Constant-Time Ray-Scene Intersection for Interactive Rendering with Thousands of Dynamic Lights

Journal:	*Transactions on Graphics*
Manuscript ID	Draft
Manuscript Type:	Previously Unpublished Research Paper
Date Submitted by the Author:	n/a
Complete List of Authors:	wang, lili; Beihang University, State Key Laboratory for Virtual Reality Technology and Systems, School of Computer Science and Engineering Meng, Chunlei; Beihang University, State Key Laboratory for Virtual Reality Technology and Systems, School of Computer Science and Engineering Popescu, Voicu; Purdue University, Computer Science
Computing Classification Systems:	Real time rendering, Many lights, Visibility determination, Photorealism
Note: The following files were submitted by the author for peer review, but cannot be converted to PDF. You must view these files (e.g. movies) online.	
manylights-93M.mp4	

Constant-Time Ray-Scene Intersection for Interactive Rendering with Thousands of Dynamic Lights

LILI WANG and CHUNLEI MENG
Beihang University
and
VOICU POPESCU
Purdue University

We present a method for constant time approximation of the intersection between a ray and the geometry of a scene. The scene geometry is approximated with a 2D array of voxelizations, with one voxelization for each direction from a dense sampling of the 2D space of all possible directions. The ray/scene intersection is approximated using the voxelization whose rows are most closely aligned with the ray. The voxelization row that contains the ray is looked up, the row is truncated to the extent of the ray using bit operations, and a truncated row with non-zero bits indicates that the ray intersects the scene. We support dynamic scenes with rigidly moving objects by building a separate 2D array of voxelizations for each type of object, and by using the same 2D array of voxelizations for all instances of an object type. We support complex dynamic scenes and scenes with deforming geometry by computing and rotating a single voxelization on the fly. We demonstrate the benefits of our method in the context of interactive rendering of scenes with thousands of moving lights.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Raytracing*

Additional Key Words and Phrases: Real time rendering, Many lights, Visibility determination, Photorealism

1. INTRODUCTION

Many scenes of interest to computer graphics applications contain a large number of dynamic light sources. Whereas the interactive computer graphics pipeline and its hardware implementation can now handle scenes with complex geometry modeled with millions of triangles, the number of lights supported in interactive rendering has not increased at a similar pace. Providing support for a large number of light sources is an important way of improving the quality of imagery rendered at interactive rates.

Lighting is computationally expensive because it implies solving a visibility problem for every point light source. Even though GPUs can now render a complex scene multiple times per frame, rendering power is not sufficient for a brute force approach that renders the scene once for everyone of thousands of lights.

In this paper we propose a method for interactive rendering with thousands of dynamic lights.Given an output image of $w \times h$ resolution and a scene with n point light sources, one has to intersect the scene geometry with $w \times h \times n$ rays from the output image samples to the lights. For a $1,024 \times 1,024$ resolution and 10,000 lights the number of intersections is 10 billion. Our method is based on an acceleration scheme that enables a constant-time intersection between a light ray and the scene geometry. The scene geometry is approximated with a 2D array of voxelizations corresponding to the 2D space of ray directions. The intersection between the scene geometry and a ray is approximated by intersecting the ray with the voxelization whose rows are most closely aligned with the ray. The row of the voxelization that is traversed by the ray is looked up, and the 1-D intersection between the row and the ray is computed with bit-shift operations. A voxelization stores one bit per voxel so a voxelization row is looked up and bit-shifted in constant time.

The constant time ray-scene intersection enables rendering with thousands of dynamic lights at interactive rates (Figure 1). Our method brings a substantial speedup over ray tracing at the cost of a small quality trade-off. Our method approximates the ray-scene intersection by approximating the scene geometry, which is independent of the lights. Consequently the lights can change freely from frame to frame at no additional cost.

Dynamic geometry affects the precomputed array of voxelizations. We support dynamic scenes in one of two ways. For scenes with rigid dynamic objects, an array of voxelizations precomputed for a moving object can be reused by transforming the light ray to the local coordinate system of each instance of the moving object. The example shown in Figure 2 left uses two arrays of voxelizations, one for the city and one for the airplane, and three lookups per ray, one for the city, and one for each of the two instances of the airplane. For scenes with deforming geometry (e.g. the running bear shown in Figure 2 middle), or for complex dynamic scenes (e.g. the amusement park shown in Figure 2 right), the array of voxelizations is approximated for every frame by computing one voxelization and rotating it with two degrees of freedom, which is substantially less expensive than computing every rotated voxelization from the original scene geometry.

In summary, our paper contributes a method for constant time approximation of the intersection between a ray and a scene's geometry. The approximation of the ray-scene intersection proceeds at two levels. At the first level, scene geometry is approximated us-

Authors' addresses: land and/or email addresses.

