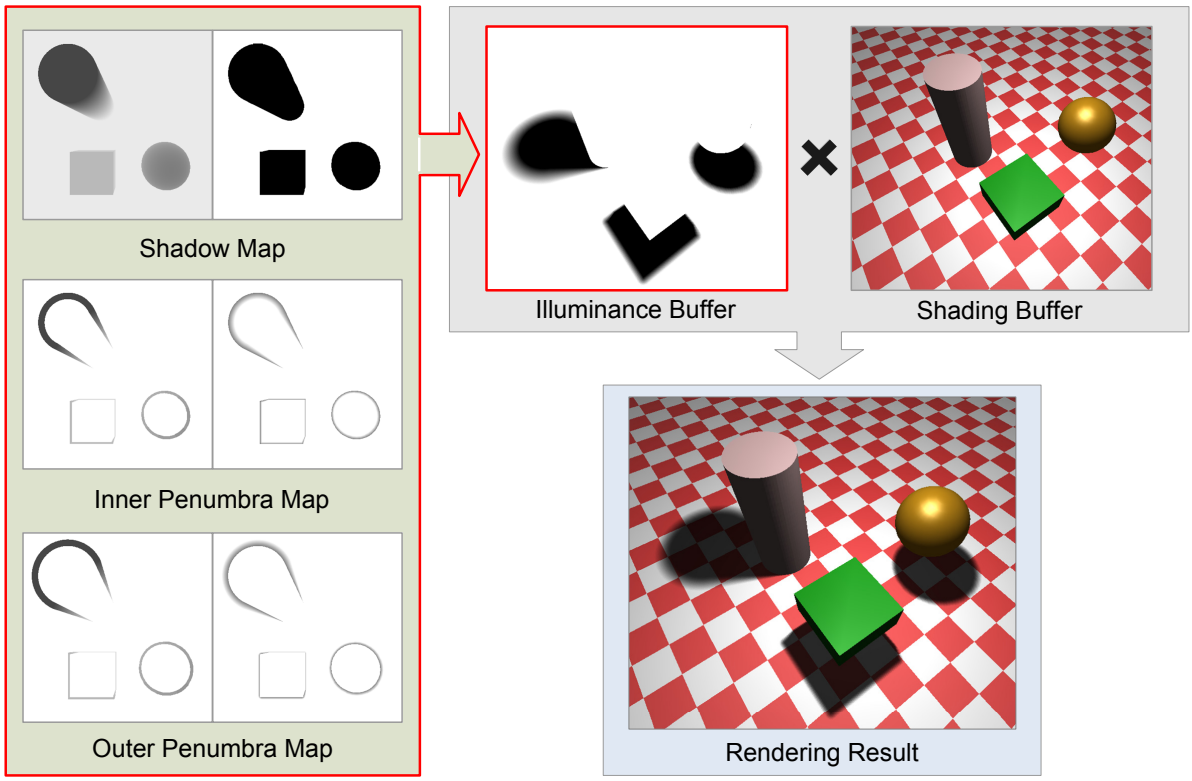

Figure 3: *The single-layered method. Left (light space): depths and weights of the shadow map, inner penumbra map, and outer penumbra map. Top right (screen space): illuminance buffer and shading buffer. Bottom right (screen space): resulting image.*

seen with a real penumbra. Thus, they all generate quite unrealistic soft shadows. If the light source is large, the poor results are evident.

One final method, based on ray tracing, by Parker et al. [PSS98], deserves to be mentioned here. They propose the concept of *soft-edge objects*, whereby objects are shrunk or magnified to approximately produce soft shadows. The amount of shrinking or magnification is decided by the size of the light source and the position of the silhouette of the object, so it is basically physically accurate. But in order to prevent light leakage between two objects which are placed in contact or close together (due to shrinkage [PSS98]), in practice, objects cannot be shrunk. This leads to this method over-estimating the umbra too, failing to simulate the smaller umbra when a penumbra is present. Furthermore, ray tracing is not suitable for real-time processing of large models.

Overall, although many methods have been devised for soft shadow generation, current approaches lack the ability to produce physically realistic results in real-time. Our approach uses shadow fins and a multi-layered approach to produce approximate shadows of high quality quickly.

3. Single-layered algorithm

In this section, we first explain some basic ideas using a *single-layered* algorithm, upon which our *multi-layered* algorithm is built.

The shape of the light source is assumed to be planar and circular in our approach. This is not as serious a restriction as it sounds: it is difficult for people to tell the shape of a light source from the shapes of soft shadows [Wan00]. We do so as such a shape makes it easier for us to solve the problem. We use the centre of the light source to generate hard shadows, and all silhouettes are computed relative to this point.

The dataflow of our single-layered algorithm is shown in Figure 3 (from left, to top right, to bottom right). The algorithm has five steps which are carried out in object space, then light space, and finally, image space:

- In object space:
  - construct inner and outer shadow fins from geometric primitives, and compute weights associated with each
- In light space:
  - render the shadow map

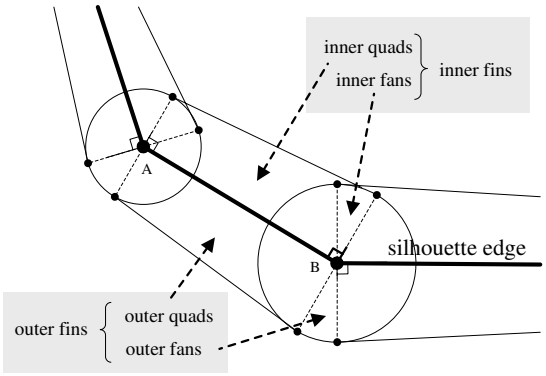

Figure 4: *Inner and outer fins*

- render inner and outer penumbra maps

- In screen space:
  - compute the illuminance buffer from the above maps; compute the shading buffer
  - render the final image

See Figure 3. Firstly, we compute the depths and weights for the shadow map, the inner penumbra map and the outer penumbra map. These 3 map images are generated with respect to the centre of the light source. Then, with respect to the eye point, we combine these maps into an *illuminance buffer*. Lastly we take the shading of the scene into account, to generate the final image with soft shadows.

