EUROGRAPHICS 2013 / I. Navazo, P. Poulin
(Guest Editors)

Volume 32 (2013), Number 2

Adaptive Quantization Visibility Caching

Stefan Popov[1] Iliyan Georgiev[1,2] Philipp Slusallek[1,2,3] Carsten Dachsbacher[4]

[1]Saarland University [2]Intel VCI, Saarbrücken [3]DFKI, Saarbrücken [4]Karlsruhe Institute of Technology

Abstract

Ray tracing has become a viable alternative to rasterization for interactive applications and also forms the basis of most global illumination methods. However, even today's fastest ray-tracers offer only a tight budget of rays per pixel per frame. Rendering performance can be improved by increasing this budget, or by developing methods that use it more efficiently. In this paper we propose a global visibility caching algorithm that reduces the number of shadow rays required for shading to a fraction of less than 2% in some cases. We quantize the visibility function's domain while ensuring a minimal degradation of the final image quality. To control the introduced error, we adapt the quantization locally, accounting for variations in geometry, sampling densities on both endpoints of the visibility queries, and the light signal itself. Compared to previous approaches for approximating visibility, e.g. shadow mapping, our method has several advantages: (1) it allows caching of arbitrary visibility queries between surface points and is thus applicable to all ray tracing based methods; (2) the approximation error is uniform over the entire image and can be bounded by a user-specified parameter; (3) the cache is created on-the-fly and does not waste any resources on queries that will never be used. We demonstrate the benefits of our method on Whitted-style ray tracing combined with instant radiosity, as well as an integration with bidirectional path tracing.

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

1. Introduction

Ray tracing has been an intensive area of research in recent years, and nowadays has a wide range of applications, from interactive to offline global illumination rendering. Consequently, researchers have addressed many different challenges, such as constructing and traversing acceleration structures for ray casting (see [ND12] for a recent overview), as well as improving Monte Carlo methods for sampling light transport in virtual scenes. However, for both interactive and offline rendering the main bottleneck when using ray tracing is typically the number of rays that can be cast within the time given to render a frame. This limitation can be adressed in two ways: further improving acceleration structures and traversal algorithms; or making use of the ray budget in a smarter way, e.g. in spirit of bidirectional path tracing [LW93] which reuses sub-paths for light transport computation, or lightcuts [WFA*05] which uses a single visibility query for a cluster of point lights.

The method we propose in this paper belongs to the latter class, however, it is orthogonal to specific light transport algorithms: we present a novel visibility caching algorithm for ray tracing that is based on quantization (or clustering) of the path space. This quantization does not require any preprocessing, as it is computed on-the-fly, and adapts to geometry, visibility sampling density, as well as to the lighting on the surfaces, e.g. to capture shadow discontinuities. The underlying idea is to replace the visibility between two arbitrary points with the cached visibility between their quantized locations (clusters). Any two points that quantize to the same cluster reuse its visibility from the cache.

To evaluate our algorithm in practice, we have performed tests on a variety of scenes under different lighting conditions. The resulting renderings match the ray traced ones almost perfectly, but require only a small fraction of the visibility queries (below 2% in some cases). This results in a speed up of up to 8× for the total rendering time. In summary, this paper makes the following contributions:

- We introduce a generic point-to-point binary visibility caching algorithm which can be easily incorporated into any ray tracing implementation.
- We describe an adaptive quantization scheme that avoids self-shadowing and provides intuitive user-controlled trade-off between performance and quality.
- We store all visibility information in a single global hashtable with efficient storage and retrieval.

2. Related Work

Visibility computation is a central problem in many computer graphics algorithms where it is required to determine (in)visible surfaces for culling, to compute shading and shadowing, and of course in global illumination (GI) methods. Cohen-Or et al. [COCSD03] and Bittner and Wonka [BW03] provide comprehensive overviews over research in visibility. In general, ray tracing is the most flexible way to determine the visibility between surface points, however, many rendering techniques use rasterization to resolve visibility. Shadow mapping [Wil78] is probably the most popular *from-point* visibility method, which has been successively improved by researches over years; Scherzer et al. [SWP11] provide a comprehensive survey. The main drawback of this method is that a single shadow map only encodes visibility between one point and many others, while GI algorithms often require queries between arbitrary pairs of points. For GI rendering, shadow mapping is typically used together with instant radiosity [Kel97].

As we aim for a general method to speedup visibility in global illumination computation, our focus is on ray tracing. Cache-based visibility acceleration for ray tracing has been done in various ways: Haines and Greenberg [HG86] propagate the occluder found for a shadow ray to nearby rays, which allows for faster identification of potential occluders, however, shadow rays are always cast. Other caching-based GI techniques store illumination instead of visibility, such as the shader cache [TPWG02], the render cache [WDP99], and irradiance caching [WRC88]. Similarly to our approach, Dietrich and Slusallek [DS07] employ an adaptive spatial cache, but also store illumination instead of visibility. All aforementioned caching methods are applied to shading or incident illumination, and are inappropriate for handling high-frequency illumination, such as hard shadows.

Visibility computation has also been accelerated by quantization before. For example, lightcuts [WFA*05] builds a tree hierarchy of virtual point light sources (VPLs), and the illumination from entire light clusters is gathered with a single shadow ray. Hašan et al. [HPB07] solve the many-light problem by clustering VPLs based on their image contribution, employing the GPU for visibility computation. Orthogonally to our approach, these methods aim to reduce the number of visibility queries, not to accelerate them. For a more comprehensive overview we refer to the recent survey by Ritschel et al. [RDGK12].

3. Visibility Caching and Quantization Theory

In this section we introduce the underlying theory of our method, and describe the principles of using visibility quantization and caching for rendering.

We aim at accelerating visibility queries between two surface points, which is a frequent computation in many rendering algorithms. This binary visibility function $V(X,Y)$, strictly speaking between two differential areas, appears in the surface area form of the rendering equation. It has a value of 1 if X and Y are mutually visible, and 0 otherwise. In some cases, e.g. the integration of infinitely distant lights, a modified version of $V(.)$ is used that operates on differential solid angle. In this case, $V(X,\omega) = 1$ if a ray with origin X and direction ω does not intersect any surface. Typically $V(.)$ is computed using ray casting, although for certain applications it is approximated, e.g. using shadow mapping (see Sect. 2). In the following, we develop a theoretical framework that will allow us to approximate $V(.)$ and to control the error introduced by the approximation.

3.1. Overview

In order to control the error caused by any visibility approximation, we have to assess how a query $V(X,Y)$ influences the final image, in case it returns the wrong result. As mentioned before, we focus on rendering algorithms based on ray tracing, which build paths from the camera and/or the light sources and then use the visibility function to connect their vertices. This includes widely used algorithms such as Whitted-style ray-tracing as well as all algorithms derived from instant radiosity and (bidirectional) path tracing.

