Accurate Shadows by Depth Complexity Sampling
Vincent Forest, Loïc Barthe, Mathias Paulin

To cite this version:

HAL Id: hal-01516270
https://hal.archives-ouvertes.fr/hal-01516270
Submitted on 11 May 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Accurate Shadows by Depth Complexity Sampling

Vincent Forest, Loïc Barthe and Mathias Paulin†

University of Toulouse, France-IRIT-CNRS

Abstract
The accurate generation of soft shadows is a particularly computationally intensive task. In order to reduce rendering time, most real-time and offline applications decorrelate the generation of shadows from the computation of lighting. In addition to such approximations, they generate shadows using some restrictive assumptions only correct in very specific cases, leading to penumbra over-estimation or light-leaking artifacts. In this paper we present an algorithm that produces soft shadows without exhibiting the previous drawbacks. Using a new efficient evaluation of the number of occluders between two points (i.e. the depth complexity) we either modulate direct lighting or numerically solve the rendering equation for direct illumination. Our approach approximates shadows cast by semi-opaque occluders and naturally handles area lights with spatially varying luminance. Furthermore, depending on the desired performance and quality, the resulting shadows are either very close to, or as accurate as, a ray-traced reference. As a result, the presented method is well suited to many domains, ranging from quality-sensitive to performance-critical applications.

Keywords: Soft Shadows, Depth Complexity Sampling, Penumbra Wedge

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Color, shading, shadowing, and texture

1. Introduction
Shadows are fundamental for the realism of virtual 3D scenes. They give information about the relationship between objects and enhance rendering quality by modelling light-surface visibility. Hard shadows are the simplest to generate. Considering the light as a point, they result from the visibility query between a point \( p \) in the scene and the light. If the light is not reduced to a simple point, more realistic soft shadows are produced by evaluating the part of

the light source visible from \( p \). Soft shadows can be seen as a direct extension of hard shadows since they require the evaluation of the visibility query between \( p \) and an infinite number of samples from the light source. The computation of soft shadows is therefore time-consuming and specific algorithms are required to reduce computational complexity.

For interactive applications, the required real-time frame rate is reached by computing approximated soft shadows. This approximation can be either visually plausible, when it is based on the filtering of hard shadow boundaries [RSC87, Fer05], or physically plausible, when it is computed by evaluating the light area visible from the points \( p \) seen in the scene [AAM03, GBP06]. However, these last algorithms generate artifacts from overlapping penumbras, light-leakage and/or have performance bottlenecks. Most of them are based on the shadow maps framework [Wil78]. As a result, they are subject to the artifacts introduced by the discrete, surjective representation of the scene. In addition, their computational complexity and memory consumption increase when they deal with omni-directional lights rather than spot lights. Others are object-based shadow volume algorithms [Cro77]. They do not exhibit the shadow map limitations and despite fill-rate bottleneck and geometry constraints they produce exact hard shadows and they naturally handle point, spot and infinite lights.

**Contribution:** Starting from these observations, we take advantage of the robustness of shadow volumes to compute soft shadows as accurately as a ray-traced reference with a controllable, quality-dependent frame-rate ranging from real-time to interactive. In addition, our unified framework provides the required information to correctly solve the rendering equation [Kaj86] for direct illumination from area light sources. Thus, it is well suited for many domains, ranging from quality-sensitive applications to performance-critical ones. Our technique is summarized as follows. For each light, we identify the visible surface points in the penumbra region using the penumbra wedge primitive [AMA02, AAM03]. The depth complexity function returns the number of occluders between two points [LAA∗05, LLA06]. We evaluate this function between each surface point in the penumbra, and a set of light samples, with a new technique better suited for our real-time requirements. This allows us to directly identify the visible light samples for the surface points, i.e., those with a depth complexity equal to zero. Then we use this information to either compute the amount of visible light or solve the rendering equation for direct illumination with an accuracy depending on the light sampling.

The remainder of the paper is organized as follows. Section 2 presents a brief overview of previous work on the generation of soft shadows. In section 3, we describe how the depth complexity function is used to compute the amount of light visible from a point \( p \) according to the opaqueness of the occluder and the luminance of the light samples. The real-time depth complexity computation is detailed in section 4 while section 5 presents the light sampling strategy. In section 6 we show how our approach is used to solve the rendering equation [Kaj86] for direct illumination. Section 7 describes the GPU implementation. Finally, we present results in section 8 and we end with a discussion of directions for future work.