3.1. Constructing shadow fins

Following the idea of *soft-edged objects* from [PSS98], we attach inward and outward planar geometric primitives to the silhouette edges of objects, to enable computation of the penumbra (see Figure 4). We call these *inner fins* and *outer fins* respectively.

Assuming, the object is facetted, *silhouette* edges are those edges which belong to two faces, one facing towards the light, and the other facing away. There are many silhouette detection algorithms, but we simply use a brute force approach, considering all edges and marking the silhouette edges.

The inner and outer fins consist of quadrilaterals and fans. Given a silhouette edge, lines are drawn outwards and inwards from the vertices at each end; these lines are perpendicular to the edge, lying in a plane parallel to the light source plane. The two inner vertices and the ones from the original edge form an inner quad, while the outer vertices and the original ones form an outer quad. Inner fans and outer fans forming circular sectors are added to connect inner and outer quads as necessary, as shown in Figure 4.

The size of the fins must depend on the location of the *receiver* where light ultimately falls in the scene after going past a silhouette edge. Because we do not yet have information about where the receivers are, we use a reference plane as a substitute. In practice, we use the *far plane* of the frustum viewed from the light source. In a later step, we use a refinement mechanism to improve the result computed using the reference plane, as will be explained in Section 3.4.

As can be seen in Figure 4, adjacent fins may overlap. We accept such cases and skip the complex computations needed to trim the fins. Instead, we control the effects computed for each fin later in such a way as to ensure a smooth variation of shadow corresponding to the overlap region.

If more than two silhouette edges meet at a vertex, we use Chan's suggestion in [CD03] to handle this special case. First, we average the adjacent face normals to obtain a shared vertex normal $\mathbf{n}$ whose projection in the screen space of the light source has length l. Then we draw triangles that connect $\mathbf{n}$ with each of the adjacent shadow fin edges.

3.2. Rendering the shadow map

We now build a shadow map by projection from the centre of the light source—this gives the hard shadow. This is done in a standard way, as described in [Wil78].

3.3. Rendering penumbra maps

Next, still starting from the centre of the light source, we compute the penumbra maps of the inner and outer fins which are used to simulate soft shadows.

Each *penumbra map* (inner or outer) is a combination of a depth map and a weight map, as shown in Figure 3—on the left side of each penumbra map is the depth map, while the right side shows the weight map. The depth map contains depth values of the appropriate (inner or outer) fins relative to the light source and is used later for soft shadow refinement. The weight map contains values used to simulate the effects of soft shadows, computed as follows.

The inner fins and outer fins of each silhouette edge are treated in a similar way, except that outer fins are used to add shadow, and inner fins are used to remove shadow, relative to the existing hard shadow. A smoothly varying shadow strength weight is used varying from 0.5 at the silhouette edge to 0 at the edge of the fin. The weights indicate the *magnitude* of illuminance adjustment (made to the shadows) at that point of the fin, the *sign* is decided by whether the fin is an inner fin or outer fin. Thus the weight is

$$s = 0.5(1-t) \tag{1}$$

where t is the fractional distance of the point from the silhouette edge (or vertex for fan) to the edge of the fin.

When producing the weight maps, if multiple fins overlap, we need to define special rules to combine the weights

from the multiple fins. As pointed out by Park [PSS98], overlapping shadows require three types of computation: addition, multiplication, and thresholding. However, distinguishing these cases and treating them correctly is difficult. Instead, we use approximate rules based on thresholding:

- for inner fins, use the minimum of their individual weights,
- for outer fins, use the maximum of their individual weights.

While ignoring fins inside hard shadows leads to errors, we overcome this problem to some extent by using multiple layers in our improved algorithm, which we describe in Section 4.

3.4. Computing the illuminance buffer

In this step, the depth and weight parts of the *shadow map*, and the inner and the outer *penumbra maps*, are combined to give an *illuminance buffer*, whose values represent how much each part is illuminated. Unlike earlier computations, this buffer is generated with respect to the eye point, not with respect to the light source; its size is the same as the output screen resolution.

Viewed from the eye point, firstly, we classify all screen fragments into two categories: shadow regions (from which the light source *cannot* be seen) and lit regions (from which the light source *can* be seen), by comparing their depths to the value stored in the depth map of our *shadow map* (just as in the original *shadow map* method), but here we assign 0 to the hard shadow region and 1 to the lit region.

Then, for each screen fragment, the corresponding values in the weight maps of the inner and outer *penumbra maps* are used to make adjustments for that fragment. Because inner fins remove shadow and outer fins add shadow, we adjust the shadow value as follows:

- for inner fins, we add its (refined—see below) weight;
- for outer fins, we subtract its (refined) weight.

If a fragment has weights corresponding to both inner fins and outer fins, this means the different fins overlap, and we simply apply their individual effects to ensure that no light leakage occurs, as illustrated later in Figure 10. However, this approximation introduces some artifacts which can be seen in that example; these will be discussed in Section 7.2.

The result of this step is a map from the eye point, storing real values in the range 0 to 1, representing how much each part is in shadow.

