Fast Ray-Scene Intersection for Interactive Shadow Rendering with Thousands of Dynamic Lights

Lili Wang, Xinglun Liang, Chunlei Meng, and Voicu Popescu

Abstract—We present a method for the fast computation of the intersection between a ray and the geometry of a scene. The scene geometry is simplified with a 2D array of voxelizations computed from different directions, sampling the space of all possible directions. The 2D array of voxelizations is compressed using a vector quantization approach. 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, where we compare our method to ray tracing, to conventional shadow mapping, and to imperfect shadow maps.

Index Terms—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. Lighting is computationally expensive because it implies solving a visibility problem for every point light source. Providing support for a large number of light sources is an important way of improving the quality of imagery rendered at interactive rates.

In this paper we propose a method for interactive rendering with thousands of dynamic lights. Our method is based on an acceleration scheme that enables the fast computation of the intersection between a light ray and the scene geometry. The scene geometry is voxelized from all possible directions, which results in a 2D array of voxelizations. Given a ray, the scene intersection is approximated using the voxelization whose rows are most closely parallel to the ray. The row traversed by the ray is looked up and the intersection is computed with bit-shift operations. To save memory, the 2D array of voxelizations is compressed using a vector quantization approach that detects and leverages the similarity between voxelization rows.

The fast ray-scene intersection enables rendering with thousands of dynamic lights at interactive rates (Fig. 1). Our method brings a substantial speedup over ray tracing at the cost of a small quality trade-off. A 2D array of voxelizations contains 90×90 voxelizations (for a two degree direction discretization), and each voxelization has a $128 \times 128 \times 128$ resolution. This requires 333 MB of storage after vector quantization compression, a substantial but practical amount of memory.

Our approximation is independent of the light sources, which can change from frame to frame at no additional cost. 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 Fig. 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 Fig. 2 middle), or for complex dynamic scenes (e.g., the amusement park shown in Fig. 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.

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

- *L. Wang, X. Liang, and C. Meng are with the State Key Laboratory of Virtual Reality Technology and Systems, Beihang University, Beijing 100191, China. E-mail: lilywang@buaa.edu.cn, {771911275, 1240957820}@qq.com.*
- *V. Popescu is with the Purdue University, West Lafayette, IN 47907. E-mail: popescu@purdue.edu.*

Manuscript received 6 Dec. 2017; revised 19 Mar. 2018; accepted 12 Apr. 2018. Date of publication 19 Apr. 2018; date of current version 1 May 2019. (Corresponding author: Lili Wang.)
Recommended for acceptance by X. Tong.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TVCG.2018.2828422