2. Related work

Shadow computation is a widely studied problem and a survey on hard shadows has been presented by Woo et al. [WPF90], while real-time soft shadows have been recently covered by Hasenfratz et al. [HLHS03]. In this section, we focus on the most recent contributions.

Shadow maps [Wil78] have been widely extended to provide soft shadows. Some algorithms use a set of shadow maps per light, limiting their use to static scenes [ARHM00, HBS00, SAPP05]. Others use a single light sample depth map and try to preserve high performance by imposing scene limitations [SS98, BS02] or using heuristics [AHT04, ED06] rather than visibility computations. Recently, Atty et al. [AHL∗06] and Guennebaud et al. [GBP06] introduced soft shadow mapping. In order to compute visibility, they consider the depth map as a discrete representation of the scene and they back-project shadow map samples [DP94] onto the light source. The same concept is presented by Aszódi and Szirmay-Kalos [ASK06] and Bavoil et al. [BCS06]. Where the back-projected samples overlap, however, this leads to penumbra over-estimation that can be reduced by a more accurate occluder detection [GBP07] or by a logic binary combination of the back-projected samples [SS07].

Although these algorithms produce convincing results, the inherently discrete, surjective nature of their image-based framework limits their accuracy. This is why, despite the constraints on the occluding geometries, the penumbra wedge algorithm [AMA02, AAM03] is particularly attractive. A penumbra wedge conservatively bounds the penumbra region defined by a silhouette edge seen from the light center. Thus, each point \( p \) inside a wedge has its light visibility potentially influenced by the corresponding edge. In the penumbra wedge algorithm, the amount of light seen by \( p \) is computed by the accumulation of the occluded area of the projected silhouette edges onto the light source. However, the overlapping of the back-projected edges leads to over-estimated shadows that can be reduced by an expensive blending heuristic [FBP06].

Laine et al. [LAA∗05] generalized the penumbra wedge method for offline rendering for planar area light sources. In order to compute physically based soft shadows they use the wedges to define a list of edges that can influence the amount of visible light for a point \( p \). In a second step, they project the edges, seen from \( p \), onto the light source and they use special edge rules to evaluate the depth complexity of
a set of light samples. Finally, to determine whether samples with the lowest depth complexity are occluded, a single shadow ray is cast to one of them. Despite its accuracy, this approach can be less efficient than common ray-traced shadows. Lehtinen et al. [LLA06] improve its efficiency and reduce its memory consumption by storing edges in a BSP tree instead of the conservative hemicube data structure. In our approach, neither data structures storing silhouette edges for each p, nor explicit edge rules for depth complexity computation, are necessary. In addition, our approach is not limited to planar area light sources and it is well suited to interactive rendering of dynamic scenes.

3. From Depth Complexity to Visibility Coefficient

Real-time physically plausible soft shadow algorithms [AAM03, GBP06] decorrelate the generation of shadows from the computation of lighting by defining a visibility buffer (v-buffer) storing for each viewed point p of the scene its corresponding visibility coefficient (v-coef ∈ [0, 1]). We denote as v-coef the percentage of visible light and, in this section, we propose a very simple v-coef formulation based on the depth complexity between p and a set of light samples.

**Overview:** The depth complexity function returns the number of occluders between two points. Considering a light sample s and a surface point p, s is visible from p if its depth complexity from p is equal to zero. Thus, a v-coef for p can be simply computed from a set of N samples uniformly distributed over a light l as:

\[
V(l \leftrightarrow p) = 1 - \frac{1}{N} \sum_{i=0}^{N-1} Sat(D(s_i \leftrightarrow p))
\]

(1)

where \(D(s_i \leftrightarrow p)\) returns the depth complexity ∈ N between the \(i^{th}\) light sample \(s_i\) and p. Sat(x) is a saturating function that clamps x into [0, 1].

**Textured light:** Area light sources such as a fire or a TV screen have a spatially varying luminance. Using depth complexity, such textured light sources can be simply taken into account during the v-coef computation by multiplying the visibility query (i.e. the saturated depth complexity) by the sample luminance \(L\).

\[
V_{rgb}(l \leftrightarrow p) = \frac{1}{N} \sum_{i=0}^{N-1} L_{rgb}^{i} - L_{rgb}^i \cdot Sat(D(s_i \leftrightarrow p))
\]

(2)

The sample luminance can be encoded with an arbitrary color space. However, since the red, green, blue (RGB) representation is well suited for common rendering engines, we define sample luminance and v-coef as RGB values ∈ [0, 1]^3. In addition, sample luminance can vary over time. We take into account these animated textured lights by adding time dependence to the luminance.