In practice, the simple idea above is modified slightly to produce improved results. Before accumulation of the values in the penumbra maps, we *refine* them to satisfy more accurately the geometric relations between shadows, the light source, and silhouette edges. When the fins were constructed, we did not know the position of the receiver

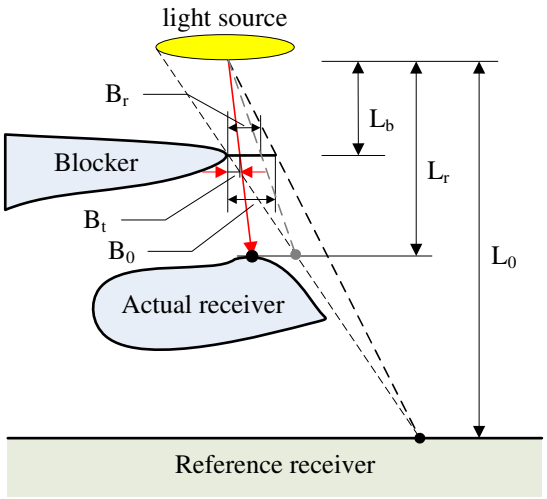

Figure 5: *Refining shadow values*

upon which light would fall during rendering, so for each silhouette edge, we used an arbitrary size for the width of the fins, computed using a reference receiver plane, as suggested by [WH03]. This reference plane is the far plane of the viewing frustum with respect to the centre of the light source. When computing the *illuminance buffer*, we now have actual depth information available about the real receiver, so we can recompute the results with more accurate fin widths to produce a more accurate shadow value.

Furthermore, real shadows do not fall off linearly, and they can be better modelled using a Bernstein interpolant [Hai01]. This correction is also performed as the penumbra maps are used.

For an actual receiver plane, the size of each fin can be computed using the following equation (see [PSS98]):

$$B_r = \frac{(D_r - D_b)R}{D_r}. \tag{2}$$

B_r is the size of the fin corresponding to the receiver r, D_r is the distance from the centre of the light to the receiver point, D_b is the distance from the centre of the light to the silhouette edge of the shadow caster, and R is the radius of the light source.

Because the (planar) fins are parallel to the light source plane, if the receiver is a plane parallel to the light source too, then Equation 2 can be rewritten as

$$B_r = \frac{(L_r - L_b)R}{L_r} \tag{3}$$

where L_r is the distance from the light source plane to the receiver plane and L_b is the distance from the light source plane to the fin plane.

Let B_t be the distance from the intersect point of the fin

and one ray of light to the silhouette edge of that fin. Then the fractional distance is B_t/B_r. Substituting the value of t from Equation 1, we get

$$s_r = \frac{1-t_r}{2} = \frac{B_r - B_t}{2B_r} \tag{4}$$

where s_r is the weight corresponding to the actual receiver.

Consider Figure 5. We wish to determine the weight s_r for the actual receiver, but we have the weight s_0 for the reference receiver before refinement, where the subscript $_0$ denotes the value for the reference plane. We can make the adjustment using:

$$\begin{aligned} s_0 &= (1-t_0)/2 = (B_0 - B_t)/2B_0, \\ s_r &= (1-t_r)/2 = (B_r - B_t)/2B_r \end{aligned}$$

where

$$B_0 = \frac{(L_0 - L_b)R}{L_0}, \qquad B_r = \frac{(L_r - L_b)R}{L_r}.$$

So,

$$s_r = \frac{1}{2} + \frac{B_0}{B_r}(s_0 - \frac{1}{2}). \tag{5}$$

In summary, when creating the fins, we use the reference plane as a receiver to determine the size of the fins, and to assign weight s_0 for fins. When computing the *illuminance buffer*, which is a per-fragment operation, we can obtain the exact receiver point of every fragment, and so the more accurate weight s_r from Equation 5 are used. The depth maps belonging to the *penumbra maps* store L_0, and are used here to compute B_0 and B_r.

3.5. Rendering the final image

Generating the final output image is accomplished by simply multiplying the image without shadows by the *illuminance buffer* and then adding the ambient colour.

4. Multi-layered algorithm

The single-layered algorithm is reasonably effective, but as mentioned before, it is not sophisticated enough to handle cases with *several* different values of shadow status along one ray of light. This will lead to errors if we only use one layer to compute the relevant shadow. An example can be seen later in Figure 7. The fins of the two-holed torus cast shadows down onto the green plane, correctly. But if we only use the weights from the two-holed torus to adjust the shadows on the ground plane, we will get very obvious errors, as the weights from the green plane should have been used instead.

To tackle this problem, we now explain an improved version of our algorithm, based on the use of *multiple layers*, each parallel to the plane of the light source. This idea is similar to that used by Soler and Sillion [SS98], although they used a convolution method which performed computations in light space. Instead, we use a *shadow map* and construct fins, which can be seen as performing convolution in object space.

Unlike the single-layered algorithm, the *multi-layered algorithm* needs to allocate inner and outer fins into appropriate layers to eliminate overlap cases. Then, we generate the *illuminance buffer* for each layer in an incremental manner, from near to far in the light space.

Following the discussion above, we again consider three steps of the revised algorithm in turn: *layering shadow fins*, computing the *illuminance buffer*, and *rendering* the final image.

4.1. Layering shadow fins

We create shadow fins using the same method as in the single-layered algorithm, and a small number of extra operations are used to allocate the fins to layers.

Sorting or splitting operations on shadow fins are avoided in our approach, by use of a series of parallel clipping planes to assign each fin to a particular layer. Our allocation method only needs to know the distances of these successive clipping planes from the light source plane. As the shadow fins are attached to silhouette edges, we simply use the centre position of each silhouette edge to determine the layer for each fin.

We try to ensure an approximately equal number of silhouette edges in each layer to minimise the number of times fins from the same layer may overlap each other. This is done as follows. In light space, we first build a set of equally spaced sub-layers, for example 1024 of them, and ultimately choose a final set of layers (for example 16 layers) from these sub-layers in an unequally spaced way. We count the number of silhouette edges in each sub-layer, and use these counts to merge the sub-layers into the final layers.