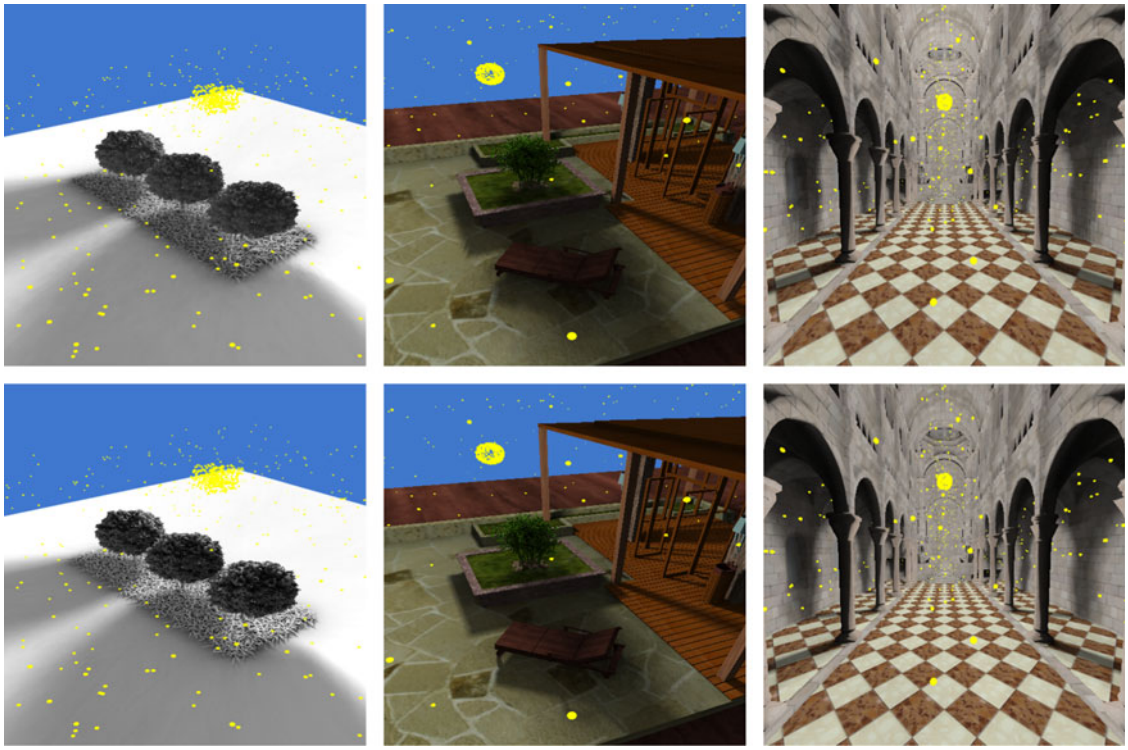


Fig. 1. Scenes with 1,024 point light sources rendered with our method (top), and with ray tracing (bottom). The resolution of voxelization are $128 \times 128 \times 128$ for *Trees*, *Garden*, and $256 \times 256 \times 256$ for *Cathedral*. Our average pixel shadow value errors are 1.8, 2.7, and 3.0 percent, respectively. Our frame rates are 26, 23, and 13 fps, which corresponds to speedups of $43\times$, $15\times$, and $16\times$ versus ray tracing (i.e., NVIDIA's Optix with BVH acceleration).

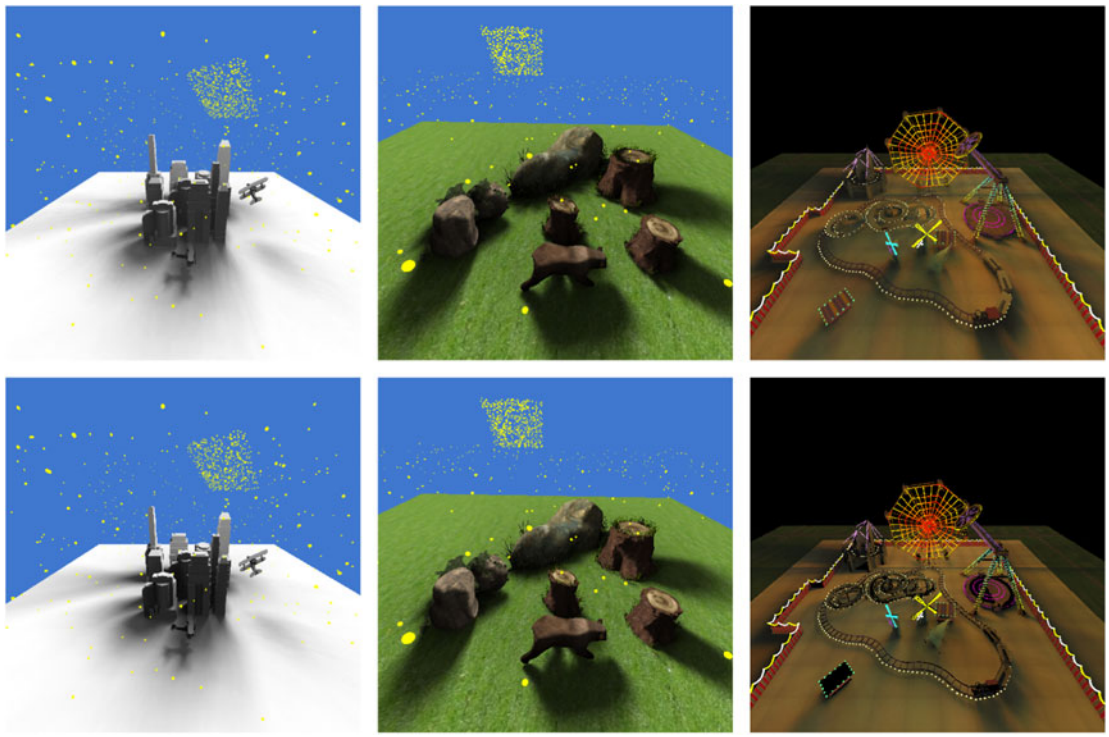


Fig. 2. Dynamic geometry scenes with 1,024, 1,024, and 7,088 lights, rendered with our method (top), and with ray tracing (bottom). Our average pixel shadow value errors are 1.7, 2.1, and 7.5 percent. Our frame rates are 15, 3, and 2 fps, which corresponds to speedups of $12\times$, $6\times$, and $35\times$ versus ray tracing.