**Surface opaqueness:** In order to compute soft shadows cast by semi-opaque occluders, we must evaluate the amount of attenuated light from s to p. This is done by replacing the depth complexity function \(D(s_i \leftrightarrow p)\) in equation 2 by a light attenuation function \(Dr(s_i \leftrightarrow p)\) computed by summing the occluders’ percentage of opaqueness. The opaqueness factor is a constant for an occluding surface and it simulates surfacic light attenuation. Functions D and Dr are equal for opaque occluders and in the following, we do not differentiate, instead denoting D and Dr as a real depth complexity function. Note that our surface opaqueness computation is an approximation producing accurate results only when a single semi-opaque surface occludes p. Indeed, the light contribution is derived from the sum of opaqueness factors while it should rather be modulated by the attenuation factor of all occluding surfaces.

4. Local Depth Complexity Computation

This section details the streamed evaluation of the depth complexity function. In a first step, we compute the silhouette edges from the entire scene w seen from a light l. Then, we initialize the depth complexity between p and a set of light samples from the pool of edges E (section 4.1). Finally, the wedges W built from silhouette edges are used to define points p whose shadow term is affected by E and to locally update their depth complexity (section 4.2).

4.1. Depth Complexity Initialization

The depth complexity between p and a light sample s was originally defined as the number of surfaces that a ray from p to s intersects. A depth complexity of n (n greater than zero) means that n surfaces occlude s as seen from p. However, potential changes in the visibility function can occur only on the silhouette edges. It is therefore sufficient to track the occluding silhouette loops rather than the occluding surfaces. In consequence, we reformulate the depth complexity function as the number of silhouette loops occluding s from p.

This reinterpretation addresses a limitation of previous algorithms. Indeed, as explained by Laine et al. [LAA*05], each silhouette edge generates a local change in the depth complexity function and the set of all silhouette edges represents its derivative. Integrating over the local changes results in integrating the derivative of the depth complexity function, and gives the depth complexity without the constant of integration. In order to define this constant, Laine et al. cast a shadow ray towards the light sample with the lowest depth complexity, thus limiting the performance and the dynamism of the rendered scene. However, the shadow volume algorithm computes for each visible point p the number of occluding silhouette loops. Using the previous reformulation, this defines the constant in the integration of the depth complexity function derivative. As a result we can avoid the
ray casting by initializing the constant of integration with the result of the shadow volume step.

Note that this is quite similar to the initialization step of the penumbra wedge approach [AAM03]. The main difference is that we consider the result of the shadow volume step of the light source. The covered samples are then incremented or decremented according to the wedge type (outer or inner).

4.2. Update of the Depth Complexity

Algorithm 1 update_depth_complexity(Wedges W)
1: for all \( w_e \in W \) do
2: for all \( p \) in \( w_e \) do
3: \( e \leftarrow \) edge associated with \( w_e \);
4: \( e^p \leftarrow e \) projected from \( p \) onto the light;
5: \( e^c \leftarrow e^p \) clipped against the light border;
6: for \( i = 0 \) to \( \text{NBR\_LIGHT\_SAMPLES} \) do
7: if \( p\_\text{sample}[i] \) is covered by \( e^c \) then
8: if \( w_e \) is an outer wedge then
9: \( p\_\text{sample}[i].\text{depth\_complexity} += 1; \)
10: else
11: \( p\_\text{sample}[i].\text{depth\_complexity} -= 1; \)
12: end if
13: end if
14: end for
15: end for
16: end for

The depth complexity integration is performed as in algorithm 1.

The depth complexity between a point \( p \) and a set of light samples is updated by processing the silhouette edges whose projections from \( p \) overlap the light source (line 3). Due to its linear nature, the integration can be performed separately for each projected edge and it is independent for each light sample (line 6). Thus, we update the depth complexity counter independently for each light sample as follows. The shadow volume quadrilateral splits each wedge into two parts: the inner part and the outer part. In both cases, the edge is projected onto the light source (line 4) to define the covered light samples (line 7). Their corresponding depth complexity counters are then incremented (line 9) or decremented (line 11) if the wedge is respectively outer or inner (figure 2). Since projected edges are clipped (line 5), silhouette edges crossing the light border are also naturally handled (standard offline integrations require specific treatments here).