In general, overlaps will become less frequent as we increase the number of layers. However, increasing the number of layers also increases the workload for the rendering pipeline. Therefore, we choose the number of layers to trade off accuracy with performance. Typically, we use 16 layers.

4.2. Computing the illuminance buffer

In the single-layered algorithm, we only compute one *illuminance buffer*, which is a map of shadow values from 0–1 representing how much each position is in shadow. In the multiple-layer algorithm, we compute an illuminance buffer for *each* layer.

A simple implementation of this idea is to compute the information for each layer directly using the method described for the single-layered algorithm. However, because each fin casts shadows on the *penumbra maps* of all layers which are below this fin, for each fragment, we in fact need to consider

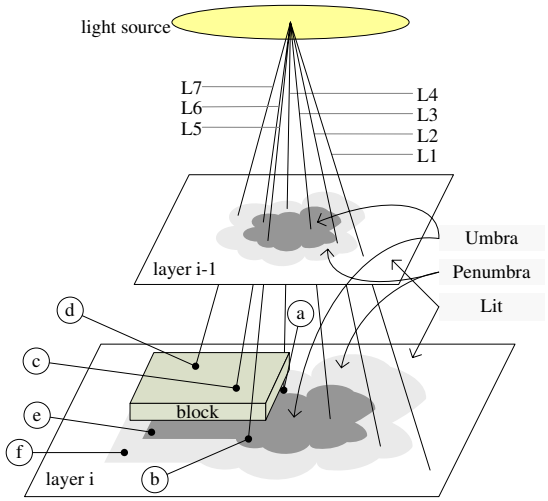

Figure 6: *Incremental computation of illuminance buffer*

not only the layer in which the fragment is located, but also the layers above it. This will result in a complex calculation when rendering the final image.

For efficiency, we do not compute the *penumbra maps* of each layer by considering all fins above it. Instead, we first use the shadow fins belonging to the current layer to compute the *penumbra maps*, and then combine this with the *illuminance buffer* of the previous layers *incrementally*. Thus, the information in the illuminance buffer for each layer provides the details of the shadows in that layer directly.

Combination of the *illuminance buffers* belonging to two layers is performed using the following rules according to the region being considered in the current layer. The value in the *illuminance buffer* is taken as follows:

- *Lit region*: use the previous layer's illuminance;
- *Umbra region*: use this layer's illuminance;
- *Penumbra region*: use the minimum illuminance of the two layers.

Consider Figure 6. Layer i shows the *illuminance buffer* for the current layer, while layer $i-1$ shows it for the previous layer. The white region in both planes represents the lit area, the dark grey region represents the umbra area, and the light grey region represents the penumbra area. The *block* is representative of the shapes contained in the current layer.

We compute the weights for the *penumbra map* of the *block* (in the current layer) first, and then combine the results for the *block* with the previous layer's *illuminance buffer*. In Figure 6, rays L_1, L_2, and L_3 intersect the current i^{th} layer in the lit region, so we simply copy the illuminance value from the $i-1^{th}$ layer. Rays L_6 and L_7 intersect the i^{th} layer inside the umbra because of the block, so values from the previous layer are not used. Rays L_4 and L_5 intersect the

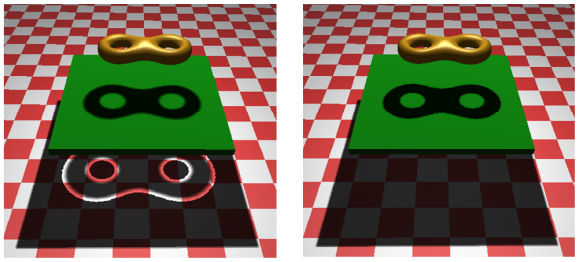

Figure 7: *Comparison: (left) result of single-layer algorithm; (right) result of multi-layer algorithm.*

current layer inside penumbra regions, so illuminance values are combined with those from the previous layer: point a gets its illuminance from layer i, and point b gets its illuminance from layer $i-1$. Points c and d are located on the block, in umbra, so their illuminance values are both 0. Points e and f contain newly cast shadow information due to the effect of the block shadowing the i^{th} layer.

We present performance data for this incremental approach and compare it to the *smoothies* method in Section 7.1.

4.3. Rendering the final image

Because the computed *illuminance buffer* for a given layer gives the shadow situation for that layer directly, it is simple to render the final image by multiplying these buffers with the unshadowed scene image and then adding the ambient colour. So, we render the scene layer by layer—triangles belonging to each layer are rendered using appropriate information from that layer; clipping plane techniques are used to select these triangles.

5. Implementation

We have implemented our algorithm using OpenGL, and have used GPU hardware to accelerate the rendering part. Finding silhouette edges and constructing shadow fins are mainly done by the CPU, as are the allocation computations (recording positions of the clipping planes) in the multi-layer approach. Other operations are performed by the GPU. We use `ARB_VERTEX_PROGRAM` and `ARB_FRAGMENT_PROGRAM` as the vertex shader and pixel shader during rendering. When generating the inner and outer weights, we use `EXT_BLEND_MINMAX`. The brute force silhouette computation takes about 1/4 of the total computation time. Using a quicker silhouette edge detection algorithm could make a small improvement to the time taken by our method.