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 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 [1].

2.1 Shadow Map Methods

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

Coherent Shadow Maps [2] 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. Since the method relies on single layer shadow maps, the method has to distinguish between "infinitely" far away light sources, medium distance ("semi-local") light sources, and nearby ("local") light sources, in order to handle visibility queries within the scene. A local light source is handled with its own cube map. The method was extended to Coherent Surface Shadow Maps (CSSMs) [3] to support light sources on scene geometry as needed for indirect lighting. A virtual area light [4] 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) [5] 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 coarse point-based approximation of scene geometry by splatting followed by pull-push reconstruction. ISM is a frequently used method for interactive rendering with many lights, so we compare our method to in detail in the Results Section. Hierarchical scene geometry approximations have been used to accelerate shadow map computation. For example, Implicit Visibility [6] uses a disk-based quadtree surface approximation, and ManyLoDs [7] uses a cut through a bounding volume hierarchy of the scene geometry. Virtual shadow maps [8] 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) [9] uses the matrix of all possible output sample/light point 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 light clustering method reduces the number of representative shadow maps [10]. The MRCS algorithm is mapped to an out-of-core GPU implementation [11], and its efficiency is improved by reducing the number of visibility estimates between representative light clusters and output image samples [12].

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 fast 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 Methods

Several techniques accelerate ray tracing visibility computation using scene geometry approximation. Micro-rendering [13] approximates geometry with a point hierarchy which accelerates ray traversal and geometry updates for dynamic scenes. Ray tracing was also accelerated using by geometry voxelization [14]. Voxel octree approaches (e.g., [15]) accelerate ray tracing by building a resolution hierarchy of sparse voxel octrees, and by using the coarser level of the hierarchy as the ray cone diverges. Our method also relies on geometry voxelization, and we reduce the cost of ray-scene intersection to a couple of texure lookups 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 [16]. 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 [17]. Lightcut [18] 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 [19] and in Nonlinear Cut Approximation [20], which are suitable for static scenes, and then in Bidirectional Lightcuts [21], which can also handle dynamic scenes. Instead of clustering lights, a different approach clusters individual rays based on direction, with a cluster containing rays from multiple lights, and solving visibility for each cluster with its own shadow map ([22], [23]), with the advantage of a smaller number of clusters, and the disadvantage of having to define and partition the entire set of rays before shadows can actually be computed.

Another approach for reducing the number of light sources that have to be considered for each output image pixel is to tile the scene into regions affected only by a subset of the scene lights [24]. The artifacts that can result from these sharply defined light regions of influence can be reduced by randomizing the cut-off distance [25]. This approach is simply a method for reducing the number of light rays to be considered, and they are complementary to methods for actually computing shadows, such as ours. In all our examples, we work in the challenging case of lights with infinite range, such that any light source can affect any output image pixel.

Some ray tracing based methods approximate both the lights and the scene geometry. VisibilityClusters [26] 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.

2.3 Fast Voxelization Methods

Our method relies on voxelizing scene geometry efficiently. Geometry voxelization is an infrastructure problem that has received considerable attention.

The insight behind fast voxelization with the help of graphics hardware is that in conventional rendering triangle fragments are assigned to voxels as the frame is rendered, but this information is discarded by z-buffering, since conventional graphics applications typically only care about the first surface encountered at each pixel. Fast voxelization methods modify the conventional graphics pipeline to store, and not discard, the fragment to voxel assignment. One approach achieves real-time voxelization by rendering the scene geometry over a virtual framebuffer that concatenates the 2D slabs of the 3D voxelization [27]. Another method uses deep pixels, with a pixel corresponding to an entire row of the voxelization, with one bit per voxel [28]. The method was subsequently extended to handle solid models by producing a voxelization with "1" or "occupied" bits inside the object [29].