4.3. The Counter Packing

The depth complexity is evaluated with edges encoded as a streamed data set rather than a common static list. Instead of iterating over the list of edges to globally compute the depth complexity between a sample \( s \) and a point \( p \), we progressively update it for each edge that potentially affects the visibility. This avoids the use of a costly pre-computed static space-partitioning data structure referencing silhouette edges [LAA*05, LLA06]. For each visible point \( p \) we maintain a set of depth complexity counters corresponding to the set of light samples. Therefore the number of available counters must be equal to the number of light samples, and the precision available has to be sufficient to store the maximum number of occluders.

Rendering engines provide render buffers with, typically, four values (red, green, blue, and alpha). In order to provide a sufficient number of counters being both efficiently stored and updated, we pack several counters into a single value as illustrated in the following example. Considering a value \( v \) encoded in a 32-bit unsigned integer. The value \( v \) can store a single counter ranging from 0 to \( 2^{32} - 1 \) or, for instance, four values ranging from 0 to 255 in four consecutive bytes. This packed representation is particularly well suited for a vectorized incrementation/decrementation. Indeed, assuming that the resulting values do not overflow the domain of the counters, the addition of a four-byte value with correctly positioned bits performs a vectorized update of the counters as shown in figure 3.

4.4. Advantage and Drawbacks

Counter packing provides an efficient update of the depth complexity counters and increases their quantity. However, only unsigned integers can be used. Even though depth complexity cannot be negative, inner wedges can affect the depth complexity counters before the outer wedges, resulting in
5. Light Sampling Strategy

The quality of the reconstructed visibility function between a point \( p \) and an extended light source depends on both the number of samples and their distribution over the light (figure 5). In this section we detail our sampling strategy and propose a method to dynamically adjust the balance between the precision of the depth complexity and the number of samples.

5.1. Samples Distribution

In order to accurately evaluate the visibility function for \( p \), it is necessary to distribute random samples onto the light source using an adapted probability distribution function [PH04]. However, a fixed sampling pattern for all \( p \) may become visible (figure 5(a)), even with several light samples. In order to alleviate this drawback, one can vary the set of samples for each \( p \) by randomly rotating the initial sampling pattern (figure 5(b)). In addition to this decorrelation, we perform a better distribution of the samples using a stratified sampling strategy. (figures 5(c) and 5(e)).

5.2. Interleaved Sampling

Correlated samples generate visible sampling patterns and their decorrelation generates noise. Interleaved sampling [KH01] takes advantage of both distributions. The main idea is that for neighbour points, a measured signal may produce the same result. Thus, the signal for nearby points is sampled with an interleaved sampling distribution and finally merged into a single sampling pattern. Segovia et al. [SIMP06] proposed a real-time framework for this technique. To conserve the coherence between neighbour pixels, they split the rendered image into sub-buffers containing the pixels that use the same distribution of samples. After sampling the desired signal, they gather the sub-buffers and filter the result according to the normal and depth discontinuities.

Interleaved sampling can be efficiently used in our framework (figure 5(d) and 5(f)). However, in order to avoid multiple rendering of shadow volumes and wedges, we use neither the image splitting nor the gather step. In spite of the loss of near points coherence, no drop of performance occurs since no coherent memory access is required (section 7).

5.3. Adaptive Sampling

The maximum depth complexity of the rendered scene depends on the scene complexity and the point of view. While rendering a plain leads to a globally low depth complexity, a forest rendering generates many occlusions. Given a fixed amount of available memory for a visible point \( p \), we propose a simple way to dynamically adjust the precision and the number of packed depth complexity counters.