We use OpenGL *extra clip planes* to rapidly partition the sub-layers into several layers rather than using algorithmic methods to sort or and split fins or scene triangles. Assuming

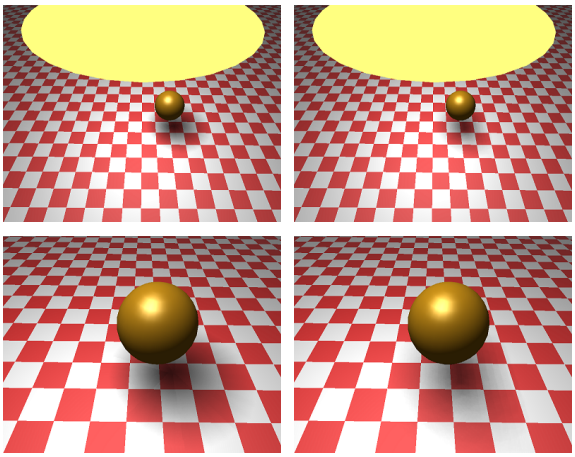

Figure 8: *Shadow with large light: (left) our rendering result; (right) ground truth. (Below): close-up.)*

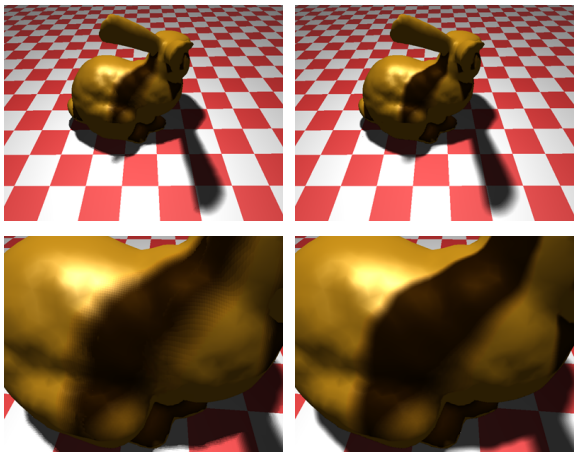

Figure 9: *Shadow with non-parallel receiver: (left) our rendering result; (right) ground truth. (Below): close-up.*

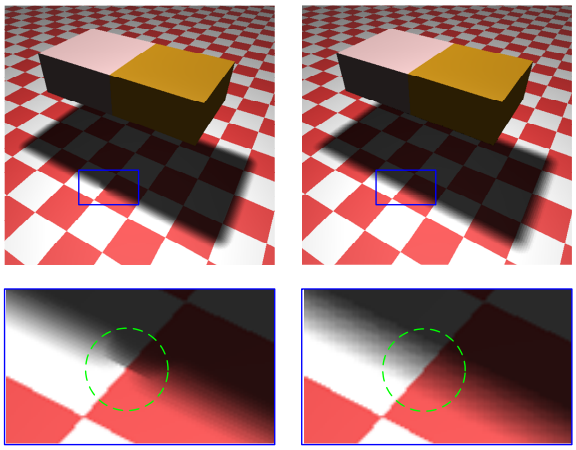

Figure 10: *Shadows for nearby objects: (above left) rendered by our method, (above right) rendered by 1024 point light source sampling; (below) artifacts arising in our method due to overlap.*

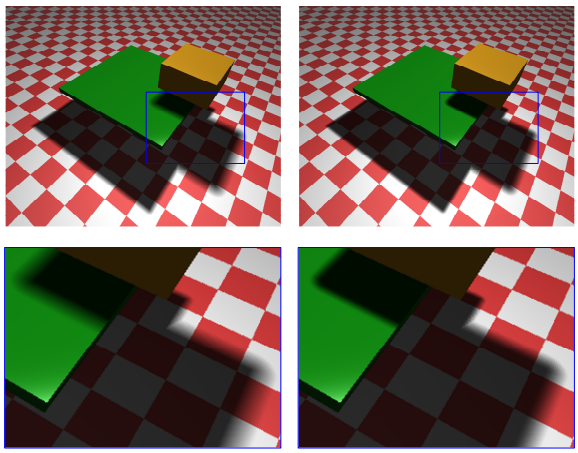

Figure 11: *Effects of shadow fin refinement: (left) without refinement; (right) with refinement.*

that we have chosen to use n layers, then the number of triangles t to be processed will be multiplied n times. Note that this extra processing is only done by the vertex shader—after this step the clip plane stage produces a single layer for later stages of processing. Thus, the pixel shader only processes a single layer, not n layers, so the number of triangles handled by the pixel shader using our algorithm is still t: this step is rapid. it is important to note that in our approach, the vertex shader is simple, while the pixel shader processing is complex, leading to a highly efficient algorithm overall. Thus, while our algorithm is slower than a typical (hard) shadow map program, it still can achieve real-time performance.

6. Results

For the tests described here, we used a Pentium 2.4GHz with 256Mb main memory and an ATI RADON 9800Pro graphics card with 128Mb of memory.

As noted earlier, Figure 7 shows a problem caused by using the single layer version of the algorithm, where shadow information goes through to the ground. It also shows the result produced by the multi-layered version, where the light is correctly blocked by the occluder. Indeed for more complex scenes, more layers are needed, and more time will be taken for rendering. If insufficient layers are used, problems similar to those in Figure 7(left) will occur. In practice, all