Several fast voxelization techniques depart from the conventional graphics pipeline and define novel voxelization pipelines using general GPU programming APIs, such as CUDA. For example, VoxelPipe [30] computes the 3D voxelization directly with a fast, sort-middle, approach, or a conservative, sort-last approach. A recent pipeline achieves a significant performance improvement by optimizing the triangle-voxel intersection test [31].

Several techniques have been developed to compute and compress light visibility information over the entire scene. Compact precomputed voxelized shadows [32] is a technique that partitions the scene with an octree and stores binary shadow information at each voxel leaf. The octree is compressed in a graph by leveraging the common subtrees. Construction and compression are too laborious for online computation, as required by dynamic lights or dynamic geometry. Furthermore the binary voxel light visibility information supports only one light. Another approach for compressing shadow map data is to find planes in the shadow map and to organize them into sparse shadow trees [33], which is an approach that works well for large scenes with a small number of lights.

This paper does not contribute a novel voxelization technique. We use the deep pixels approach [28] because it is fast and because it can be easily integrated in the shader framework used by our application to render shadows. Our acceleration data structure contains thousands of voxelizations, but only the first one is computed directly from the scene geometry, whereas the subsequent voxelizations are computed efficiently by rotating this initial voxelization. Rotating a voxelization remains orders of magnitude faster than the fastest method for computing a voxelization from scratch, i.e., from scene geometry.

2.4 Low-Level Visibility Query Acceleration Schemes

Our method accelerates visibility queries by computing a data structure where the visibility queries can be answered trivially. This general approach is similar in spirit to the epipolar space voxel grid used in voxelized shadow volumes [34] to compute light source visibility simultaneously for all points along an output image ray, as needed in the case of rendering in participating media (e.g., "God rays"). Specular reflection rendering was also accelerated by approximating an object close to a reflector with a depth image, and by looking up the intersection between a reflected ray and a depth image in a simplified rotated depth map that where the ray projects along a row [35].

Another attempt to accelerate ray-scene intersections precomputes depth maps from viewpoints on an object's bounding sphere, and combines the 2D array of depth maps into a volume texture [36]. The method provides the distance to the object surface with a lookup, but the ray has to originate outside the object, so the method is not suitable for inside-looking-out scenes.

VoxLink is proposed to accelerate ray-casting for volumetric data rendering [37]. The method extends per-pixel linked lists to occupied voxel lists, and subdivides the bounding volume of the scene into multiple bricks to support empty-space skipping. VoxLink can be used to render the scenes with transparent objects and shadows interactively. OSPRay implements a fast ray tracing framework for rendering volumes [38]. While both VoxLink and OSPRay cannot render scenes with thousands light sources in real time because the intersection computations are still not fast enough.

Our method does not store distance but rather a discretization of scene geometry with voxel occupancy bits, which does not provide a direct read of the distance to the intersection, but which does allow the ray segment to originate anywhere. The idea of sampling visibility by discretizing scene geometry is also used in global visibility methods. In adaptive global visibility sampling [39], the visibility information captured by a visibility sample is propagated to all the empty cells in between scene geometry.

3 Fast Ray-Scene Intersection

The ray-scene intersection is accelerated by approximating the scene geometry with a 2D array of voxelizations. Section 3.1 describes the construction of the 2D array of voxelizations, Section 3.2 describes the vector quantization compression of the 2D array of voxelizations, and Section 3.3 describes using the 2D array of voxelizations to approximate the intersection of a ray with scene geometry.

3.1 Construction of 2D Array of Voxelizations

The voxelizations are computed as shown in Algorithm 1. 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_{ij}, and the 90 degree rotated voxelization $V_{ij'}$, are computed using a prior art approach for single-pass voxelization [28] (line 4). Single-pass voxelization renders the scene with an orthographic view that matches the voxelization and sets the occupancy bit for the voxel that contains each fragment. We compute two voxelizations at the time, leveraging the fact that the voxelization for $(\theta, \phi + 90)$ is aligned with the voxelization for (θ, ϕ), and it can simply be computed by transposing the indices of the voxel where the occupancy bit is written. Both original and transposed voxelizations have to be stored since we store rows, and since a row cannot be recreated from columns quickly. For each (θ, ϕ) pair, voxelization is performed three times (not shown in Algorithm 1 for simplicity), once for each of the x, y, and z directions. This makes sampling more robust to surface orientation. For example, voxelization only along the z direction would miss surfaces that are parallel to z.