DOI 10.1145/1559755.1559763
http://doi.acm.org/10.1145/1559755.1559763

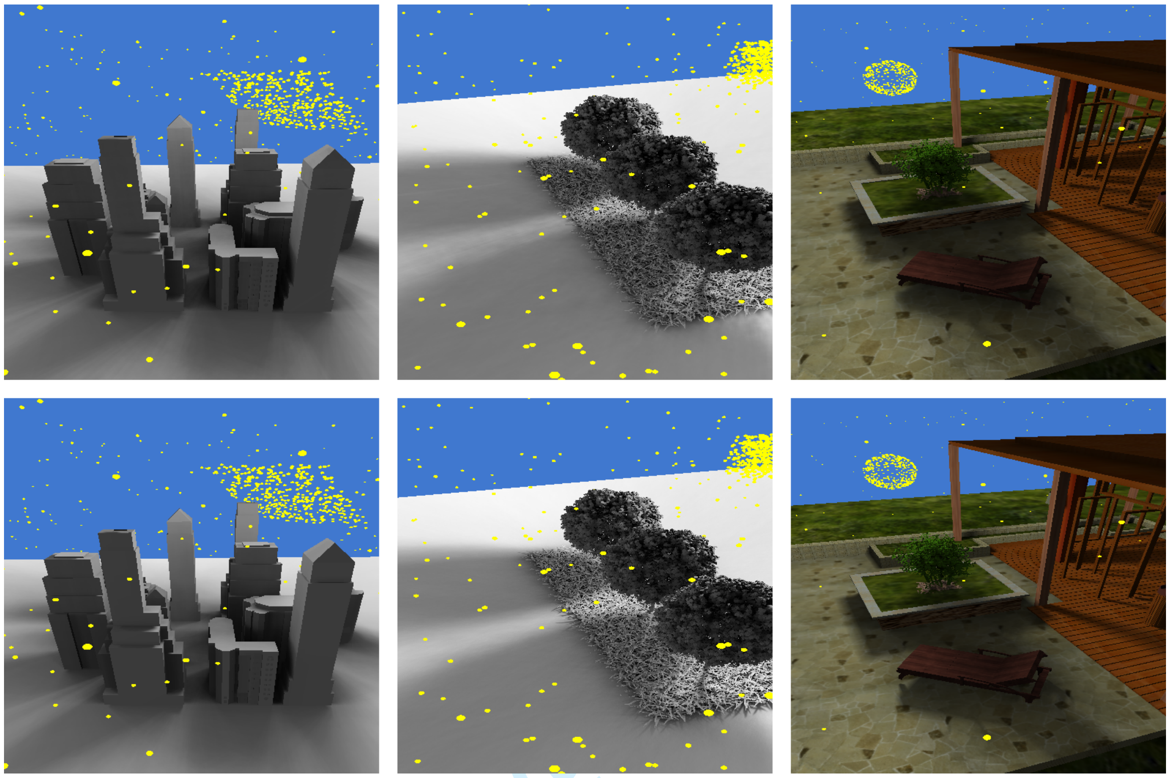

Fig. 1: Scenes with 1,024 lights rendered with our method (top), and with ray tracing (bottom). Our average pixel shadow value errors are 0.5%, 2.1%, and 2.2%. Our frame rates are 23fps, 22fps, and 16fps, which corresponds to speedups of 38×, 76×, and 38× versus ray tracing (i.e. NVIDIA's Optix with BVH acceleration).

ing a conventional voxelization. This is not enough, however, since tracing a ray diagonally through the voxelization is still too expensive. At the second level, the ray direction is approximated by discretizing the 2D space of possible ray directions, and by voxelizing the scene geometry for each discrete ray direction. This way a ray is always intersected with a voxelization whose rows are aligned with the ray, which avoids the expensive diagonal traversal of the voxelization. We demonstrate the benefits of our method in the context of interactive rendering of scenes with thousands of lights. Compared to ray tracing, our method brings substantial performance gains at the cost of small approximation errors. Compared to prior techniques for interactive rendering with many lights, our method brings a significant quality increase for the same performance.

2. PRIOR WORK

The need to estimate visibility to a large number of light sources arises both in the case of the direct illumination of scenes with complex lighting, and in the case of global illumination where scene geometry samples turn into secondary light sources. The classical methods for computing visibility to a light source are shadow mapping and ray tracing. However, these methods are too slow for scenes with a large of number of lights. Acceleration was pursued along two main directions: scene geometry approximation, to reduce the cost of estimating visibility to a light source, and light clustering, to reduce the number of lights. In addition to the brief overview of prior work given below, we also refer the reader to a recent survey of techniques for rendering with a large number of lights [Dachsbacher et al. 2014].

2.1 Shadow map based methods

Shadow mapping is the approach of choice for interactive rendering with a small number of lights. A shadow map is a view-dependent approximation of scene geometry that can be naturally computed on the GPU. However, rendering a shadow map for each one of a large number of lights is too slow.

Ritschel et al. introduced Coherent Shadow Maps [Ritschel et al. 2007], which are compressed, orthographic depth maps precomputed for n viewing directions, with n much smaller than the number of lights. For each ray, visibility is approximated using the shadow map with the view direction closest to the direction of the ray. The method was extended to Coherent Surface Shadow Maps (CSSM) [Ritschel et al. 008a] to support light sources on scene geometry as needed for indirect lighting. A virtual area light [Dong et al. 2009] is a group of virtual point light sources, and the visibility to a virtual area light is computed using CSSMs extended with parabolic projection, which avoids having to compute visibility to each individual virtual point light source.

Imperfect Shadow Maps (ISM) [Ritschel et al. 008b] is a technique that renders one shadow map for each point light source. To achieve interactive performance the resolution of the shadow maps is low, and the shadow maps are rendered from a point-based approximation of scene geometry by splatting followed by pull-push reconstruction. ISM is a frequently used method for interactive ren-