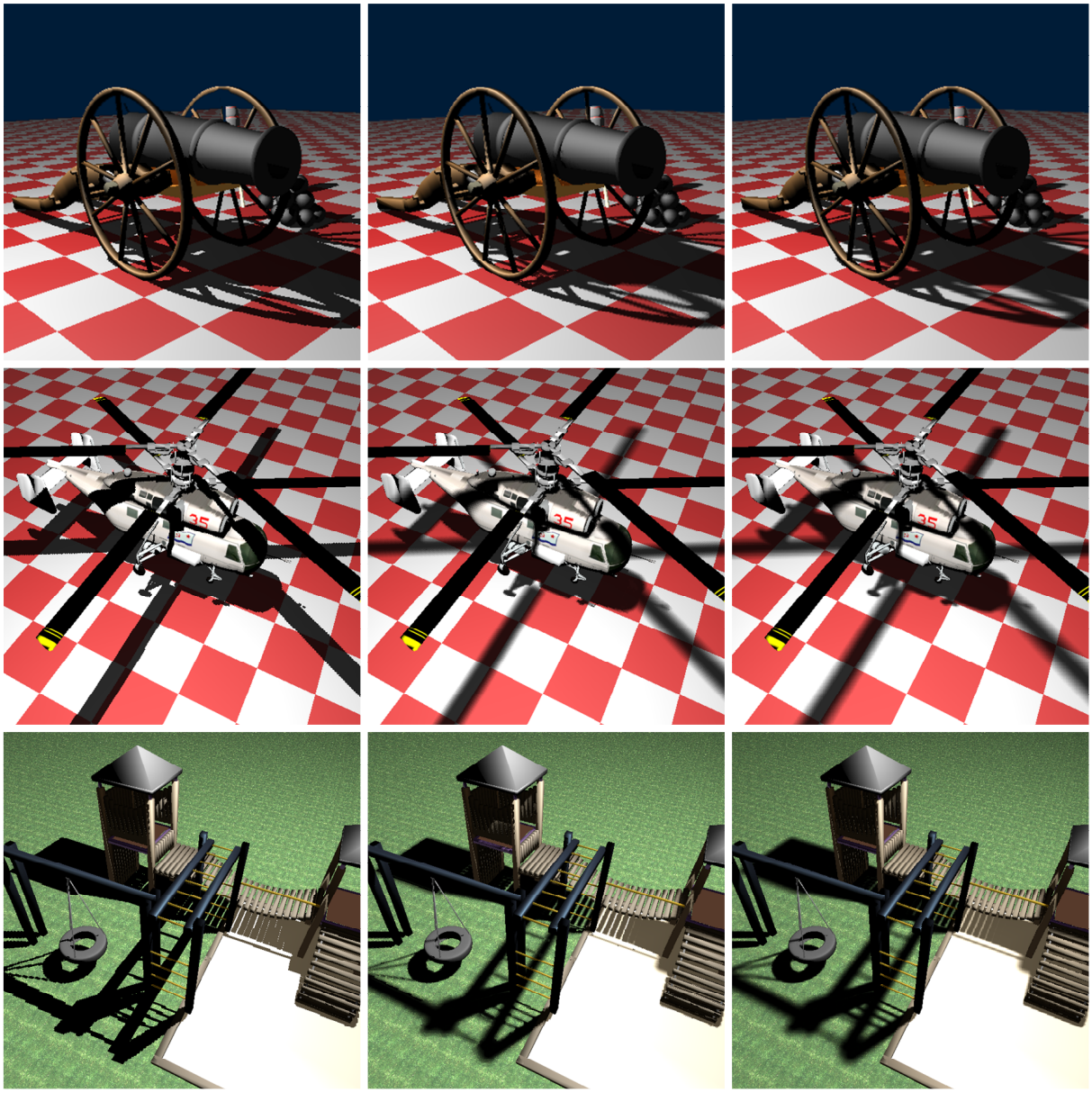

Figure 12: *Complex scenes, rendered using hard shadows, our method, and 1024 point light source sampling respectively.*

scenes used for testing can be rendered well using no more than 16 layers.

Figure 8 is a scene to illustrate that our shadow algorithm is able to handle the case when the light source is large relative to the occluder. Figure 9 shows shadows generated for a receiver not parallel to the light source plane; again good results are obtained.

Figure 10 shows shadows generated for two boxes in close proximity. Note that there is no light leakage into the shadow area where they nearly meet. However, there *are* obvious visible artifacts in the regions near the ends of the overlap of their shadows. We discuss these cause of such artifacts in Section 7.2.

Figure 11 shows the improvements produced by the refinement step of our method. After refinement (right), the shadow falling on the green plane is more realistic.

Figure 12 provides three sets of comparative results for complex models. In each case the three images compare

results produced using hard shadows, our method, and the combination of hard shadows from 1024 point samples to give a reference image. There are 24366, 8184, and 9135 triangles in the cannon scene, the helicopter scene, and the playground scene respectively. These scenes can be displayed at 7, 15 and 14 frames per second respectively. All were rendered using 16 layers.

7. Discussion

We now discuss various issues concerning our new algorithm, in particular a comparison with [CD03], aliasing arising in our method, and rendering complexity.

7.1. Comparison to smoothies

Our algorithm has many similarities to the *smoothies* method [CD03]—for example our outer fins are similar to smoothies. Both start from Park's *soft object* idea [PSS98], and both combine Soler and Sillion's [SS98] convolution approach.

However, there are important differences between their method and ours:

- They only construct *outward* primitives (smoothies) on objects, omitting the inner penumbra, so they suffer from umbra over-estimation. We create both inner fins and outer fins on objects, which overcomes the umbra over-estimation problem. However, we have to introduce multiple layers to tackle complex cases involving inner penumbras.
- They do not consider the effect of the size of the light source when constructing smoothies, but instead use a single global control parameter to vary the size of the shadows. Poor choice of this parameter relative to the size of the light source leads to inappropriate shadows. Their shadows are not based on correct geometry, while ours truly reflect the size of the light source.

Our algorithm is more complex than the *smoothies* approach, and so takes more time. We construct twice as many fins compared to *smoothies*. However, this does not greatly affect the performance, because the number of silhouette edges is insignificant compared to the total number of edges in the scene. During vertex shader computations, our computation time is approximately n times that used by *smoothies*: our method uses n layers, where each layer represents the whole scene. During pixel shader computations, the number of triangles handled is about the same as the number in the original scene (we have a few inner fins extra), while we have two more operations to perform: combination and refinement. Because the pixel shader has a greater effect on the efficiency of the algorithm than the vertex shader, our algorithm overall takes about 3–4 times longer than *smoothies*. When rendering the scenes in Figure 12, our algorithm can do so at 7, 15 and 14 frames per second, while *smoothies* can do so at 25, 61 and 55 frames per second.