One important observation is that a query $V(X,Y)$ between two points might be used in two or more different light transport paths, and a cached result thus can cause a different error in the image. This has to be considered in an adaptive caching scheme, and we express this dependence on the actual path by writing a *path visibility function* $\bar{V}(\bar{p}_e, \bar{p}_l)$, where $\bar{p}_e$ denotes an eye sub-path whose last vertex is connected to the last vertex of a light sub-path $\bar{p}_l$.

We approximate visibility by quantizing the domain of $\bar{V}(.)$ which is the space of all paths that connect the camera with the light sources. For this we introduce a mapping $K(\bar{p}_e, \bar{p}_l) \to \mathbb{N}$ and approximate the path visibility function:

$$\bar{V}(\bar{p}_e, \bar{p}_l) \approx V^c(K(\bar{p}_e, \bar{p}_l)). \qquad (1)$$

Each $K(.)$ value $j \in \mathbb{N}$ defines a *quantization cluster* (or simply cluster). For each cluster, we choose two representative points (described in Sect. 3.4), and the *cluster visibility function* $V^c(j) \to \{0,1\}$ returns the binary visibility of the segment connecting the two endpoints.

Based on Eq. 1, we construct an algorithm for efficiently computing the approximation $\bar{V}(\bar{p}_e, \bar{p}_l)$, described in Alg. 1. To this end, we can think of V^c as an array of binary values $\{0,1\}$, and we compute its elements on demand using ray casting. Once computed, we store and reuse each value for subsequent queries to the same cluster, i.e. we trace exactly one ray per quantization cluster, and only if at least one path from the cluster is used in the rendering process; we will refer to this from now on as *visibility caching*.

The most crucial component obviously is the quantization $K(.)$ which we discuss in the remainder of this section.

Algorithm 1 Compute approximate visibility $\tilde{V}(\bar{p}_e, \bar{p}_l)$

```
1: $V^c(j) \leftarrow$ an array initialized to nil on every frame
2: $j \leftarrow K(\bar{p}_e, \bar{p}_l)$
3: if $V^c(j) = $ nil then
4:     $\bar{p}_e(j), \bar{p}_l(j) \leftarrow$ RepresentativePath$(j)$
5:     $X \leftarrow$ LastVertex$(\bar{p}_e(j))$
6:     $Y \leftarrow$ LastVertex$(\bar{p}_l(j))$
7:     $V^c(j) \leftarrow V(X,Y)$          ▷ Use ray-tracing for $V(.)$
8: end if
9: return $V^c(j)$
```

First, we define $K(.)$ through surface patches around the last vertices of eye and light sub-paths (Sect. 3.2). Next, we refine these patches by computing an appropriate shape based on the position, normal, and sampling probability of the last vertices of the sub-paths (Sect. 3.3). In Sect. 3.4 we show how to choose their size w.r.t. the screen space error, and how to pick cluster representatives where the visibility is actually evaluated. In Sect. 3.5 we show how to extend $K(.)$ to also support visibility queries for point and infinite light sources. Lastly, we improve the approximation algorithm to adapt to local variations in visibility (Sect. 3.6).

3.2. Defining the Quantization

Given a path $\bar{p}_e\bar{p}_l$, we define $K(\bar{p}_e, \bar{p}_l)$ by taking two surface patches $A(\bar{p}_e)$ and $A(\bar{p}_l)$, such that the last vertex X of $\bar{p}_e$ lies on $A(\bar{p}_e)$ and the last vertex Y of $\bar{p}_l$ lies on $A(\bar{p}_l)$. As a consequence, similar (nearby) paths, whose eye sub-paths end within $A(\bar{p}_e)$ and whose light sub-paths end within $A(\bar{p}_l)$ are quantized to the same visibility cluster (see Fig. 1).

To define $A(\bar{p}_e)$ and $A(\bar{p}_l)$, we first analyze how their surface areas $|A(\bar{p}_e)|$ and $|A(\bar{p}_l)|$ affect the final image. In the rendering algorithm (e.g. bidirectional path tracing), we construct the sub-paths $\bar{p}_e$ and $\bar{p}_l$ using local path sampling [Vea98, Sect. 8.2.2]. Thus, we can interpret the sampling probabilities $P(\bar{p}_e)$ and $P(\bar{p}_l)$ of the paths as an estimate of how many similar paths end within the unit area patch around X and Y, respectively. Assuming that the rendering algorithm samples N_P paths in total, we estimate the number of paths $N_P(\bar{p}_e, \bar{p}_l)$ clustered together with $\bar{p}_e\bar{p}_l$ as:

$$N_P(\bar{p}_e, \bar{p}_l) \approx N_P\; P(\bar{p}_e)\; |A(\bar{p}_e)|\; P(\bar{p}_l)\; |A(\bar{p}_l)|. \tag{2}$$

This estimation assumes $P(\bar{p}_e)$ and $P(\bar{p}_l)$ are constant within $A(\bar{p}_e)$ and $A(\bar{p}_l)$. Thus, we use $P(\bar{p}_e)|A(\bar{p}_e)|P(\bar{p}_l)|A(\bar{p}_l)|$ as the probability of a path to go through the two patches. In the case of Whitted-style ray tracing, we could also compute the first order derivatives of $P(\bar{p}_e)$ through ray differentials [Ige99], which yields a better estimate for $N_P(\bar{p}_e, \bar{p}_l)$.

The previous equation provides the means to control the quantization error. Choosing all $A(\bar{p}_e)$ and $A(\bar{p}_l)$ such that

$$|A(\bar{p}_e)||A(\bar{p}_l)| = \frac{(C_E)^2}{P(\bar{p}_e)P(\bar{p}_l)N_P}, \tag{3}$$

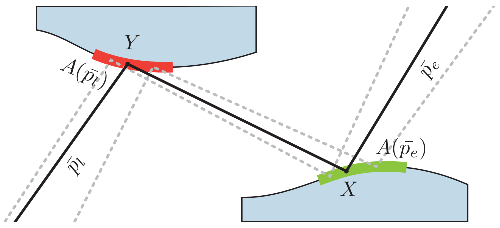

Figure 1: *Similar paths where the last vertex of the light sub-path lies within a region $A(\bar{p}_l)$ and the last vertex of the eye sub-path lies within $A(\bar{p}_e)$ are grouped together and share the same visibility query.*

we can expect that each cluster contains $(C_E)^2$ paths on average. That is, assuming that paths in a cluster are coherent (similar), and thus have similar *unoccluded* contribution to the final image, we can expect that the error introduced by the quantization will change the final image's energy by at most $(C_E)^2$ times. This, however, is the worst case scenario, when the visibility for the representative path is different from the visibility of *all other paths* in each cluster.