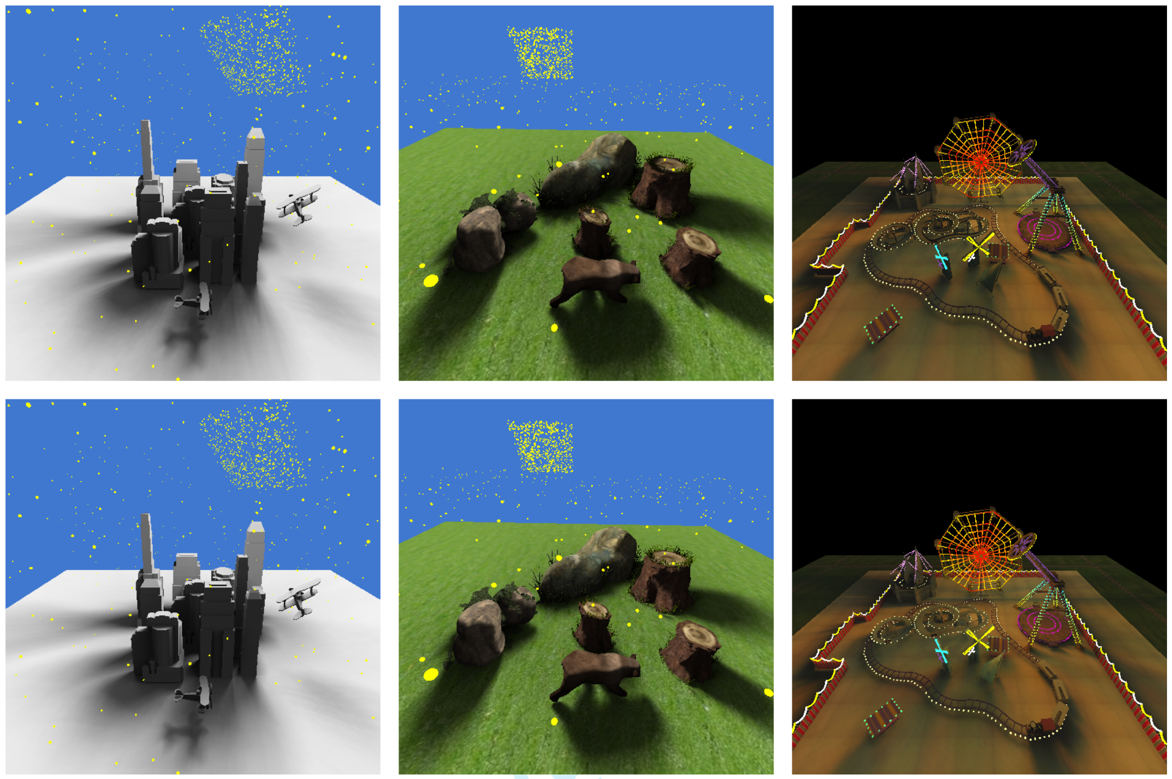

Fig. 2: Dynamic geometry scenes with 1,024, 1,024, and 7,088 lights, rendered with our method (top), and with ray tracing (bottom). Our average pixel shadow value errors are 0.8%, 2.1%, and 7.5%. Our frame rates are 11fps, 2fps, and 1fps, which corresponds to speedups of 22×, 6×, and 34× versus ray tracing.

dering with many lights, so we compare our method to ISM in detail in the Results Section. Hierarchical scene geometry approximations have been used to accelerate shadow map computation. For example, Implicit Visibility [Dong et al. 2007] uses a disk-based quadtree surface approximation, and ManyLoDs [Hollander et al. 2011] uses a cut through a bounding volume hierarchy of the scene geometry. Virtual shadow maps [Olsson et al. 2014] partition shadow casting scene geometry into clusters for which cube maps of appropriate resolution are rendered, achieving interactive performance for complex scenes with hundreds of lights.

Matrix Row-Column Sampling (MRCS) [Hašan et al. 2007] uses the matrix of all possible output sample/light points pairs to determine output sample and light clusters for which to compute a set of representative shadow maps. The visibility of individual sample/light pairs is interpolated from a few relevant representative shadow maps. A subsequent light clustering method reduces the number of representative shadow maps [Davidovič et al. 2010]. The MRCS algorithm is mapped to an out-of-core GPU implementation [Wang et al. 2013], and its efficiency is improved by reducing the number of visibility estimates between representative light clusters and output image samples [Huo et al. 2015].

The use of a large number of shadow maps to approximate visibility to a large set of lights has the limitation of redundancy between shadow maps constructed from nearby viewpoints or with similar orthographic view directions. The higher the complexity of the scene and the higher the number of lights, the higher the redundancy. Our method uses a 3D approximation of the scene (i.e. a voxelization) which captures multiple layers of occlusion without redundancy. Our method introduces redundancy by computing a 2D array of voxelizations, which is needed to achieve the constant-time ray-scene intersection. However, for our method, redundancy is bounded by the discretization of all possible ray directions, and it does not increase with the number of lights or with the complexity of occlusions in the scene. Our method does not cluster lights, but rather computes visibility to each one of the scene light sources, which provides good shadow quality.

2.2 Ray tracing based methods

Several techniques accelerate ray tracing visibility computation using scene geometry approximation. Micro-rendering [Ritschel et al. 2009] approximates geometry with a point hierarchy which accelerates ray traversal and geometry updates for dynamic scenes. Ray tracing was also accelerated using geometry voxelization [Nichols et al. 2010]. Our method also relies on geometry voxelization, and we further reduce the cost of ray-scene intersection using a 2D array of voxelizations.

Many ray tracing methods focus on simplifying the set of lights. An octree light hierarchy was used to cluster lights based on their positions and their spheres of influence [Paquette et al. 1998]. Lights were grouped in an unstructured light cloud and the light vectors at each vertex are compressed using PCA, which achieves high quality and high frame rates for low-frequency lighting environments [Kristensen et al. 2005]. Lightcut [Walter et al. 2005] is a popular method for shading with many lights based on clustering scene lights in a binary tree. A cut through the tree is selected for each output sample, under the assumption that all lights are visible. The method is extended to include visibility computation, i.e. to account for shadows, in Precomputed Visibility Cuts [Akerlund et al. 2007] and in Nonlinear Cut Approximation [Cheslack-Postava et al. 2008], which are suitable for static scenes, and then in Bidirectional Lightcuts [Walter et al. 2012], which can also handle dynamic scenes.

Some ray tracing based methods approximate both the lights and the scene geometry. VisibilityClusters [Wu and Chuang 2013] group geometry and lights using a sparse matrix whose non-zero submatrices correspond to visibility interactions between geometry clusters and light clusters. Such methods trade off approximation construction complexity for approximation efficiency, which pays of for scenes with non-uniform light and geometry complexity.

Our method is essentially a ray tracing acceleration method. Unlike the light clustering ray tracing acceleration schemes discussed above, our method does not reduce the number of rays by reducing the number of lights. This brings a quality advantage since we estimate visibility for each light individually. Moreover, this also ensures good temporal coherence in the case of dynamic lights, where each light can move independently without abrupt lighting changes caused by sudden light cluster changes. Compared to approaches that rely on a hierarchical subdivision of geometry, our method has the advantage of a small and bounded ray-scene intersection cost.

3. CONSTANT-TIME RAY-SCENE INTERSECTION

The ray-scene intersection is accelerated by approximating the scene geometry with a 2D array of voxelizations. The voxelizations are computed as shown in Algorithm 1.

Algorithm 1 Computation of 2D array of voxelizations.

```
Input: Scene $S$ modeled with triangles
Output: 2D array $V$ of scene voxelizations
 1: for $\theta$ from 0 to 180 with $k$ degree increment do
 2:     for $\phi$ from 0 to 180 with $k$ degree increment do
 3:         $V[\theta/k][\phi/k] = \emptyset$
 4:         $previousLayer = near\ plane$
 5:         repeat
 6:             $layer = DepthPeel(previousLayer, \theta, \phi)$
 7:             $V[\theta/k][\phi/k]$ += $layer$
 8:             $previousLayer = layer$
 9:         until $layer == \emptyset$
10:     end for
11: end for
12: return $V$
```

The scene S is voxelized for a dense discretization of the 2D space of all directions. The nested for loops (lines 1-2) iterate over all pairs of angles (θ, ϕ) with a k degree increment. Given a pair (θ, ϕ), the voxelization $V[\theta/k][\phi/k]$ is computed incrementally by depth peeling (lines 3-9). Initially, the voxelization is empty (line 3), and the previous layer is the near clipping plane (line 4), as nothing has been peeled away yet. Then the current layer is computed by depth peeling repeatedly until the last layer (line 6).