7.2. Shadow overlap aliasing

Given two adjacent objects, our algorithm does not leak light (see Figure 10). However, there *are* obvious visible artifacts in the regions near the ends of the overlap of their shadows, caused by improper handling of the shadow overlap case.

While in most cases using max and min operations to combine shadows gives essentially correct results, in more complex cases, addition may be required, as discussed in Sections 3.3 and 3.4. In the particular case illustrated, the artefacts occur in the hard shadows of the boxes where three fins overlap: two inner quads and one outer quad (for simplicity we ignore the fans here). Let s_a and s_b indicate the inner fins' parameters, and s_c indicate the outer fin's parameter. In this case, they should be combined using

$$s_a + s_b + s_c \tag{6}$$

whereas our rules use

$$\min(s_a, s_b) + s_c. \tag{7}$$

This causes the computed shadow to be darker than the correct result. This happens because we consider inner fins and outer fins separately in two passes. We have no simple answer to this issue—we cannot perform the correct computation without modifying our two pass methodology. Furthermore, correctly identifying and processing all such special cases would be very time consuming. However, in justification of our approach, we note that in general scenes, such cases are uncommon, and occupy relatively little area on the screen.

7.3. Shadow fin aliasing

Figure 4 shows shadow fin construction for silhouettes. For most cases, it works very well. Where two silhouette edges meet at a very sharp angle, the shadow fins constructed cause aliasing as shown in Figure 13: there is a soft shadow distortion (see inside the blue circles). This is caused by the structure of the shadow fins, and is a general problem for all methods which construct additional primitives in this kind of way. Clearly, the shadow intensity at the corner (the sharp intersection point of two silhouettes) should be larger than 0.5, as more than half of the light can be seen from there. The smaller the angle is, the larger the shadow illuminance should be at the corner. However, as we place the ends of the shadow quads (inner quads and outer quads) at the end point of the silhouette edge, the maximum shadow illuminance inside the quads can be no greater than 0.5 (from Equation 1), leading to a soft shadow discontinuity between the shadow quads and the shadow fans; this problem will be larger with sharper angles.

One solution to this issue is to adjust the position of the end point of the shadow quads in a manner proportional to the size of the light, and to add another primitive between the

Model (Polygons)	Our algorithm (fps)	Shadow map method (fps)	1024 light samples (fps)
984	33.12	350.0	1/1.765
1536	31.68	300.2	1/2.094
3516	29.12	200.6	1/4.688
6620	23.84	120.3	1/6.171
7172	19.36	95.21	1/7.031
9906	14.08	54.80	1/15.47

Table 1: *Rendering speed (frames per second) for various models, using different algorithms.*

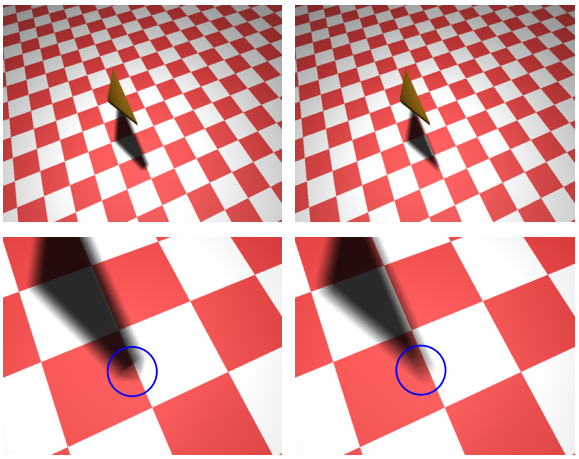

Figure 13: *Shadow fin aliasing for sharp angles. (Left) rendered by our algorithm; (right) rendered by 1024 point light source sampling.*

shadow quads and the fans. This will make shadow fins construction and computation more complicated. Again, however, in many real scenes, such cases will be rare, and occupy little screen area, and the extra complexity is probably not worthwhile.

7.4. Rendering complexity

Table 1 gives the rendering times for various dynamic scenes, using our algorithm (16 layers), the shadow map method, and 1024 light samples as ground truth. All shadow maps used were of size 512×512. Given a fixed number of layers, our algorithm exhibits nearly linear complexity in the number of polygons, with a good fit to

$$t = 22.69 + n/238.72\text{ms} \quad (8)$$

where t is rendering time, and n is the number of polygons in the model.

In more detail, the rendering complexity of our algorithm depends on the following factors:

- *Shadow map generation* Our method is a shadow map based method. This part is very fast and its computation time is independent of scene complexity.
- *Silhouette edge detection and shadow fin construction* Like shadow volume methods, we have to detect silhouette edges and construct fins as the scene dynamically changes. For each silhouette edge, we only have to construct two fins, and the number of silhouette edges is relative small compared to the total number of polygons. This part is also fast, taking about 10% of the total rendering time.
- *Multiple layers* The more layers used, the longer the time taken to render the scene. However, as explained in Section 5, the main work of the shadow computation is executed by the pixel shader, and this does not increase much as the number of layers increases. Overall, for example, when rendering a scene using 16 layers, the complexity will be 3-4 times higher than when using a single layer, which can still gives real-time performance.

8. Conclusions and future work

In this paper, we have provided a new algorithm to approximate soft shadows. Our method is based on creating shadow fins, and uses an incremental multi-layer approach to shadow computation. Our contribution has the following attributes:

- It is a fast soft-shadow method based on shadow maps; its performance varies nearly linearly with scene complexity.
- It generates approximate shadows based on accurate geometric relations, leading to shadows of generally accurate size and intensity in both umbra and penumbra areas.
- It uses an incremental method for penumbra map calculations, to keep rendering overheads low.