The constant C_E is the most important parameter of our algorithm. It is specified by the user and controls the trade-off between quality and performance. Based on this constant, we derive two alternatives for computing $K(.)$ in Sect. 3.4.

3.3. Cluster Shapes

The next step is the construction of $K(.)$, which maps a pair of surface points to a unique cluster. Since we want to avoid the necessity of a surface parametrization, we compute this mapping on-the-fly. For convenience, we will assume that $K(.)$ does not return a single cluster ID, but a tuple of integers representing the quantized surface information.

To this end, we embed the scene's surface into a set of *virtual* multi-resolution, overlapping, and differently oriented voxel grids (see Fig. 2). These grids are never allocated and are only used to describe the quantization. For a vertex X (on an eye or light path) with normal $N(X)$, we first compute a *quantized direction* d_z (where C_N is a user-specified parameter, see below):

$$\omega^q = \left\lfloor \frac{N(X)+1}{2} C_N \right\rfloor, \quad d_z = \frac{2\omega^q}{C_N} - 1. \tag{4}$$

This quantized direction is used to define the orientation of a 3D grid which contains the entire scene, and whose Z axis is parallel to d_z. The orientation of its X and Y axes is not important, as long as they are chosen consistently. The grid resolution is $R(\bar{p}) \times R(\bar{p}) \times C_Z R(\bar{p})$, where C_Z is again a user-parameter and $R(\bar{p})$ is computed from the error threshold (see Sect. 3.4). In total, $K(\bar{p}_e, \bar{p}_l)$ returns a tuple of 14 integers: 3 for the orientation index of X, 3 for the coordinates of the grid voxel containing X, and 1 for the grid resolution $R(\bar{p}_e)$; the 7 integers for Y chosen analogously as for X.

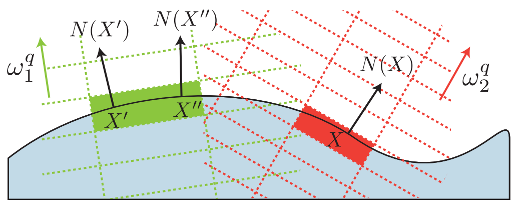

Figure 2: *Visibility domain quantization: X' and X'' reside on surfaces with similar orientation, and are quantized to the same grid and grid cell. The red grid has a finer resolution and represents surfaces with a different orientation.*

Discussion. The parameters C_N and C_Z provide a trade-off between how well the quantization adapts to the surface and the expected acceleration. In all our results we used $C_N = 8$ and $C_Z = 8$, which we found best by experimenting with values in the range $[2, 32]$. We also tried making $K(.)$ cheaper to compute by using the same orientation for all grids and more uniform directional quantization [ED08]. In the first case, we saw no performance improvement but quality degradation due to light leaks. In the latter, we found no difference in the image quality nor rendering performance.

3.4. Grid Resolution and Cluster Representatives

Once we have defined $K(.)$, the next step is to choose the grid resolution $R(\bar{p})$ and the cluster representatives. For $R(\bar{p})$, we first observe that the area of a surface patch around X is

$$|A(\bar{p}_e)| \approx \left(\frac{2\mathcal{B}_R}{R(\bar{p}_e)}\right)^2, \tag{5}$$

where $\mathcal{B}_R$ is the radius of the bounding sphere around the scene; $|A(\bar{p}_l)|$ is defined analogously using $R(\bar{p}_l)$. Plugging Eq. 5 into Eq. 3, we obtain one possible way to determine the grid resolutions:

$$R(\bar{p}_e) = R(\bar{p}_l) = 2\mathcal{B}_R \left(\frac{P(\bar{p}_e)P(\bar{p}_l)N_P}{(C_E)^2}\right)^{-4}. \tag{6}$$

In principle, this is the optimal way to determine the grid resolution, as it produces the smallest patches around both end points and thus increases the chances that our uniform visibility assumption holds (Sect. 3.2). We use Eq. 6 when rendering with bidirectional path tracing.

For Whitted-style ray-tracing, we derive an alternative to determine the resolutions (still based on Eqs. 3 and 5):

$$R(\bar{p}_e) = 2\mathcal{B}_R\sqrt{\frac{P(\bar{p}_e)N_S}{(C_E)^2}},\ R(\bar{p}_l) = 2\mathcal{B}_R\sqrt{\frac{P(\bar{p}_l)N_P}{N_S}}, \tag{7}$$

where N_S is the number of eye paths (i.e. N_P/N_S is the average number of illumination samples per eye path). The advantage of this approach is that C_E can be interpreted in screen space: the error introduced by any cluster $K(.)$ is expected to be within a screen region of size $C_E \times C_E$.

Note that $R(.)$ contains the probabilities of constructing the paths $\bar{p}_e$ and $\bar{p}_l$. As $R(.)$ affects the quantization function $K(.)$, paths sharing a single segment between two points X and Y might still be quantized differently. This is important, as their contributions to the image, and thus the potential error due to quantization and caching, can be different.

To store the values of $R(.)$ (normalized w.r.t. to the scene scale) in the integer tuples returned by $K(.)$, we quantize them using an exponential curve, to converse precision:

$$R \leftarrow \left\lfloor C_R^{\left\lfloor \log_{C_R} R + 0.5 \right\rfloor} \right\rfloor, \tag{8}$$

where the constant C_R controls the quantization. We use $C_R = 1.2$ in all our scenes; again this value has been experimentally derived by testing in the range $C_R \in [1.1,\ 2.5]$.

Lastly, we have to choose the cluster representative endpoints. As the only available geometric information are the two query vertices (Alg. 1, line 7), we simply choose this pair as the representatives. This is the optimal choice for the first path quantized to this cluster, and this choice also avoids the risk of self-shadowing. But it has another important consequence: the cluster representatives now depend on the processing order of the paths. We discuss how to avoid this correlation in Sect. 4.3.

3.5. Point and Directional Light Sources

To handle point, directional, and environment lights with visibility caching, we need special definitions of $K(.)$ for these light source types. A point light is represented by a location Y in space, while a directional light source is defined by a direction ω. For these lights, $K(\bar{p}_e, Y)$ (respectively $K(\bar{p}_e, \omega)$) returns a tuple of 10 integers: again 7 values are computed from $\bar{p}_e$ (Sect. 3.3), where we use Eq. 7 to determine the grid resolution. The three remaining components uniquely represent Y (resp. ω) and are obtained by bitwise reinterpreting the components of Y or ω as integers.