Depth peeling is computed with a conventional rendering pass using the z-buffer of the previous pass that provides a per-pixel near clipping value. The layer is added to the voxelization (line 7), and the previous layer is updated to the current layer to prepare for the next depth peeling pass (line 7). Depth peeling is performed with an orthographic projection three times (not shown in Algorithm 1 for simplicity), once for each of the x, y, and z directions, after rotation

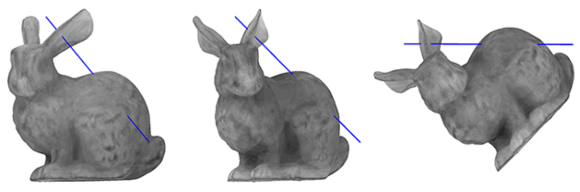
Fig. 3: Voxelization selection for ray intersection. The world (left) is rotated with $\theta = 30^o$ (middle), and then $\phi = 42^o$ (right) to find the voxelization whose rows are aligned with the ray (blue segment).

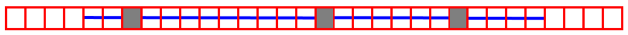
Fig. 4: Intersection of ray (Figure3) with voxelization row.

by θ and ϕ. This triple projection makes sampling more robust to surface orientation. For example, depth peeling only along the z direction would miss surfaces that are parallel to the z direction.

In practice we use a $k = 2^o$ increment, which yields 90×90 voxelizations. For a resolution of $128 \times 128 \times 128$, one voxelization row has 128 bits or 4 32-bit integers, one voxelization takes 256KB, and the 2D array of voxelizations takes slightly less than 2GB.

The intersection between the ray r and the 2D array of voxelizations V is computed as shown in Algorithm 2. r is intersected with the voxelization whose row direction most closely approximates the direction of r. The (θ, ϕ) angles that define the direction of r are computed with the same k angle increment that was used when computing the 2D array of voxelizations (line 1). In Figure 3, the ray direction is most closely approximated by θ = 30 and ϕ = 42, therefore the ray-scene intersection is approximated using the voxelization $V[30/2][42/2]$.

Algorithm 2 ConstantTimeRaySceneIntersection(V, k, r)

```
Input: 2D array of scene voxelizations $V$, direction discretization
       increment $k$, ray $r$
Output: $boolean$ that indicates whether $r$ intersects $V$ or not
 1: $(\theta, \phi)$ = DiscretizeDirection$(r.direction, k, k)$
 2: $row = V[\theta/k][\phi/k]$.LookupRow$(r.midpoint)$
 3: $(i, j) = V[\theta/k][\phi/k]$.ComputeExtent$(r.endpoints)$
 4: $clippedRow = row \ll i$
 5: $clippedRow = clippedRow \gg (i + rowLength - j)$
 6: return $clippedRow \neq 0$
```

The intersection is computed in the row of $V[\theta/k, \phi/k]$ that contains the midpoint of the ray segment (line 2). The row variable contains one bit for every row voxel. The voxel bit is 1 if the voxel contains geometry and 0 otherwise. The ray endpoints are projected onto the row to define the subset of row voxels (i, j) that is traversed by the ray (line 3). The row voxel data is then clipped to the extent of the ray with left and right bit shift operations (lines 4-5). The ray intersects the scene iff the clipped row data contains a non-zero bit (line 6). In Figure 4 the voxelization row has 32 bits. The bits of the row containing the ray (blue segment) are 0000 0010 0000 0000 1000 0001 0000 0000, corresponding to two geometry spans of 1 and 8 voxels for the ear and the body of the bunny. The ray extends from i = 4 to j = 27, so the clipped row data is 0010 0000 0000 1000 0001 0000, which is not zero, and therefore the ray intersects the scene (i.e. at the ear and body of the bunny).

4. INTERACTIVE RENDERING WITH THOUSANDS OF DYNAMIC LIGHTS

The constant-time scene-ray intersection enables rendering scenes with thousands of dynamic lights at interactive rates, according to Algorithm 3.

Algorithm 3 Lighting using 2D array of scene voxelizations.

```
Input: scene $S$, set of $n$ light points $L$, output image camera $C$,
    2D array of scene voxelizations $V$, discretization increment $k$.
Output: $S$ rendered from $C$ lighted with $L$.
 1: $I$ = Render $S$ from $C$ without lighting
 2: for every pixel $p$ in $I$ do
 3:     $P = Unproject(p, C)$
 4:     $hiddenLightsN = 0$
 5:     for every light $L_i$ in $L$ do
 6:         $ray = (P, Li)$
 7:         $hiddenLightsN$ +=
 8:             ConstantTimeRaySceneIntersection$(V, k, ray)$
 9:     end for
10:     $p.outputColor$ = Shade$(hiddenLightsN, n)$
11: end for
```

The algorithm first renders the output image I without any lighting (line 1). Then each pixel p is lit by estimating the visibility of each light L_i from the surface point P acquired at p (line 2-9). The 3D point P is computed by unprojection (line 3). The number of lights hidden from P is initialized to 0 (line 4) and then incremented for every light L_i for which the ray (P, Li) intersects the scene (lines 5-9).

5. 2D VOXELIZATION FOR DYNAMIC SCENES

For scenes where geometry is static, a precomputed 2D array of voxelizations supports a large number of dynamic lights. However, when geometry changes, recomputing each voxelization of the 2D arrat using Algorithm 1 is too slow for interactive rendering. We support dynamic scenes in one of two ways.

5.1 Scenes with dynamic rigid objects

Consider a scene with several types of objects, with each type replicated to several instances, and with each instance moving rigidly through the scene. We support such dynamic scenes as shown in Algorithm 4.

The ray is first intersected with 2D array of voxelizations of the static part of the scene (lines 1-3). If no intersection is found, for each instance of a dynamic object, the ray is transformed to the local coordinate system of the instance, and the transformed ray is intersected with the 2D array of voxelizations of that object type (lines 4-10). For example, for the *Planes* scene in Figure 2 left, we precompute two 2D array of voxelizations: one for the buildings without the planes V_B, and one for the plane V_P; then the intersection between a ray r and the scene is computed by intersecting r once with V_B and twice with V_P, in the local coordinate systems of each of the two planes.

5.2 Scenes with deforming objects

For scenes with many moving objects or with objects that deform, we recompute the scenes 2D array of voxelizations for every frame as shown in Algorithm 5.