The maximum depth complexity between the light samples and \( p \) is equal to the maximum number of occluding silhouette loops. Thus, in a first step, we track the maximum integration constant \( m_t \) resulting from the depth complexity initialization of the current visible points. Then we use this value to define the counter packing encoding. As an illustration, consider the previous example where four-byte are available for each visible point. With \( m_t \) in \([256, 65535]\), two counters of two-byte \((\in [0, 65535])\) can be packed without overflow, while \( m_t \) in \([128, 255]\) allows the use of four counters of one byte \((\in [0, 255])\).
6. Sampling of the Direct Illumination

The depth complexity function is used to compute a visibility coefficient of a point \( p \). The v-coef is then used to modulate the direct illumination of \( p \) computed from the center of the light. Hence, considering a set of \( j \) lights, the direct illumination according to a given point of view \( x' \) is computed as:

\[
L(p \rightarrow x') = L_e(p \rightarrow x') + \sum_j f_j(x \rightarrow p \rightarrow x') L(x \rightarrow p)
\]

\[
V(l_i \rightarrow p) \cos \theta_i \cos \theta_s \frac{dA}{||x-p||^2}
\]

(3)

where \( L_e \) is the emitted radiance and \( f_j \) is the BSDF. Even though the visibility is evaluated for an area light source, the direct lighting is still computed using point lights. In order to compute an accurate direct illumination for extended light sources, we have to solve the rendering equation [Kaj86] for the surfaces \( S \) of the area light sources:

\[
L(p \rightarrow x) = L_e(p \rightarrow x) + \int_S f(x \rightarrow p \rightarrow x') L(x \rightarrow p)
\]

\[
v(x \rightarrow p) \cos \theta_i \cos \theta_s \frac{dA}{||x-p||^2}
\]

(4)

Note that the binary visibility \( v(x \rightarrow p) \) can be very simply evaluated from the depth complexity function as:

\[
v(x \rightarrow p) = H(D(x \rightarrow p))
\]

(5)

where \( H(x) \) is the Heaviside function that returns one if \( x \) is strictly positive and zero otherwise. Thus, we can numerically solve the equation 4 for the direct illumination, using a set of \( N \) samples uniformly distributed over each extended light source, and their corresponding depth complexity:

\[
L(p \rightarrow x) = L_e(p \rightarrow x) + \sum_j \frac{A(l_j)}{N} \sum_{k=0}^{N-1} f_k(x_{k,j} \rightarrow p \rightarrow x') L(x_{k,j} \rightarrow p)
\]

\[
H(D(x_{k,j} \rightarrow p)) \cos \theta_i \cos \theta_s \frac{dA}{||x-p||^2}
\]

(6)

where \( x_{k,j} \) is the \( k^{th} \) sample of the light \( j \) and \( A(l_j) \) is the area of the \( j^{th} \) light source. This equation 6 naturally handles textured lights since the incoming radiance \( L(x_{k,j} \rightarrow p) \) of the light samples is explicitly defined. Note that we replace the Heaviside function by the Saturate one when simulating semi-opaque occluders.

Figure 5: Comparison of the soft shadow quality according to the number of samples and the sampling strategy. Shadows are generated with either 64 (a, b, c, d) or 16 (e, f) samples using a correlated (a), decorrelated (b), decorrelated and stratified (c, e) or interleaved sampling pattern (d, f).

Figure 6: Overview of our algorithm for GPU implementation of the proposed depth complexity evaluation. All computations are performed on the GPU without transferring data to main memory.

7. Implementation

In this section, we present a GPU-intensive implementation of our algorithm. The proposed rendering algorithm (algorithm 2) is developed for the latest generation of graphics hardware. We therefore use fragment programs (FP), vertex programs (VP) and geometry programs (GP). Figure 6 summarizes the GPU implementation and the render context organization, which we elaborate in the following explanations.

7.1. Sampling Distributions

Depending on the sampling strategy and light type we pre-compute and store into textures, the corresponding 2D sample distribution (we assume that omni-directional light sources can be parametrized in 2D). In order to limit the requirements on memory and the number of texture accesses,
we pack several 2D sample positions per texel. Since the sample positions are computed in the normalized texture space, a precision of one byte per coordinate does not introduce a significant error. This precision allows us to encode two sample positions in a texture channel of four-byte. Hence, using four channels (i.e. a RGBA texture), we obtain eight sample positions in one texture fetch. Note that textured lights require the luminance of each sample. Thus, in addition to their packed position, we pre-compute for these lights an $N \times M$ 24-bit RGB texture that stores the $N$ sample luminances for $M$ sets of decorrelated sample patterns.

7.2. Soft Shadow Volumes Framework

Silhouettes detection: We perform silhouette detection on the GPU using a specific GP. The detected silhouettes are stored on the GPU in a Transform Feedback Buffer (TFB). Since writing to the TFB is asynchronous, CPU computations such as scissor rectangle definition or depth bounds evaluation [Len05] are performed in parallel. Note that according to the desired quality and performance trade-off, the silhouette determination can be computed with either accurate or standard detection.