For environment maps, a light path $\bar{p}_l$ represents a differential solid angle ω chosen with probability $P(\omega)$. To compute $K(\bar{p}_e, \omega)$, following the argumentation from Sect. 3.2, we define $A(\bar{p}_l)$ as a patch on the unit sphere (i.e. the solid angle itself). We use octahedron mapping [ED08] to map solid angles onto the unit square. We then partition the latter using a regular 2D grid with a resolution of $R(\omega) \times R(\omega)$ dependent on $P(\omega)$. Finally, we choose the patch $A(\bar{p}_l)$ to be the grid cell that contains the mapped point of ω. $R(\bar{p}_e)$ is computed using Eq. 7 and there also follows that

$$R(\omega) = \sqrt{\frac{P(\omega)N_P}{4\pi N_S}}, \tag{9}$$

taking into account the unit sphere's surface area. Note that we cannot use Eq. 6 in this case, as it does not guarantee that $A(\omega) < 4\pi$. $R(\omega)$ is quantized using the same value for C_R as reported in Sect. 3.4. The result of $K(\bar{p}_e, \omega)$ then again

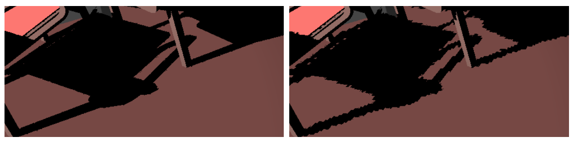

Figure 3: *An image rendered with (left) and without (right) adaptive refinement. Without refinement jagged shadow edges become visible under strong directional lighting.*

becomes a tuple of 10 integers, the first 7 of which computed from $\bar{p}_e$ as in Sect. 3.3. The next two are the coordinates of the chosen grid cell, followed by $R(\omega)$.

To summarize, we compute $K(.)$ depending on the semantic of its second argument: (1) point, (2) direction, (3) differential surface area, or (4) differential solid angle. The range of values can overlap even though the parameter domains are disjoint. To correctly store visibility information (Alg. 1, line 7), we prepend a two-bit index that specifies the semantics.

Algorithm 2 Compute $\tilde{V}(\bar{p}_e, \bar{p}_l)$ with adaptive refinement

```
1:  $V^c(j) \leftarrow$ an array initialized to nil on every frame
2:  for $d = 0 \rightarrow C_{MSR} - 1$ do
3:      $j \leftarrow K(\bar{p}_e, \bar{p}_l, C_E/2^d)$          ▷ $j$ is global variable
4:      if $V^c(j) =$ nil then
5:          $(X,Y) \leftarrow$ LASTVERTICES$(\bar{p}_e(j), \bar{p}_l(j))$
6:          $V^c(j) \leftarrow V(X, Y)$
7:          break
8:      end if
9:      $a \leftarrow$ true          ▷ See if all neighbours agree
10:     for $offset \in \{(1,0),(0,1),(-1,0),(0,-1)\}$ do
11:         $j_n \leftarrow j + (0,0,0,offset,0,\ldots)$
12:         $a \leftarrow a \wedge (V^c(j_n) =$ nil $\vee V^c(j_n) = V^c(j))$
13:     end for
14:     if $a$ then break
15: end for
16: return $V^c(j)$
```

3.6. Adaptive Refinement

The piece-wise constant approximation described above assumes that all paths passing through the same cluster have the same visibility. This obviously does not always hold in practice, in particular for paths in a cluster that pass near the edge of an occluder. In this case, our approximation introduces an error, which is mostly visible in the form of jagged shadow edges on the screen (see Fig. 3 right).

To alleviate this problem, we improve the quantization function to detect and to adapt to such non-uniform conditions. Whenever we resolve visibility using a cluster from the cache, we also examine the visibility of its neighboring clusters: If the paths in a cluster pass near the edge of a large occluder, we can expect that some of the surrounding clusters will report different visibility. In this case, we refine the approximation by reducing C_E for the current query and recurse. To locate neighboring clusters, we exploit the 3D grid cluster structure: by adding +1/-1 to the appropriate quantized locations in the tuple $K(\bar{p}_e, \bar{p}_l)$ we obtain the indices of each of the 64 neighbors (8 around $X \times 8$ around Y). This assumes a locally flat surface and that nearby clusters use the same grid resolution, but proved robust in our experiments.

Adaptive refinement makes most sense when the quantization clusters span multiple screen samples. In case of pixel super-sampling, the jagginess from the non-adaptive approximation is mostly invisible, as long as C_E^2 is smaller than the number of samples per pixel. This is why we only use adaptive refinement when generating the eye paths with Whitted-style ray tracing. For efficiency reasons, in our implementation (Alg. 2) we only explore the 4 neighbors around X. Because of the way we chose $R(\bar{p}_e)$ in Eq. 7 this roughly corresponds to exploring the $C_E \times C_E$ squares above, under, left, and right of the screen area of our cluster. We reduce C_E by a factor of $2\times$ at each recursion step, which roughly corresponds to halving each screen dimension of our cluster. The refinement depth is limited by a user specified parameter C_{MSR} which we set to $\log_2 C_E$ by default. Note that if no visibility information about the neighbors is present, we do not trace rays, as we have no knowledge about the surfaces in these regions. Thus, the adaptive refinement efficiency is sensitive to the path sampling order (discussed in Sect. 4.3).

4. Rendering with Visibility Caching

In this section we discuss the use of visibility caching in rendering algorithms. We discuss important practical issues, including peculiarities in CPU and GPU implementations and the renderers that we use to demonstrate them.

4.1. Cache Storage

Recall from Sect. 3.3 that V^c is defined over a 15-dimensional domain: the 14 integers of $K(.)$ and the prepended semantic index. Thus, it cannot be easily stored in an array. For this reason, and to exploit its sparseness, we store it in a hash map (similarly to [DS07]). To index into the hash map, we compute a 32-bit key $\kappa(j)$ from $j = K(.)$ (defined below) and take its modulo with the hash map size C_T, known as the division method. We do not resolve collisions, and simply overwrite the data instead. While this can lead to redundant cluster visibility computations, it has the advantage of a fixed storage size. To avoid storing the whole tuples j, a 31-bit message digest $c(j)$, which can be seen as a checksum, is prepended to the 1-bit “payload”, the visibility $V^c(j)$. The hash map then becomes a 32-bit integer array with size C_T.

We use two approaches to compute $\kappa(j)$ and $c(j)$. The first one uses two linear congruential generators C_1 and C_2 [PTVF07, Section 7.1], and takes $\kappa(j) = C_1(\kappa_{16}(j))$ and

$c(j) = C_2(c_{16}(j))$ from the recurrence relations

$$\begin{aligned} \kappa_{n+1}(j) &= C_1(\kappa_n(j)) \text{ xor } j(n) \\ c_{n+1}(j) &= C_2(c_n(j)) + j(n). \end{aligned} \tag{10}$$

The second approach computes a 64-bit digest from $d(j)$ and takes the upper 32 bits for $\kappa(j)$ and the lower 31 bits for $c(j)$. It combines a linear congruential generator C with a xor-shift generator A, using $d(j) = A(C(d_{16}(j)))$ with the recurrence relation $d_{n+1}(j) = A(C(d_n(j)))$ xor $j(n)$. The first approach generates more collisions than the second one, but is cheaper to compute and was faster on the CPU due to its large caches. On the GPU, where the cheaper arithmetic operations make collisions more expensive, the second approach was faster.