Algorithm 4 Intersection of a ray with a scene with dynamic rigid objects.

```
Input: 2D array $V_0$ of voxelizations of the static part of the scene,
    2D array $V_i$ of voxelizations for each type of dynamic object
    $DOT_i$, current coordinate system $CS_j$ of each dynamic object
    instance $DO_j$, ray $r$.
Output: $boolean$ that is true iff the ray intersects the scene.
 1: if ConstantTimeRaySceneIntersection$(V_0, k, r_j)$ then
 2:     return $true$
 3: end if
 4: for all dynamic object instances $DO_j$ do
 5:     $r_j$ = Transform$(r, CS_j)$
 6:     $i = DO_j.objectType$
 7:     if ConstantTimeRaySceneIntersection$(V_i, k_i, r_j)$ then
 8:         return $true$
 9:     end if
10: end for
11: return $false$
```

Algorithm 5 Recompute the scenes 2D array of voxelizations for every frame

```
Input: dynamic scene $S$ updated for current frame, direction dis-
    cretization increment $k$
Output: 2D array of scene voxelizations $V$ for the current frame
 1: Compute $V[0][0]$ using depth peeling as shown in Algorithm 1.
 2: for $\theta$ from 0 to 180 with $k$ degree increment do
 3:     for $\phi$ from 0 to 180 with $k$ degree increment do
 4:         if $(\theta, \phi) == (0, 0)$ then
 5:             continue
 6:         end if
 7:         $V[\theta/k][\phi/k]$ = Rotate$(V[0][0], \theta, \phi)$
 8:     end for
 9: end for
```

Instead of recomputing each voxelization from scratch, we only use depth peeling to compute an initial voxelization $V[0][0]$ (line 1). $V[0][0]$ is computed as shown in Algorithm 1 (lines 3-10). This initial voxelization is then rotated to approximate the other voxelizations (lines 2-9). The rotation traverses $V[0][0]$, and, for each occupied voxel $[i][j][t]$, it sets the corresponding voxel $[i'][j'][t']$ in the rotated voxelization $V[\theta/k][\phi/k]$. The correspondence is computed by rotating voxel $[i][j][t]$ about the center of $V[0][0]$ by (θ, ϕ). No trigonometric function is evaluated since the rotation matrices are precomputed for all (θ, ϕ) pairs and stored in a lookup table. Computing the voxelizations by rotating the initial voxelization is less accurate but much faster than computing each voxelization by depth peeling from the original scene geometry.

6. RESULTS AND DISCUSSION

We have tested our approach on several scenes: *City* (871ktris, Figure 1 left), *Trees* (626ktris, middle), *Garden* (416ktris, right), *Planes* (982ktris, Figure 2 left), *Bear* (1,486ktris, middle), and *Park* (499ktris, right). All scenes have 1,024 lights, except for *Park* which has 7,088 lights. For *City*, *Garden*, and *Trees* the geometry is static, and for *Planes*, *Bear*, and *Park* the geometry is dynamic. We also refer the reader to the video accompanying our paper. All the performance figures reported in this paper were measured on a workstation with a 3.5GHz Intel(R) Core(TM) i7-4770 CPU, with 8GB of RAM, and with an NVIDIA GeForce GTX 780 graphics card.

6.1 Quality

We measure the quality produced by our rendering technique using two error metrics. The first one, ϵ_v, is defined as the relative number of light rays at a pixel for which visibility is evaluated incorrectly. For example, if there are 1,000 lights, and if at a pixel p 10 lights were incorrectly labeled as visible from p, and 5 lights were incorrectly labeled as invisible from p, $\epsilon_v = (10+5)/1,000 = 1.5\%$. The second one, ϵ_s, is defined as the relative shadow value error at a pixel. For the example used above, $\epsilon_s = (10-5)/1,000 = 0.5\%$. ϵ_v is a stricter error measure since for ϵ_s errors can cancel each other out. ϵ_s is a better indication of the pixel intensity errors observed in the final image. The correct visibility and shadow values at each pixel are computed by ray tracing (we use NVIDIA's Optix ray tracer [Nvidia 2016]). Furthermore, we compare our technique to *Imperfect Shadow Maps (ISM)* [Ritschel et al. 008b], a state of the art method for interactive rendering with a large number of lights.

Table I. : Average pixel visibility and pixel shadow value errors for our method and for the prior art Imperfect Shadow Maps method.

Scene		*City*	*Trees*	*Garden*	*Planes*	*Bear*	*Park*
$\epsilon_v\,[\%]$	Ours	1.2	4.0	5.7	1.8	3.3	21.0
	ISM	5.9	7.5	16.0	5.7	6.1	27.0
$\epsilon_s\,[\%]$	Ours	0.5	2.1	2.2	0.8	2.1	7.5
	ISM	3.2	4.7	8.8	4.1	4.9	12.1

Table I shows the average pixel visibility error ϵ_v and the average pixel shadow value error ϵ_s for our scenes, for both our method and for ISM. The number of samples used in ISM was chosen to obtain the same frame rate as our method. In other words, Table I provides an equal-performance quality comparison between our method and ISM. As can be seen, the approximation errors produced by our method are consistently small, and they are consistently smaller than those produced by ISM. Figure 5 shows the six images from Figure 1 and Figure 2 rendered with our method, with ray tracing, and with ISM. The approximation errors produced by ISM are salient: the shadow of the buildings is incorrect and the space in between the buildings is too bright (row 1), the tree canopies are too bright (row 2), the flower bed and column shadows are missing (row 3), the shadow of the low plane is missing (row 4), the bear shadow is poorly defined and it does not convey the contact with the ground (row 5), and the train shadow is poorly defined (row 6). Figure 6 visualizes the approximation errors from Table I, highlighting the much smaller errors of our method compared to ISM.

Table II. : Errors as a function of the number of lights.

Lights		*512*	*1,024*	*2,048*	*4,096*	*10,000*
City	$\epsilon_v\,[\%]$	1.3	1.2	1.2	1.2	1.2
	$\epsilon_s\,[\%]$	0.5	0.5	0.5	0.5	0.4
Garden	$\epsilon_v\,[\%]$	5.5	5.7	5.7	5.7	5.7
	$\epsilon_s\,[\%]$	2.3	2.2	2.2	2.2	2.1

Table II shows the approximation errors of our method as a function of the number of lights. The errors vary little with the number of lights, which is expected since the errors are relative measures, normalized by the number of lights. For all the experiments described so far, we used a $128 \times 128 \times 128$ voxelization to approximate ray-scene intersections, resolution that is sufficient for small errors. Table III shows the approximation errors of our method for lower voxelization resolutions. The errors grow, but good results are also obtained for a coarser $64 \times 64 \times 64$ resolution.