Shadow volumes: Silhouette edges are then used in the shadow volume pass to define the initial value of the depth complexity function. We perform silhouette edge extrusion with a GP. In order to avoid a costly access to the stencil buffer, the z-fail stencil test [Car00] is performed in a single pass in a simple FP. The resulting value is then cumulatively blended in the alpha channel of the color buffer:

```
# face.x = fragment is front-facing ? 1 : -1
ATTRIB face = fragment.facing;
ATTRIB fPos = fragment.position;
TIND r6;
# texture[0] == zBuffer
TEX r0.x, fPos.x, texture[0], RECT;
SGE r6.x, fPos.x, r0.x;
MUL result.color.w, r0.x, -face.x;
```

For a robust z-fail stencil update, the shadow volumes have to be capped. Thus, we use a specific GP to compute the shadow volumes capping in an interleaved geometric pass.

Wedges: In order to reduce memory consumption, wedges are robustly extruded to infinity during the depth complexity evaluation, rather than being pre-computed and stored. We perform this construction with a GP. However, the performance of a GP is influenced by the number of generated data. Thus, the number of required vertexes is minimized using a single triangle strip per wedge and only twelve vertexes per half-wedge. Figure 7 illustrates this generation of wedges. Also, common fill-rate optimisations and fragment rejection are used during the wedges rendering. We refer to [Len05] and [FBP06] for additional details.

7.3. The Depth Complexity Computation Step

Counter packing representation: Since reading and writing in the same buffer is prohibited, we use the blending operations of the GPU to perform the counter updates. Unfortunately, even though the latest GPUs support integers, their blending is not yet supported. This avoids a naïve GPU implementation of the depth complexity update. On the one hand, the counter packing corresponds to a base decomposition where each base factor encodes the value of a counter with a precision up to the base—1 e.g. two counters $c_0$ and $c_1 \in [0 \cdot 255]$ packed in a two-byte value $k$ is decomposed in
base 256 as \( k = c_0 \times 256^0 + c_1 \times 256^1 \). On the other hand the simple precision floating point values are supported for both buffer format and blending operations. With this representation, one can count up to \( 2^{24} - 1 \) without missing an integer value and this value can be expressed in the following base decomposition:

\[
2^{24} - 1 = 255 \times (256^0 + 256^1 + 256^2)
\]

\[
= 63 \times (64^0 + 64^1 + 64^2 + 64^3)
\]

\[
= 15 \times (16^0 + 16^1 + 16^2 + 16^3)
\]

\[
= 7 \times (8^0 + 8^1 + 8^2 + 8^3 + 8^4 + 8^5 + 8^6)
\]

Thus, we use the floating point representation and the base decompositions exposed in equations 7, 8, 9 and 10, to pack the depth complexity counters.

Depth complexity initialization: We present the depth complexity initialization with a fixed Base 64 (B64) counter packing representation (equation 8). A B64 counter packing provides sixteen depth complexity counters per 128-bits RGBA buffer. We then use Multi Render Target (MRT) to increase the number of counters. Even tough recent GPUs support up to eight MRT, we limit the memory bandwidth and its consumption by using at most four MRTs (i.e. 64 counters). The depth complexity initialization of the 64 counters is then very simply performed via an FP as:

\[
\text{ATTRIB fPos = fragment.position;}
\]

\[
\text{TEMP r0;}
\]

\[
\text{# texture[0].w == stencil value}
\]

\[
\text{# 266305 = 64 \times (64^1 + 64^2 + 64^3)}
\]

\[
\text{TEX r0.w, fPos, texture[0], RECT;}
\]

\[
\text{MUL r0.w, r0.w, 266305;}
\]

\[
\text{MOV result.color[0], r0.w;}
\]

\[
\text{MOV result.color[1], r0.w;}
\]

\[
\text{MOV result.color[2], r0.w;}
\]

\[
\text{MOV result.color[3], r0.w;}
\]