4.2. Rendering Algorithms

We integrated our visibility caching into two different renderers, one based on bidirectional path-tracing (BPT), and the other one on instant radiosity. For BPT, we use quantization without adaptive refinement, and cache visibility for the connecting segments of all eye and light sub-paths.

The second renderer combines Whitted-style eye path tracing, instant radiosity and Monte Carlo direct illumination sampling. We first store eye paths in deep deferred buffers, then generate VPLs, resampled to a user-defined number as in [SP06], and finally shade the eye samples using adaptive refinement to avoid artifacts from clusters spanning multiple pixels. We handle direct lighting *independently* for each pixel, and also use a multiple importance sampling combination of BRDF and environment power sampling [Vea98].

4.3. Pixel Enumeration Order

When using Whitted-style ray tracing, scanline pixel enumeration can cause the approximation error to be more apparent, due to the correlation between the cluster representative assignment and the eye sample processing order. A similar problem also occurs in irradiance caching [WRC88], and in our case it can result in jaggy shadow edges and shadows shifted to the bottom-right, as seen in Fig. 4 left. The scanline processing order also reduces the effectiveness of our adaptive refinement, as the upper and left neighbors of a cluster (from a camera perspective) will be in the cache most of the time, while the bottom and right ones will not.

To avoid these unwanted correlations, we traverse the pixels in an order both coherent and irregular. Besides for improved quality, this is also important with regard to the massive parallelism of GPUs (as discussed below). To this end, we enumerate all pixels in the order they would be visited by a 2D Halton sequence [Hal64] with bases 2 and 3. We then exploit the property of this sequence that any continuous sub-sequence will cover the whole domain with low discrepancy: we partition the generated order into segments, choosing 0, C_{PI}, $C_{PI}C_{PG}$, $C_{PI}C_{PG}^2$, etc. as boundaries, and sort the elements within each segment using Morton codes [Mor66].

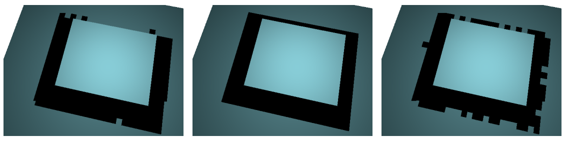

Figure 4: *The effect of a scan line processing order (left image) on the approximation error for non-adaptive quantization. The shadows from the cube appear shifted towards the bottom right. The right image uses random order, and the middle one is ground-truth.*

By this the screen is progressively covered by the segments, each with a density C_{PG}^2 times higher at each iteration, and the pixels within one segment are visited in a coherent order. Experimentally, we have found $C_{PI} = T/(C_E)^2$ (where T is the number of pixels on the screen) and $C_{PG} = 3.7$ to work best across our test scenes.

4.4. Implementation Details

Our bidirectional path tracer runs on the CPU only, while we have both CPU and GPU versions of the second renderer that combines Whitted-style ray tracing with instant radiosity.

Thread-safety and race conditions. Our implementation of the visibility cache is thread safe, i.e. even if two or more threads work on the same element of V^c, the final result will still be correct, as these elements are simple 32-bit integers. Such non-critical race conditions can, however, negatively impact performance. Fortunately, the pixel processing order from Sect. 4.3 helps alleviate this problem: when two threads process pixels close enough to each other to potentially cause a conflict, then many of the surrounding pixels will have been already visited. Thus, visibility will most likely be resolved from the cache, avoiding race conditions.

CPU implementation. On CPUs, which are characterized by moderate MIMD parallelism and large per-core caches, we parallelize rendering by dividing the screen into tiles of 32×32 pixels. For the Whitted-style renderer, we assign a separate visibility cache to each thread. We keep its size low at 4MB, since the number of different cache requests inside one tile is low, and this size also ensures that the visibility cache mainly stays in the CPU caches. For the adaptive refinement we compute $K(j)$, $c(j)$ and $\kappa(j)$ for 4 levels at once leveraging SSE instructions. For the bidirectional path tracer, we use one large shared cache of 1GB.

GPU implementation. On the GPU, we use the ray tracing kernels of Aila and Laine [AL09]. We assign each thread to one pixel and schedule the work using persistent threads. We use a global cache of 256MB. In all our scenes, a larger cache increased the rendering performance only slightly, e.g. a 1GB cache resulted in $\approx$5% faster rendering times.

There is one very important aspect in the efficiency of the GPU implementation: a 32-wide SIMD warp always takes as long as tracing 32 rays, even if only one of its threads really

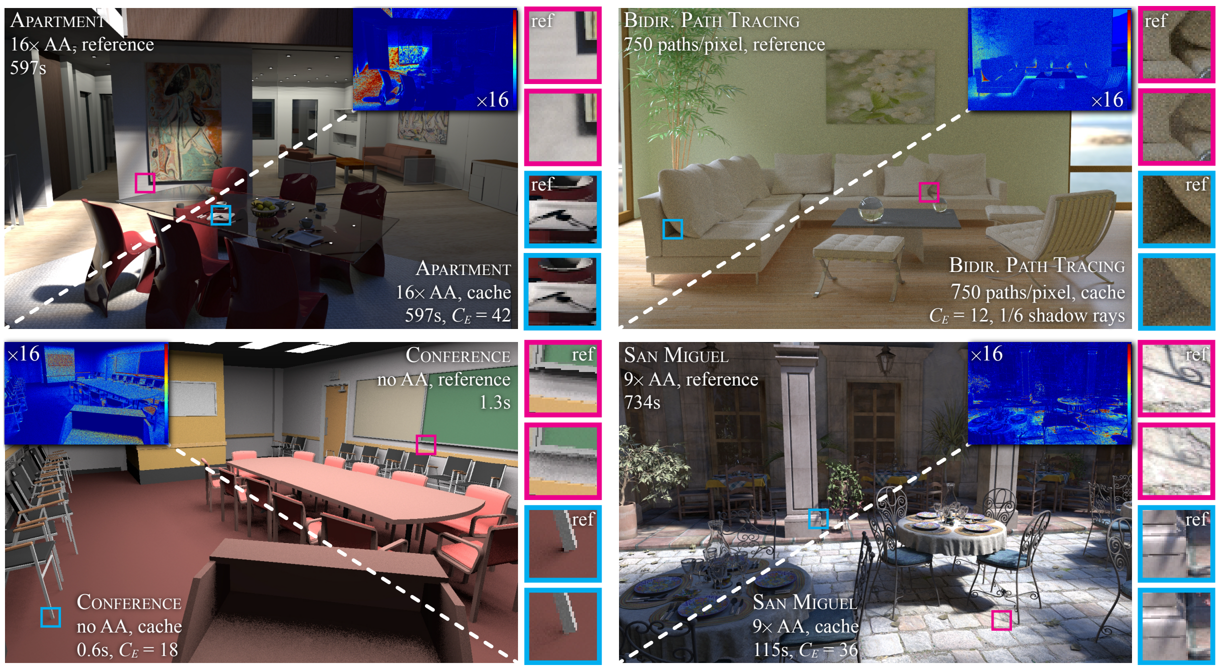

Figure 5: *We compare reference images computed with exact visibility (above the diagonal) to images rendered with our visibility caching (below the diagonal). The total render times are given in seconds, the difference images show the 16× amplified luminance error. The two close-ups for each scene highlight the visual differences. The rendering setup is described in Sect. 5.*