Table III. : Errors as a function of voxelization resolution.

Voxelization res.		$32 \times 32 \times 32$	$64 \times 64 \times 64$	$128 \times 128 \times 128$
City	$\epsilon_v\,[\%]$	4.7	2.2	1.2
	$\epsilon_s\,[\%]$	2.9	1.1	0.5
Garden	$\epsilon_v\,[\%]$	19.7	8.5	5.7
	$\epsilon_s\,[\%]$	16.0	5.1	2.2

For all the experiments described so far, we used a 90×90 2D array of voxelizations, which corresponds to 2 degree rotation angle increments. Table IV shows the approximation errors of our method for smaller voxelization arrays, i.e. for larger rotation angle increments. Using 60×60 voxelizations, i.e. a rotation angle increment of 3 degrees, produces similar quality while memory usage is reduced by a factor of 2.25.

Table IV. : Errors as a function of the voxelization array resolution.

Voxelization rotation res.		30×30	60×60	90×90
City	$\epsilon_v\,[\%]$	2.5	1.5	1.2
	$\epsilon_s\,[\%]$	0.9	0.6	0.5
Garden	$\epsilon_v\,[\%]$	9.0	6.4	5.7
	$\epsilon_s\,[\%]$	3.2	2.3	2.2

6.2 Speed

We compared the rendering speed of our method to ray tracing and to ISM. For ray tracing we used NVIDIA's Optix with BVH (bounding volume hierarchy) scene partitioning for acceleration, which yields the fastest Optix rendering times. Table V shows the frame rendering times for our method and the speedup versus ray tracing. Our method is subsantially faster than ray tracing. The *Planes* scene is rendered using Algorithm 4, which implies two voxelization sets, one for the buildings and one for the airplane, and three intersection lookups per ray, one for the buildings and one for each of the two moving instances of the airplane. The smallest speedup of 5.6 is obtained for the Bear scene where the non-rigidly deforming bear model requires computing the voxelizations on the fly using Algorithm 5.

Table V. : Rendering times of our method and speedup versus ray tracing.

Scene	*City*	*Trees*	*Garden*	*Planes*	*Bear*	*Park*
Ours [ms]	43	46	64	93	586	839
Speedup vs. RT	38.2×	76.4×	37.7×	21.5×	5.6×	33.9×

We have attempted to perform an equal quality comparison to ISM. However, even when substantially increasing the number of geometry sample points used by ISM, the quality plateaus, and it does not reach the quality generated by our method, as shown in the graph in Figure 7. Furthermore, once the number of samples increases above what can be handled in a single rendering pass, the additional rendering pass makes ISM slower than ray tracing. ISM defines samples relative to scene triangles, therefore the samples do not have to be recomputed for dynamic scenes, as the updated vertices of a deforming model implicitly define the updated sample location. This gives ISM a performance advantage for scenes with

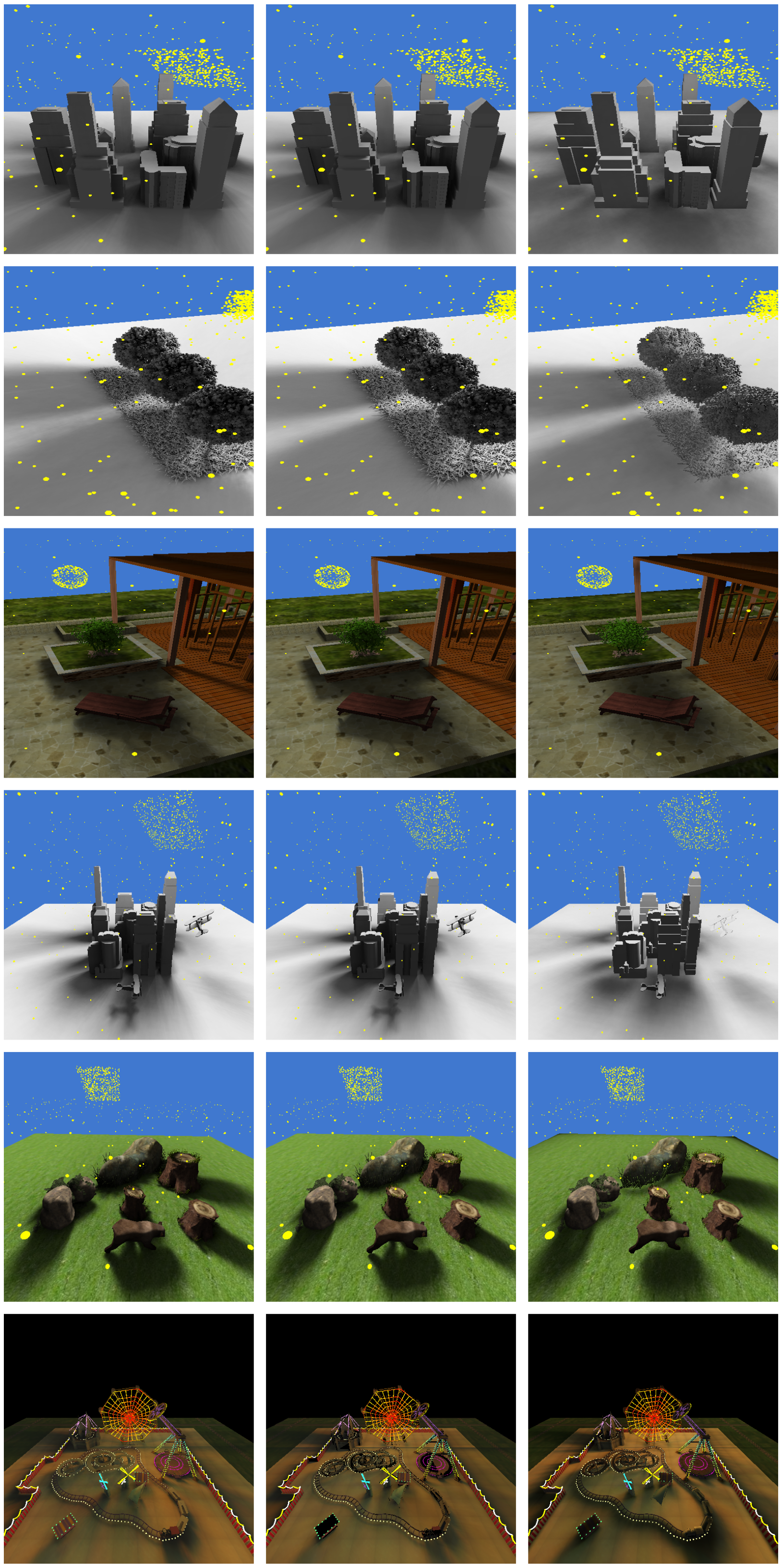

Fig. 5: Comparison between our method (left), ray tracing (middle), and imperfect shadow maps (right).