Depth complexity update: One of the main challenge we face while updating the depth complexity is efficiently determining samples covered by the projected clipped edge \( e_p \). Despite having access to eight sample positions in one texture fetch, a naïve search leads to a complexity in \( O(N) \) where \( N \) is the number of samples. We therefore propose an efficient discrete approach to finding the covered samples. A pre-computed 4D texture \([AAM03]\) uses large amount of memory and produces discretization artifacts (figure 9(a)), while the Hough transform, as in \([ED07]\), requires that we either approximate, or pre-compute, the unsupported arc-cosine function. This leads to either additional texture fetches or heavy computation. Hence, we propose a new discrete representation (figures 8 and 9(b)). First, covered samples lying in a sector defined by the origin and vector \( v_x \) \((x \in \{0, 1\})\) are encoded in a bit field and stored in a cube map. Then, a 2D texture indexed by the orthogonal projection of the light center onto the line \( e_p \) stores the bit field of samples covered by this line. The final bit mask \( M \) is then simply defined as \( M = v_1 \cdot \text{bits} \& \sim \text{v}_0 \cdot \text{bits} \& e_p \cdot \text{bits} \). Finally, we iterate through the bits of \( M \) to update the corresponding counters. Despite the use of integer operations, which are less efficient than floating point, this method requires only three texture fetches and a logical combination to define all the covered samples. Note that this discretization is robust even though the light center passes through the line \( e_p \). Indeed, in this case, the effective covered samples are just defined by the cube map and so the 2D texture has to store a field of bits set at one.
7.4. Rendering the Direct Illumination

Finally, the depth complexities associated with a fragment are used to solve either the equation 6 or to compute the corresponding v-coef (equation 2). The first approach merges the direct illumination computation with the visibility queries while the second performs an additional step to compute the v-coef used to modulate the direct lighting. The resulting lighting contribution is then cumulatively blended. Note that in both cases, the interleaved sampling strategy requires an additional filtering step combining the interleaved sampling patterns [SIMP06].

8. Results

8.1. Memory Cost

<p>| Table 1: Detailed memory costs of our algorithm implementation with a 1024² image resolution. An edge LUT (c) stores our discreet covered samples representation while a per light LUT (d) stores the luminance texture lookup defined per textured light. The v-buffer (g) is not used when the equation 6 is solved. Finally, discontinuity (h), temp filtered (i) and deferred (j) buffers are only necessary with an interleaved sampling strategy. |</p>
<table>
<thead>
<tr>
<th>幅</th>
<th>Format</th>
<th>Memory cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Silhouettes TFB</td>
<td>1,310×720×32F</td>
<td>5MB</td>
</tr>
<tr>
<td>(b) Samples position</td>
<td>64 samples packed in RGBA 32F</td>
<td>128B</td>
</tr>
<tr>
<td>(c) Edge LUT</td>
<td>((512^2 + 6 + 64^2) \times RGBA\ 32F)</td>
<td>4.375MB</td>
</tr>
<tr>
<td>(d) Per light LUT</td>
<td>64 × 64 RGB 3UB</td>
<td>12KB</td>
</tr>
<tr>
<td>(e) Color + Stencil buffer</td>
<td>RGBA 16F</td>
<td>8MB</td>
</tr>
<tr>
<td>(f) Z-buffer</td>
<td>DEPTH_COMPONENT24</td>
<td>3MB</td>
</tr>
<tr>
<td>(g) V-buffer</td>
<td>RGBA 3UB</td>
<td>3MB</td>
</tr>
<tr>
<td>(h) Discontinuity buffer</td>
<td>Alpha 4UB</td>
<td>1MB</td>
</tr>
<tr>
<td>(i) Temp filtered buffer</td>
<td>RGBA 3UB</td>
<td>4MB</td>
</tr>
<tr>
<td>(j) Deferred buffer</td>
<td>RGBA 32F</td>
<td>12MB</td>
</tr>
<tr>
<td>(k) DC-buffer(s)</td>
<td>RGBA 32F</td>
<td>16MB</td>
</tr>
</tbody>
</table>

Table 1 shows the memory used by data structures in our implementation. We do not apply any texture compression in order to present significant memory cost without the influence of a specific/subjective compression method. In addition, we reserve 5MB of memory for the silhouettes Transform Feedback Buffer, allowing us to store up to 655,360 silhouette edges.

The memory requirement of our algorithm is independent of the scene complexity and depends only on the algorithm parametrization and the light types of the scene. Using only one depth complexity buffer (DC-buffer) for direct illumination without textured light and covered samples discretization requires only \((a) + (b) + (e) + (f) + (h) \approx 32MB\) of memory. On the other hand, computing an image modulated by the v-buffer, with ten textured lights, four DC-buffers, an interleaved sampling strategy and the discrete covered samples representation requires \((a) + (b) + (e) + 10 \times (d) + (e) + (f) + (g) + (h) + (i) + (j) + 4 \times (k) \approx 105.575MB\) of memory. However, this memory requirement is not a limitation on current high-end GPUs.