traces a ray. In visibility caching, it often happens that only a few threads cast rays and the rest draw results from the cache. We solve the problem by using a mechanism to postpone computation: every time we are about to shoot a ray, we count how many other threads in the warp want to trace rays as well. If the number is below a threshold C_W, we save the local state of each thread in a small per-warp queue with 64 entries and do not trace rays immediately; all other threads in the warp use the cached results and proceed with connecting the eye and light paths. Whenever the number of elements in the queue exceeds 32, we finish the postponed work for its first 32 entries by shooting the rays, filling V^c, and connecting the paths. We found that high thresholds provide the best performance (we use $C_W = 31$), as the overhead of postponing work is small and tracing with even one thread less than the warp measurably increases rendering time.

5. Results

In this section we benchmark the practical performance of our visibility caching algorithm. We render several scenes with the renderers described in Sect. 4.2 and compare them to exact visibility reference images. The focus of our analysis is on rendering performance and quality degradation.

Scenes. We show results for the following scenes (Fig. 5, 6):

- Apartment consists of ≈750K triangles and is illuminated by 24 omni-directional point lights, 1024 VPLs, and 256 environment map samples.
- Apartment Delta is the same scene as above, but illuminated by 24 omni-directional point lights and 1 directional light source.
- Conference consists of ≈280K triangles and is illuminated by overhead area lights with 64 samples.
- San Miguel is our largest scene, consisting of ≈10M triangles; illumination is computed with 256 environment samples and 1024 VPLs.
- Hairball has fine geometry with ≈3M triangles, and is illuminated by 64 environment samples and 256 VPLs.

We rendered all scenes with and without anti-aliasing (AA) on the GPU, and also Apartment and Apartment Delta without AA on the CPU. AA employs 16× supersampling, except for 9× on San Miguel and Hairball due to insufficient GPU memory for both the large model and the deferred shading buffers. We perform all our measurements on an Intel Core i7 2600K CPU with an NVIDIA Quadro 6000 GPU at resolution 1280 × 800 (and 1024^2 for Hairball). We also achieved an absolute performance speedup of 2× over the Quadro on a GTX 680 GPU, but not all rendering setups could fit into its smaller memory. We therefore report results from the former GPU only.

Range of $\mathbf{C_E}$. We rendered each scene with different values for C_E to assess the influence and choice of this parameter. Super-sampling allows for a wider range for C_E, as it increases the image plane density, effectively decreasing the area of influence of any given cluster on the screen. For point

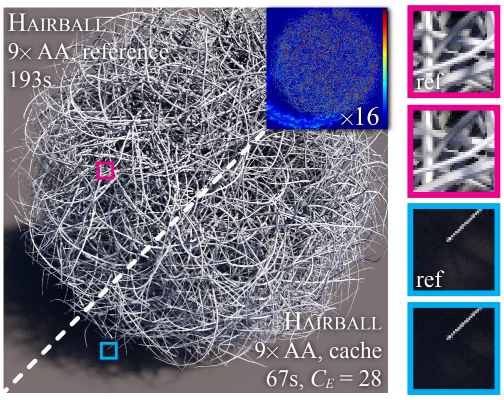

Figure 6: *The fine geometry of* HAIRBALL *reduces the achieved ray reduction. Nevertheless, we achieve rendering acceleration of* 3× *due to the high ray intersection cost.*

and directional light illumination (on APARTMENT DELTA), we choose $C_E \in [4,36]$ and $C_E \in [4,18]$ for images with and without anti-aliasing respectively. Since VPL, environment map, and area light illumination is generally smoother, we use $C_E \in [10,64]$ and $C_E \in [4,24]$ respectively. For APARTMENT we fix $C_E = 4$ for the omni-directional lights, and vary C_E for other light sources. This improves image quality without a measurable performance penalty.

Quality metrics. To assess rendering quality, we use the perception driven metric PREDICTED QMOS, proposed by Mantiuk et al. [MKRH11]. It measures the quality of a "distorted" picture compared to a given reference as mean opinion score. We also measure the change of energy L^INF, as the ratio of the sum of absolute differences of the two images' pixels and the total energy of the reference image.

Performance metrics. To assess the performance benefit of using visibility caching, we analyze (1) the shadow ray reduction (named RAY REDUCTION in the performance charts in Fig. 7), and (2) the total frame rendering time (named SPEEDUP). The first can be seen as an upper bound for the speedup we can expect, while the second shows the actually achieved speedup with our implementation. Both are given relative to a reference implementation using exact visibility. The measured frame times include all phases of the rendering algorithm, i.e. eye path and VPL generation (which typically take little time compared to shading).

Cache performance can also be derived from Fig. 7 as $(1 - HitRatio) = MissRatio = 1/(RayReduction + 1)$. Note that cache misses occur also due to collisions, but their frequency can only be measured accurately in the single-threaded case, where they were well below a few percent for all scenes. With multiple threads the measurement itself changes the cache access pattern and thus the collision rates.

Relative performance results for the different values of C_E are also summarized in Fig. 7. Absolute frame times are given in Fig. 5 and 6, where we compare reference images to results obtained with a value for C_E that (we think) yields the optimal trade-off between quality and performance. For reasons explained above, we present results obtained with anti-aliasing separately in Fig. 7. Our algorithm achieves significant shadow ray reduction, up to 50×, while mostly preserving the rendered image quality (QMOS remains high, above 77%). Note that, due to numerical precision issues in the original authors' implementation used, QMOS cannot exceed 93.4%, even when comparing an image to itself.