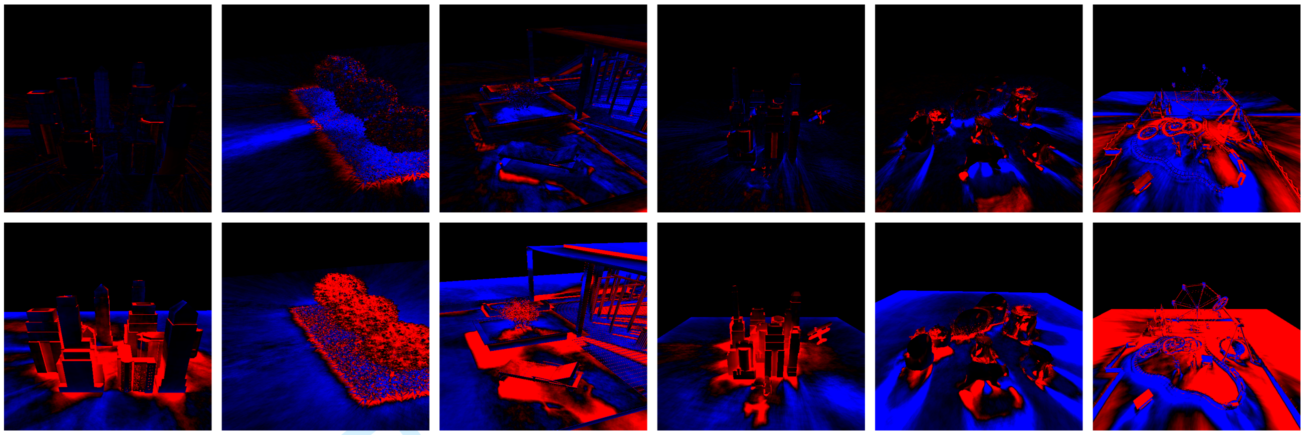

Fig. 6: Visualization of approximation errors ϵ_s for our method (top) and ISM (bottom), for the same frame rate, as reported in Table I. The images correspond to Figures 1 and 2. Darker and brighter regions are highlighted with blue and red. The error is scaled by a factor of 10 for illustration purposes.

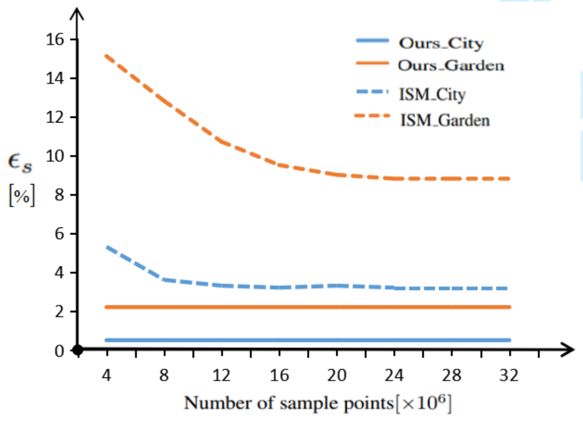

Fig. 7: Pixel shadow value errors for ISM as a function of the number of samples used, and for our method, for comparison. The ISM errors stop decreasing as the number of samples increases, and they do not reach the error values of our method.

deforming geometry like the *Bear*, where ISM is five times faster than our method, which comes however at the cost of a shadow error ϵ_s that is twice as large (i.e. 5.4 for ISM vs. 2.1 for our method). In conclusion, compared to ISM, our method has the advantage of better quality for equal performance, as shown in Table I, and also of providing quality levels that cannot be matched by ISM, whereas ISM has a speed advantage for non-rigidly deforming scenes.

Table VI shows the frame rendering times for our method as a function of the number of lights. As expected, for static scenes (i.e. *City*, *Garden*, and *Trees*) and for the scene with rigidly moving objects (i.e. *Planes*), the frame times double as the number of lights doubles, since almost all of the frame time goes to looking up light ray-voxelization intersections. For the *Bear* the voxelization computation time dominates, so supporting a larger number of lights comes at a relatively smaller additional cost.

Table VI. : Rendering times [ms] as a function of the number of lights.

Lights	*512*	*1,024*	*2,048*	*4,096*	*10,000*
City	22	42	84	171	416
Garden	33	64	130	262	639
Trees	23	46	93	181	447
Planes	46	93	185	370	895
Bear	547	586	648	787	1,185

6.3 Limitations

Our method relies on several approximations. First, the scene geometry is approximated by voxelization. Second, the light ray direction is discretized based on angle increments. These approximation errors are easily controlled and our method will be able to leverage any advances in GPU storage and computing capability. Our method reduces the complexity of the per-ray computation at the cost of storage. The 2D array of voxelizations requires substantial GPU memory resources. Like with any ray tracing acceleration scheme, our method handles dynamic scenes with the additional cost of updating the acceleration data structure for every frame. Unlike hierarchical data structures that do not map well to the GPU, our voxelization is computed with GPU-friendly depth peeling.

7. CONCLUSIONS AND FUTURE WORK

We have presented a method for interactive rendering with thousands of dynamic lights based on a constant-time approximation of the intersection between a ray and the scene geometry. Our method has a significant frame rate advantage over ray tracing, while quality remains very good. Compared to imperfect shadow maps, our method produces significantly more accurate results for the same frame rate. Our method computes visibility for each one of the many lights, and it does *not* cluster the lights. As the lights move from one clustered distribution to another, our method produces smoothly changing shadows, avoiding the temporal artifacts caused by sudden changes in light cluster topology. Visibility is *not* computed by interpolation, as visibility is notoriously discontinuous, but rather by intersecting individual light rays with the scene.