Various future work remains to be done:

- *Adaptive partitioning* Our current implementation uses a fixed number of partitioning layers, and uses the midpoints of silhouette edges to assign them to layers. Layers are selected to give layers of equal complexity. However, adaptively determining the required number of layers, and using a more sophisticated layer assignment algorithm, should give better results.

- *Shadow overlap artefacts* Overcoming shadow overlap artefacts, or minimising their visual effects, requires further study.
- *Construction of fins* Presently, we use a simple method to construct shadow fins, and we ignore overlaps of adjacent fins. More sophisticated geometric offsetting methods should be considered.

Acknowledgements

This work was supported by the Natural Science Foundation of China (Project Number 60225016, 60321002), and the National Basic Research Project of China (Project Number 2002CB312101). This work was done in Key Lab of Pervasive Computing, Ministry of Education, China.

References

[AAM03] ASSARSSON U., AKENINE-MÖLLER T.: A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Computer Graphics (Proc. of SIGGRAPH '03) 26*, 2 (2003), 511–520.

[AL04] AILA T., LAINE S.: Alias-free shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2004* (2004), Eurographics Association, pp. 161–166.

[AMA02] AKENINE-MÖLLER T., ASSARSSON U.: Approximate soft shadows on arbitrary surfaces using penumbra wedges. *Proceedings of the 2002 Eurographics Symposium on Rendering* (2002), 309–318.

[ARHM00] AGRAWALA M., RAMAMOORTHI R., HEIRICH A., MOLL L.: Efficient image-based methods for rendering soft shadows. *ACM Computer Graphics (Proc. of SIGGRAPH '00)* (2000), 375–384.

[BS02] BRABEC S., SEIDEL H.-P.: Single sample soft shadows using depth maps. *In Proceedings of Graphics Interface* (2002), 219–228.

[CD03] CHAN E., DURAND F.: Rendering fake soft shadows with smoothies. *Proceedings of the 2003 Eurographics Symposium on Rendering* (2003), 208–218.

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. *Computer Graphics 18*, 3 (1984), 137–145.

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. *ACM Computer Graphics (Proc. of SIGGRAPH '77) 11*, 2 (1977), 242–248.

[DF94] DRETTAKIS G., FIUME E.: A fast shadow algorithm for area light sources using backprojection. *ACM Computer Graphics (Proc. of SIGGRAPH '94)* (1994), 223–230.

[FFBG01] FERNANDO R., FERNANDEZ S., BALA K., GREENBERG D. P.: Adaptive shadow maps. *ACM Computer Graphics (Proc. of SIGGRAPH '01)* (2001), 387–390.

[Hai01] HAINES E.: Soft planar shadows using plateaus. *Journal of Graphics Tools 6*, 1 (November 2001), 19–27.

[HBS00] HEIDRICH W., BRABEC S., SEIDEL H.-P.: Soft shadow maps for linear lights. *Proceedings of the 2000 Eurographics Symposium on Rendering* (2000), 269–280.

[HDG99] HART D., DUTRÉ P., GREENBERG D. P.: Direct illumination with lazy visibility evaluation. *ACM Computer Graphics (Proc. of SIGGRAPH '99)* (August 1999), 157–154.

[Hei91] HEIDMANN T.: Real shadows, real time. *Iris Universe* (November 1991), 18:23–31. Silicon Graphics Inc.

[HH97] HECKBERT P. S., HERF M.: Simulating soft shadows with graphics hardware. *Carnegie Mellon University Tech. Rep*, CMU-CS-97-104 (January 1997).

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F. X.: A survey of real-time soft shadows algorithms. *Comput. Graph. Forum 22*, 4 (2003), 753–774.

[PSS98] PARKER S., SHIRLEY P., SMITS B.: Single sample soft shadows. *University of Utah Tech. Rep.*, TR UUCS-98-019 (October 1998).

[RSC87] REEVES W. T., SALESIN D. H., COOK R. L.: Rendering antialiased shadows with depth maps. *ACM Computer Graphics (Proc. of SIGGRAPH '87) 21*, 3 (July 1987), 283–291.

[SCH03] SEN P., CAMMARANO M., HANRAHAN P.: Shadow silhouette maps. *ACM Computer Graphics (Proc. of SIGGRAPH '03) 22*, 3 (July 2003), 521–525.

[SD02] STAMMINGER M., DRETTAKIS G.: Perspective shadow maps. *ACM Computer Graphics (Proc. of SIGGRAPH '02) 21*, 3 (2002), 557–562.

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Computer Graphics (Proc. of SIGGRAPH '02) 31*, 3 (2002), 527–536.

[SKvW*92] SEGAL M., KOROBKIN C., VAN WIDENFELT R., FORAN J., HAEBERLI P.: Fast shadows and lighting effects using texture mapping. *ACM Computer Graphics (Proc. of SIGGRAPH '92)* (1992), 249–252.

[SR00] STARK M. M., RIESENFELD R. F.: Exact illumination in polygonal environments using vertex tracing. *Proceedings of the 2000 Eurographics Symposium on Rendering* (2000), 149–160.

[SS98] SOLER C., SILLION F. X.: Fast calculation of soft shadow texture using convolution. *ACM Computer Graphics (Proc. of SIGGRAPH '98)* (1998), 321–332.

[SUC95] SLATER M., USOH M., CHRYSANTHOU Y.: The influence of dynamic shadows on presence in immersive virtual environments. 8–21.

[Wan00] WANGER L.: The effect of shadow quality on the perception of spatial relationships in computer generated imagery. *In Proceedings of Symposium on Interactive 3D Graphics* (2000), 39–42.

[WH03] WYMAN C., HANSEN C.: Penumbra maps: Approximate soft shadows in real-time. *Proceedings of the 2003 Eurographics Symposium on Rendering* (2003), 202–207.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. *ACM Computer Graphics (Proc. of SIGGRAPH '78) 12*, 3 (1978), 270–274.

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