Algorithm 1. Computation of 2D Array of Voxelizations

Input: Scene S modeled with triangles
Output: 2D array V of scene voxelizations
1: **for** θ from 0 to 180 with k degree increment **do**
2: **for** ϕ from 0 to 90 with k degree increment **do**
3: $i = \theta/k$; $j = \phi/k$; $j' = (\phi + 90)/k$;
4: $(V_{ij}, V_{ij'})$ = SPVoxelization(θ, ϕ)
5: **end for**
6: **end for**
7: **return** V

3.2 Compression of the 2D Array of Voxelizations

A 2D array of voxelizations requires significant storage. A typical angular resolution value (i.e., k in Algorithm 1) is 2 degrees, which results in a 90×90 2D array of voxelizations. A typical voxelization resolution is $128 \times 128 \times 128$. Consequently, a typical storage requirement for a 2D array of voxelizations is just below 2 GB. However, many of the voxelization rows are quite similar and we have developed a vector quantization compression method that leverages row similarity to reduce the storage requirement. The vectors are the rows of all the voxelizations. For the typical parameter values given above, there are $90 \times 90 \times 128 \times 128$, or about 126M rows. The compression method proceeds in two steps, as shown in Algorithm 2.

In a first step, a dictionary of n most popular rows is computed (lines 1-2). A histogram H of the rows is computed by sorting the rows and counting the number of occurrences for each row. Then the unique rows are sorted based on the number of occurrences and the n most frequently encountered rows are selected to define the dictionary D. In a second step, the rows in the initial, uncompressed 2D array of voxelizations V are mapped to rows in D (lines 3-15). The algorithm uses a sequence of s compression steps, from less to more aggressive.

The rows of the dictionary (lines 4-5) and the yet to be mapped rows (line 8) are simplified to force a match. At step c, a row is simplified by down-sampling the row with a factor of 2^c. The downsampling of 2^c bits to a single bit sets the output bit to 1 iff any of the input bits has a value of 1, which preserves the blocker sample, albeit at a less accurate location. The simplified row r' is searched in the simplified dictionary D' and mapped to a matching row j, if such a row is found (line 9). We typically use four compression steps (i.e., $s = 4$). The first step ($c = 0$) does not down-sample to map the voxelization rows that were used to build the dictionary. In the last step ($c = 3$), the rows are down-sampled by a factor of 8. The rows that remain unmapped after the last compression step are mapped to a random dictionary row (lines 13-14), which achieves the same shadow error as down-sampling with higher factors, but at a lesser computational cost.

Algorithm 2. Compression of 2D Array of Voxelizations

Input: uncompressed 2D array of scene voxelizations V, number of rows in dictionary n, number of compression steps s
Output: dictionary D and mapping V' of V to D
1: H = Histogram of all rows in V
2: D = n most frequent rows in H
3: **for** compression step c = 0 to $s - 1$ **do**
4: **for** each row i in D **do**
5: $D'[i]$ = Simplify($D[i]$, 2^c)
6: **end for**
7: **for** each unmapped row i **do**
8: r' = Simplify($V[i]$, 2^c)
9: **if** (j = Find(r', D')) **then** $V'[i] = j$
10: **end if**
11: **end for**
12: **end for**
13: **for** each remaining unmapped row i **do**
14: $V'[i]$ = Random(0, n-1)
15: **end for**
16: **return** (D, V')

The compressed 2D array of voxelizations is defined by the dictionary D and the mapping V'. If the original number of rows is N, and if a row has b bits, the compression factor is (Nb) / $(nb + Nlog_2 n) \approx b$ / $log_2 n$. For larger dictionaries the compression is less lossy, but that comes at the cost of a smaller compression factor. As described in the results section, we use 1M row dictionaries which corresponds to a compression factor of about 128/20. This reduces the storage requirement from 2 GB to 333 MB, and the compression losses translate to minimal shadow errors.