Ray-geometry intersection is a primitive operation in computer graphics and our acceleration scheme could benefit a number of rendering techniques, including ambient occlusion, soft shadows, and specular and diffuse reflections. We make the distinction between the question of whether a ray intersects a scene's geometry, and the question of where the ray-scene intersection occurs. Some applications, including the lighting context explored by this paper, only need to answer the first question, whereas other applications, such as for example specular reflections, also need to answer the second question. The first question is answered by simply testing whether the voxelization row truncated to the extent of the ray is non-zero. The second question requires locating the first non-zero bit in the truncated row, which can be done with a binary search in $\log w$ steps, where w is the voxelization row resolution (e.g. 7 steps for our 128bit voxelization rows).

Our method relies on a scene geometry approximation that not only reduces the complexity of the scene geometry, but that also anticipates all possible directions of the rays with which the scene has to be intersected. The scene geometry approximation scheme is simple and uniform, so its construction, storage, and use map well to the GPU. The scheme reduces the cost of intersecting a ray with a scene to the smallest possible value. With a four channel, 32bit per channel lookup, the intersection with a $128 \times 128 \times 128$ voxelization is essentially obtained with one lookup. The ray-scene intersection is accelerated by "throwing memory at the problem". Our method is already practical in the context of today's GPUs, and it has the potential to become the standard approach for estimating scene-ray intersections in interactive graphics applications, much the same way trivial z-buffering has supplanted complex polygon sorting visibility algorithms.

Our method moves towards making complex dynamic lighting practical in the context of interactive graphics applications. As the number of supported dynamic lights increases, so does the challenge of lighting design and animation. An important direction of future work will have to devise algorithmic approaches for assisting digital content creators with the complex task of defining, calibrating, and animating tens of thousands of lights.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China through Projects 61272349, 61190121 and 61190125,by the National High Technology Research and Development Program of China through 863 Program No.2013AA01A604.

REFERENCES

AKERLUND, O., UNGER, M., AND WANG, R. 2007. Precomputed visibility cuts for interactive relighting with dynamic brdfs. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*. IEEE, 161–170.

CHESLACK-POSTAVA, E., WANG, R., AKERLUND, O., AND PELLACINI, F. 2008. Fast, realistic lighting and material design using nonlinear cut approximation. In *ACM Transactions on Graphics (TOG)*. Vol. 27. ACM, 128.

DACHSBACHER, C., KŘIVÁNEK, J., HAŠAN, M., ARBREE, A., WALTER, B., AND NOVÁK, J. 2014. Scalable realistic rendering with many-light methods. In *Computer Graphics Forum*. Vol. 33. Wiley Online Library, 88–104.

DAVIDOVIČ, T., KŘIVÁNEK, J., HAŠAN, M., SLUSALLEK, P., AND BALA, K. 2010. Combining global and local virtual lights for detailed glossy illumination. In *ACM Transactions on Graphics (TOG)*. Vol. 29. ACM, 143.

DONG, Z., GROSCH, T., RITSCHEL, T., KAUTZ, J., AND SEIDEL, H.-P. 2009. Real-time indirect illumination with clustered visibility. In *VMV*. 187–196.

DONG, Z., KAUTZ, J., THEOBALT, C., AND SEIDEL, H. P. 2007. Interactive global illumination using implicit visibility. In *Conference on Computer Graphics & Applications*. 77–86.

HAŠAN, M., PELLACINI, F., AND BALA, K. 2007. Matrix row-column sampling for the many-light problem. In *ACM Transactions on Graphics (TOG)*. Vol. 26. ACM, 26.

HOLLANDER, M., RITSCHEL, T., EISEMANN, E., AND BOUBEKEUR, T. 2011. Manylods: Parallel many-view level-of-detail selection for real-time global illumination. In *Computer Graphics Forum*. Vol. 30. Wiley Online Library, 1233–1240.

HUO, Y., WANG, R., JIN, S., LIU, X., AND BAO, H. 2015. A matrix sampling-and-recovery approach for many-lights rendering. *ACM Transactions on Graphics (TOG) 34,* 6, 210.

KRISTENSEN, A. W., AKENINE-MÖLLER, T., AND JENSEN, H. W. 2005. Precomputed local radiance transfer for real-time lighting design. In *ACM Transactions on Graphics (TOG)*. Vol. 24. ACM, 1208–1215.

NICHOLS, G., PENMATSA, R., AND WYMAN, C. 2010. Interactive, multiresolution image-space rendering for dynamic area lighting. In *Computer Graphics Forum*. Vol. 29. Wiley Online Library, 1279–1288.

NVIDIA. 2016. Nvidia optix ray tracing engine. *http://developer.nvidia.com/optix*.

OLSSON, O., SINTORN, E., KÄMPE, V., BILLETER, M., AND ASSARSSON, U. 2014. Efficient virtual shadow maps for many lights. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 87–96.

PAQUETTE, E., POULIN, P., AND DRETTAKIS, G. 1998. A light hierarchy for fast rendering of scenes with many lights. In *Computer Graphics Forum*. Vol. 17. Wiley Online Library, 63–74.

RITSCHEL, T., ENGELHARDT, T., GROSCH, T., SEIDEL, H.-P., KAUTZ, J., AND DACHSBACHER, C. 2009. Micro-rendering for scalable, parallel final gathering. *ACM Transactions on Graphics (TOG) 28,* 5, 132.

RITSCHEL, T., GROSCH, T., KAUTZ, J., AND MÜELLER, S. 2007. Interactive illumination with coherent shadow maps. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*. Eurographics Association, 61–72.

RITSCHEL, T., GROSCH, T., KAUTZ, J., AND SEIDEL, H.-P. 2008a. Interactive global illumination based on coherent surface shadow maps. In *Proceedings of Graphics Interface 2008*. Canadian Information Processing Society, 185–192.

RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., AND KAUTZ, J. 2008b. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics (TOG) 27,* 5, 129.

WALTER, B., FERNANDEZ, S., ARBREE, A., BALA, K., DONIKIAN, M., AND GREENBERG, D. P. 2005. Lightcuts: A scalable approach to illumination. *Acm Transactions on Graphics 24,* 3, pgs. 1098–1107.

WALTER, B., KHUNGURN, P., AND BALA, K. 2012. Bidirectional lightcuts. *ACM Transactions on Graphics (TOG) 31,* 4, 59.

WANG, R., HUO, Y., YUAN, Y., ZHOU, K., HUA, W., AND BAO, H. 2013. Gpu-based out-of-core many-lights rendering. *ACM Transactions on Graphics (TOG) 32,* 6, 210.

WU, Y.-T. AND CHUANG, Y.-Y. 2013. Visibilitycluster: Average directional visibility for many-light rendering. *Visualization and Computer Graphics, IEEE Transactions on 19,* 9, 1566–1578.