5.1. Quality Degradation

Our caching method substitutes path visibility with cluster visibility. This approximates the energy transferred to the screen by all paths $\bar{p}$ quantized into a given cluster:

$$I^c = \sum V(X,Y) f(\bar{p}) \approx V^c \sum f(\bar{p}), \tag{11}$$

where $f(\bar{p})$ is the unoccluded contribution of $\bar{p}$, and X and Y are the path connection points. The cluster visibility $V^c \in \{0,1\}$ is a binary random variable, whose value is determined by the cluster representative. The probability that $V^c = 1$ is equal to the average path visibility:

$$\Pr(V^c = 1) = V^c_{\text{avg}} = \frac{\int_{A_l} \int_{A_e} V(X,Y)\, dX dY}{|A_l||A_e|}, \tag{12}$$

with A_e and A_l being the patches defining the cluster, where for simplicity we have assumed that all paths in the cluster have the same sampling density. Clusters with $V^c_{\text{avg}} < 0.5$ will on average reduce the brightness of the affected screen regions, while the rest will increase it. The CONFERENCE zoom-ins in Fig. 5 show examples of the former case. The blue zoom-ins of the HAIRBALL in Fig. 6 demonstrate the latter case. When a single cluster contributes the entire illumination of each pixel, cluster shapes become visible, resulting in jaggies (Fig. 8). Note that visibility caching produces correct results for clusters whose two patches are mutually fully (un)occluded, which is often the case in our test scenes.

5.2. Speedup

The total rendering speedup is expectedly smaller than the shadow ray reduction: besides visibility, the frame time includes shading which remains unchanged. Moreover, cache lookups also incur some cost, and the actual shadow rays traced are much less coherent and thus more expensive. On the other hand, the speedup we achieve is close to optimal on most scenes. The ratio of shading to total frame time on the GPU for the reference solution is approximately 12 for APARTMENT, 18 for APARTMENT DELTA, 5 for CONFERENCE, 37 for SAN MIGUEL, and 26 for HAIRBALL. Thus, any other algorithm cannot be faster than ours by more than 2.5×, 3.4×, 2.1×, 4.7×, and 6.7× respectively, even if it completely eliminates the visibility query cost.

As a rule of thumb, the higher the ray tracing cost, the more benefit can be expected from caching. We did not see

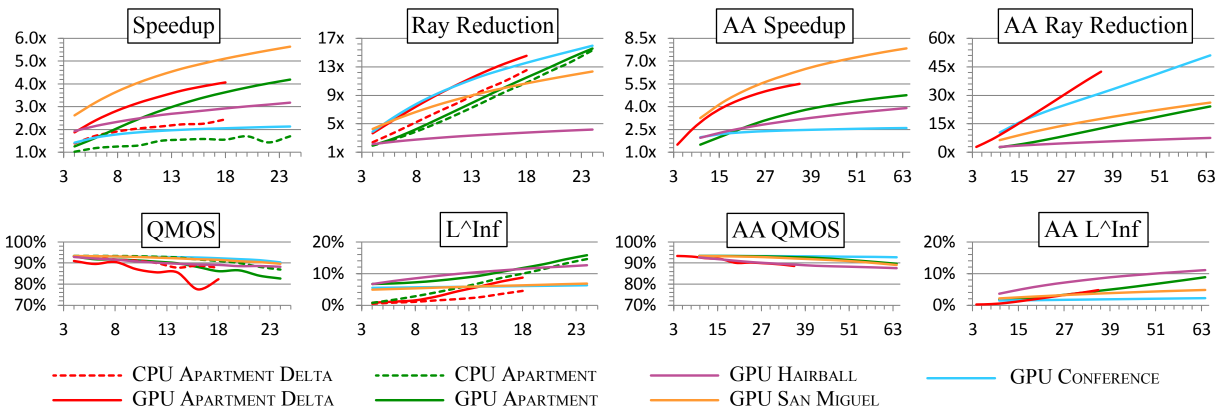

Figure 7: *Performance characteristics of rendering with visibility caching as a function of C_E ("AA" denotes anti-aliasing).*

any speedup on simple scenes (e.g. Cornell box), while the large unoccluded regions in Conference result in low ray cost and low speedup (Fig. 7). The more complex occlusion in Apartment results in higher speedup. San Miguel exhibits the largest speedup, due to the especially high cost of shadow rays that pass through the fine geometry of vegetation models. The ray cost is very high in Hairball as well, but the high frequency geometry around the eye samples limits the size of the cluster patches and thus the ray reduction; the speedup is therefore almost identical to the ray reduction.

The higher relative shading cost on the CPU leads to a lower speedup than on the GPU. In Apartment shading takes 32% of the CPU frame time, i.e. we cannot expect to accelerate rendering by more than $3\times$ in this case.

The top-right scene in Fig. 5 has been rendered with bidirectional path tracing using visibility caching, without adaptive refinement. Since shadow ray casting in BPT takes only a small fraction of the frame time, we use this scene to assess the applicability of visibility caching to path tracing algorithms. We used $C_E = 12$, which reduced the shadow ray count by $\approx 6\times$. Similarly to the other results, quantization errors are mostly visible around small-scale shadow regions.

5.3. Interactive Performance

Although our method is meant mainly for interactive preview rendering (several seconds per frame), we have showed images with longer computation times until now. We did this to faithfully show the artifacts of visibility caching, without distractions due to noisy undersampled illumination. By lowering the sampling settings, we have achieved interactive walkthroughs in the Apartment and San Miguel scenes at approx. 2 FPS with acceptable image quality, as demonstrated in the accompanying video. Interactive performance of our method can also be observed on Apartment Delta (3 FPS) and Conference (1.5 FPS).

A typical disadvantage associated with any form of subsampling is temporal flickering. In our case, adaptive refinement almost entirely eliminates it. High-frequency temporal noise is visible mostly on hard shadows with higher than average values for C_E and sometimes on soft shadows for high values of C_E, as we show in the accompanying video. In any case, low frequency flickering due to wrong visibility in large surface regions is unlikely to occur, since artifacts are smaller than $C_E \times C_E$ pixels on the screen.

5.4. Comparison to Shadow Maps

Visibility caching suggests a comparison to shadow mapping. Fig. 8 provides a comparison to plain shadow mapping on Apartment Delta for varying C_E values and shadow map resolutions. The quality achieved by hardware shadow mapping, even at resolution 6000×6000, is inferior to visibility caching with moderate C_E values. Surprisingly, on the same hardware, shadow mapping was *slower* than our method. At 1024×1024, the lowest resolution with acceptable quality, it performed $2\times$ slower than our algorithm with $C_E = 14$. Expectedly, shadow mapping performs better for point light sources. Still, visibility caching remains superior for close-range viewpoints, thanks to its automatic screen-space error control.

We only compare to plain GPU shadow mapping, as the presence of refractive objects prevents the use of warping and partitioning techniques [SWP11], while focusing and culling [BMSW11] have to be modified to compute the bounds of the deferred eye samples. Furthermore, focusing and culling make little sense in our test scene, since the eye samples span almost the complete volume of the scene.