8.2. Performances

Figure 10: Time in milliseconds of the rendering for a 1024² image. Japan1: 4 lights, 501,650 polygons; Japan2: Japan1 scene with a reduced number of polygons (246,176 triangles); Kitchen: 2 lights, 146,131 polygons; Doom3: 2 lights, 17,693 polygons.

Figure 11: Time in milliseconds of the rendering steps according to the algorithm parametrization. The two test scenes are lit by a single light that generates large penumbra region. Image resolution: 1024²; Polycount: Japan=501,650, Doom3=17,693

We present the performance on a complete 1024 × 1024 image rendering. Indirect lighting is approximated with an irradiance map [RH01] and the Blinn BRDF [Bli77] is used for the materials’ appearance. Our renderer is based on the OpenGL API and all the shaders are written in the pseudo assembly language of the NV_gpu_program4 extension. The results are measured on a 64-bits Linux workstation with a Core 2 Duo E6700, 4GB of DDR2 800Mhz and a Geforce 8800GTX. Our benchmarks measure the global rendering time (figure 10) and the cost of the different steps (figure 11) on three test scenes (figure 12) with varying algorithm parametrization. To stress our approach, the light sources are not attenuated and so they do not take advantage of per-light scissor and depth bound optimisations. Finally, we use a counter packing representation where each DC-buffer stores 16 depth complexity counters with a precision up to 63 (equation 7).

Figure 10 illustrates that the Depth Complexity Sampling with 16 samples (DCS16) is approximately 5 times faster than using 64 samples (DCS64). This performance improvement is explained by the limitation on memory bandwidth and texture fetches, in addition of the reduction in working
Figure 12: The three test scenes used in our benchmark. (a) and (b) are high-polygon scenes while (c) is a game scene including animated skinned characters.

Figure 13: Comparison of the shadows generated by the penumbra wedge algorithm (a), our Depth Complexity Sampling (b) and a Ray-Traced reference (c). In addition of the overlapping penumbrae artifact, the penumbra wedge algorithm produces an incorrect lighting due to the single light sample direct lighting computation.

9. Discussion and Future Works

The presented depth complexity sampling algorithm allows the generation of fast and accurate shadows. However, since it uses the soft shadow volume framework, it handles only polygonal models and is submitted to fill-rate bottleneck. Furthermore, the splitting of wedges in inner and outer parts generates aliasing due to precision errors. Nevertheless, our approach is orthogonal to the previous object-based methods and thus it may take advantage of all their current and future improvements.

Due to its sampling nature, and despite the use of the adaptive sampling strategy, the proposed algorithm is still exposed to the sub-sampling artifacts. However, since we use a Monte Carlo sample distribution, averaging the result of several runs would give a solution that would be statistically very close to the exact solution. One can therefore imagine a progressive rendering or a multi-GPU system where each picture is computed according to different sampling distribution and then averaged in the final image.

The adaptive sampling algorithm defines the counter packing representation for a given point of view. Note that the counter packing encoding can be locally defined per pixel rather than globally set for the current rendered image. A simple idea is to track the number of wedges bounding each pixel.

The lack of integer support in the blending stage limits both the precision of the counters and the efficiency of memory usage. In addition, despite the use of an adaptive sampling strategy, the counter packing robustness is still compromised by an eventual overflow error. We propose an API extension that address these limitations by exposing a buffer as a pool of m available bits for each pixel. The application would explicitly define how to partition m into N channels, while a pool of bits would be reserved to save the state of each channel (overflow error, NaN indicator etc.).

10. Conclusion

We have presented an accurate and efficient soft shadow volume algorithm based on a new fast evaluation of the depth complexity function. The proposed algorithm addresses the limitations of both the penumbra wedge approach [AAM03] (figure 13) and the offline soft shadow volume methods [LAA*05, LLA06]. We have demonstrated that, despite its
object-based nature, this algorithm can efficiently deal with complex animated models lit by a complex direct illumination. Contrary to the image-based approaches, our method does not use a discrete, surjective scene representation. In addition it naturally handles the omni-directional surfacic lights without any additional drop in performance or specific treatments.

Acknowledgements

We thank Tom Cashman for his help in the completion of the paper.

References