3.3 Intersection of a Ray with a 2D Array of Voxelizations

The intersection between the ray r and the compressed 2D array of voxelizations V' is computed as shown in Algorithm 3. 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 Fig. 3, the ray direction is most closely approximated by θ = 30 and ϕ = 42, therefore the ray-scene intersection is approximated using the voxelization $V[30/2][42/2]$.

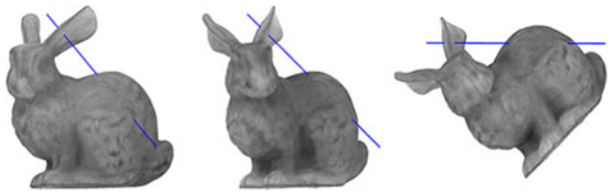


Fig. 3. Voxelization selection for ray intersection. The scene (left) is rotated with $\theta = 30^o$ (middle), and then $\phi = 42^o$ (right) to find the voxelization whose rows are aligned with the ray (blue segment).

The intersection is computed in the row of V'_{ij} that contains the midpoint of the ray segment (line 3), where V'_{ij} is the voxelization with rows most aligned to ray r (lines 1-2). Since the 2D array of voxelizations V' is compressed, the row of V' does not contain actual geometry information, but rather the index in the dictionary D from where to retrieve the actual row.The midpoint is used to find the best row in case the ray traverses more than one row. Due to rotation angle discretization, rays are not perfectly aligned with the voxelization row, and it can happen that the ray traverses several rows. A more conservative solution would partition the ray based on the rows it traverses and would handle each subray in its own row. Our experiments reveal that the quality improvement brought by this approach does not warrant the additional cost (i.e., a 4 percent quality improvement for a 45 percent performance penalty).

Algorithm 3. Intersect$((V', D), k, r)$

Input: compressed 2D array of scene voxelizations (V', D), direction discretization increment k, ray r
Output: $boolean$ that approximates whether r intersects the scene or not
1: (θ, ϕ) = DiscretizeDirection$(r.direction, k, k)$
2: $i = \theta/k$; $j = \phi/k$;
3: $row = D[V'_{ij}$.LookupRow$(r.midpoint)]$
4: $(s, e) = V'_{ij}$.ComputeExtent$(r.endpoints)$
5: $clippedRow = row \ll s$
6: $clippedRow = clippedRow \gg (s + rowLength - e)$
7: **return** $(clippedRow \neq 0)$

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 (s, e) that is traversed by the ray (line 4). The row voxel data is then clipped to the extent of the ray with left and right bit shift operations (lines 5-6). The ray intersects the scene iff the clipped row data contains a non-zero bit (line 7). In Fig. 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 s = 4 to e = 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 fast scene-ray intersection enables rendering scenes with thousands of dynamic lights at interactive rates, according to Algorithm 4.

Fig. 4. Intersection of ray with the aligned voxelization row that contains it (see right image in Fig. 3).

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 2-10). The 3D point P is computed by unprojection (line 3). The number of lights hidden from P is initialized to 0 (lines 4) and then incremented for every light L_i for which the ray (P, L_i) intersects the scene (lines 5-8).

Algorithm 4. Lighting Using 2D Array of Scene Voxelizations

Input: scene S, set of n light points L, output image camera C, compressed 2D array of scene voxelizations (V', D), 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: $shadow = 0$
5: **for** every light L_i in L **do**
6: $ray = (P, L_i)$
7: $shadow$ += Intersect$((V', D), k, ray)$
8: **end for**
9: $p.outputColor$ = Shade$(shadow, n)$
10: **end for**

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 array 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 5.

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 Fig. 2 left, we precompute two 2D array of voxelizations: one for the buildings without the planes V_s, and one for the plane V_0; then the intersection between a ray r and the scene is computed by intersecting r once with V_s and twice with V_0, 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 use one 2D array of voxelizations for the entire scene, which is recomputed for every frame. As mentioned above, Algorithm 1 is too slow for real time performance.