6. Conclusion and Future Work

We presented a novel visibility caching algorithm for ray tracing that reduces the number of shadow rays by up to $50\times$, and accelerates global illumination rendering by only slightly affecting image quality. Our algorithm is scalable w.r.t. scene complexity and its effectiveness increases with

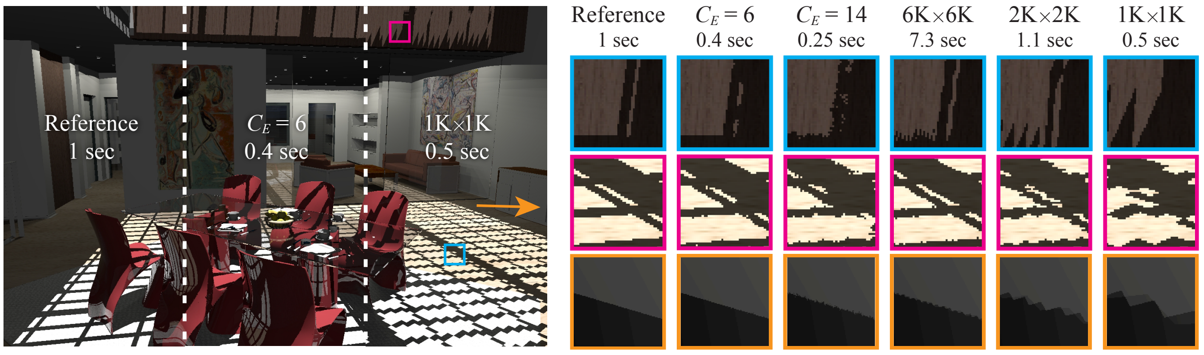

Figure 8: *Our visibility caching (columns 2, 3) compared to shadow mapping (columns 4-6) on* APARTMENT DELTA. *Regions with large errors are magnified. The bottom row of zoom-ins is from a different view point indicated by the orange arrow.*

larger and more complex scenes. The method supports arbitrary visibility queries and naturally handles illumination from point and area lights as well as environment maps.

As future work, we plan to incorporate our method into other rendering algorithms. Lightcuts [WFA*05] seems particularly interesting, as nearby pixels typically share many VPLs and thus exhibit great potential for accelerating the rendering. We would also like to extend our algorithm to support participating media, where the main challenge would be the adaptive refinement algorithm, as it is no longer easy to compare the visibility of neighboring grid cells.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics* (2009), pp. 145–149. 6

[BMSW11] BITTNER J., MATTAUSCH O., SILVENNOINEN A., WIMMER M.: Shadow caster culling for efficient shadow mapping. In *Proc. I3D* (2011), ACM, pp. 81–88. 9

[BW03] BITTNER J., WONKA P.: Visibility in computer graphics. *Environment and Planning B: Planning and Design 30*, 5 (2003), 729–756. 2

[COCSD03] COHEN-OR D., CHRYSANTHOU Y. L., SILVA C. T., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 09*, 3 (2003), 412–431. 2

[DS07] DIETRICH A., SLUSALLEK P.: Adaptive spatial sample caching. In *Proc. IEEE/EG Symposium on Interactive Ray Tracing 2007* (September 2007), pp. 141–147. 2, 5

[ED08] ENGELHARDT T., DACHSBACHER C.: Octahedron environment maps. In *Proc. VMV* (2008), pp. 383–388. 4

[Hal64] HALTON J. H.: Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of ACM 7* (1964). 6

[HG86] HAINES E., GREENBERG D.: The light buffer: A shadow-testing accelerator. *IEEE Computer Graphics and Applications 6*, 9 (1986), 6 –16. 2

[HPB07] HAŠAN M., PELLACINI F., BALA K.: Matrix row-column sampling for the many-light problem. *ACM Transactions on Graphics (Proc. SIGGRAPH) 26*, 3 (2007). 2

[Ige99] IGEHY H.: Tracing ray differentials. In *Proc. SIGGRAPH'99* (1999), pp. 179–186. 3

[Kel97] KELLER A.: Instant radiosity. In *SIGGRAPH '97* (1997), pp. 49–56. 2

[LW93] LAFORTUNE E. P., WILLEMS Y. D.: Bi-directional path tracing. In *Proc. SIGGRAPH'93* (1993), pp. 145–153. 1

[MKRH11] MANTIUK R., KIM K. J., REMPEL A. G., HEIDRICH W.: HDR-VDP-2: a calibrated visual metric for visibility and quality predictions in all luminance conditions. *ACM Transactions on Graphics (Proc. SIGGRAPH) 30*, 4 (2011). 8

[Mor66] MORTON G. M.: *A computer oriented geodetic data base and a new technique in file sequencing*. Tech. rep., IBM Ltd., 1966. 6

[ND12] NOVÁK J., DACHSBACHER C.: Rasterized bounding volume hierarchies. *Comptuter Graphics Forum 31*, 2 (2012). 1

[PTVF07] PRESS W. H., TEUKOLSKY S. A., VETTERLING W. T., FLANNERY B. P.: *Numerical Recipes 3rd Edition*, 3 ed. Cambridge University Press, Sept. 2007. 5

[RDGK12] RITSCHEL T., DACHSBACHER C., GROSCH T., KAUTZ J.: The state of the art in interactive global illumination. *Comptuter Graphics Forum 31*, 1 (2012), 160–188. 2

[SP06] SEGOVIA B., PÉROCHE J.-C. I. B.: Bidirectional instant radiosity. In *Proc. EGSR* (2006). 6

[SWP11] SCHERZER D., WIMMER M., PURGATHOFER W.: A survey of real-time hard shadow mapping methods. *Computer Graphics Forum 30*, 1 (2011), 169–186. 2, 9

[TPWG02] TOLE P., PELLACINI F., WALTER B., GREENBERG D. P.: Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics (Proc. SIGGRAPH) 21*, 3 (2002). 2

[Vea98] VEACH E.: *Robust monte carlo methods for light transport simulation*. PhD thesis, Stanford University, Stanford, CA, USA, 1998. Adviser-Guibas, Leonidas J. 3, 6

[WDP99] WALTER B., DRETTAKIS G., PARKER S.: Interactive rendering using the render cache. In *Proc. Eurographics Workshop on Rendering* (1999), pp. 235–246. 2

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 24*, 3 (2005), 1098–1107. 1, 2, 10

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. *Computer Graphics (Proc. SIGGRAPH) 12*, 3 (1978). 2

[WRC88] WARD G. J., RUBINSTEIN F. M., CLEAR R. D.: A ray tracing solution for diffuse interreflection. *Computer Graphics (Proc. SIGGRAPH) 22*, 4 (1988), 85–92. 2, 6