TABLE 1
Average Pixel Visibility and Pixel Shadow Value Errors for our Method and for the Prior Art Imperfect Shadow Maps Method

Scene		*Trees*	*Garden*	*Cathedral*	*Planes*	*Bear*	*Park*
$\epsilon_v[\%]$	Ours	4.8	7.8	6.4	3.4	3.3	21.0
	ISM	8.5	13.1	20.1	6.5	6.1	27.0
$\epsilon_s[\%]$	Ours	1.8	2.7	3.2	1.7	2.1	7.5
	ISM	6.7	5.8	9.8	3.4	4.9	12.1

Instead of using single pass voxelization for every (θ, ϕ) direction, we extend single pass voxelization to voxelize along all directions with a single pass over the scene geometry. The scene triangles are rendered with the orthographic view of the first voxelization $V[0][0]$ (i.e., $(\theta, \phi) = (0, 0)$). Each triangle fragment is processed with Algorithm 6.

Algorithm 5. Intersection of a Ray with a Scene with Dynamic Rigid Objects

Input: compressed 2D array of voxelizations (V'_s, D_s) of the static part of the scene, compressed 2D arrays of voxelizations (V'_i, D_i) for each type of rigid 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 Intersect((V'_s, D_s), 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 Intersect((V'_i, D_i), k_i, r_j) then
8:       return true
9:    end if
10: end for
11: return false
```

Algorithm 6. Fragment Shader Algorithm for Recomputing the Scene's 2D Array of Voxelizations for Every Frame

Input: fragment f of scene triangle rendered with orthographic view of $V[0][0]$, direction discretization increment k
Output: 2D array of scene voxelizations V for the current frame

```
1: p = Unproject(f)
2: for θ from 0 to 180 with k degree increment do
3:    for φ from 0 to 90 with k degree increment do
4:       p' = Rotate(p, θ, φ)
5:       v = Voxelize(p')
6:       V[θ/k][φ/k][v] = 1
7:       V[θ/k][(φ + 90)/k][Transpose(v)] = 1
8:    end for
9: end for
```

The 3D point p corresponding to fragment f is computed by unprojection (line 1). Then p is rotated to each voxelization local coordinate system (line 4). No trigonometric function is evaluated since the rotation matrices are precomputed for all (θ, ϕ) pairs and stored in a lookup table. The voxel v containing the rotated point p' is set to occupied (line 6). Like in Algorithm 1, as we compute voxelization $V[\theta/k][\phi/k]$ we also compute the transposed voxelization $V[\theta/k][(\phi + 90)/k]$, for efficiency (line 7). Computing the rotated voxelizations by rotating the fragments rasterized for the initial voxelization is significantly faster than computing each voxelization by single pass voxelization from the original scene geometry. This significant performance gain only implies a small quality reduction cost, as shown in the Results Section. The compression of the 2D array of voxelizations is too slow to run for every frame, so the 2D array of voxelizations for scenes with deforming objects has to be stored without compression.

6 Results and Discussion

We have tested our approach on several scenes: *Trees* (626ktris, Fig. 1 left), *Garden* (416ktris, middle), *Cathedral* (456ktris, right), *Planes* (982ktris, Fig. 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 *Trees*, *Garden*, and *Cathedral*, 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 3.5 GHz Intel(R) Core(TM) i7-4770 CPU, with 8 GB of RAM, and with an NVIDIA GeForce GTX 1,080 graphics card. We discuss the shadow quality (Section 6.1), the frame rate (Section 6.2), the memory requirements (Section 6.3), the extensions (Section 6.4) and the limitations (Section 6.5) of our method.

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 percentage 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 percentage 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 [40]). We also compare our technique to conventional shadow mapping, and to *Imperfect Shadow Maps (ISM)* [5], a state of the art method for interactive rendering with a large number of lights.

Table 1 shows the average pixel visibility error ϵ_v and the average pixel shadow value error ϵ_s for our scenes, for both our method (with compression) and for ISM. ISM relies on a point-based representation of the scene, which is then rendered by splatting and pull-push hole-filling to create a low resolution shadow map for each light. In this comparison we used approximately 11,000 point samples to render each 128×128 ISM, which yields a frame rate comparable to that of our method. In other words, Table 1 provides an equal-performance quality comparison between our method and ISM. An equal-performance comparison between our method and conventional shadow mapping is not possible. Rendering a shadow map for each light is significantly slower than our method even for low shadow map resolutions that

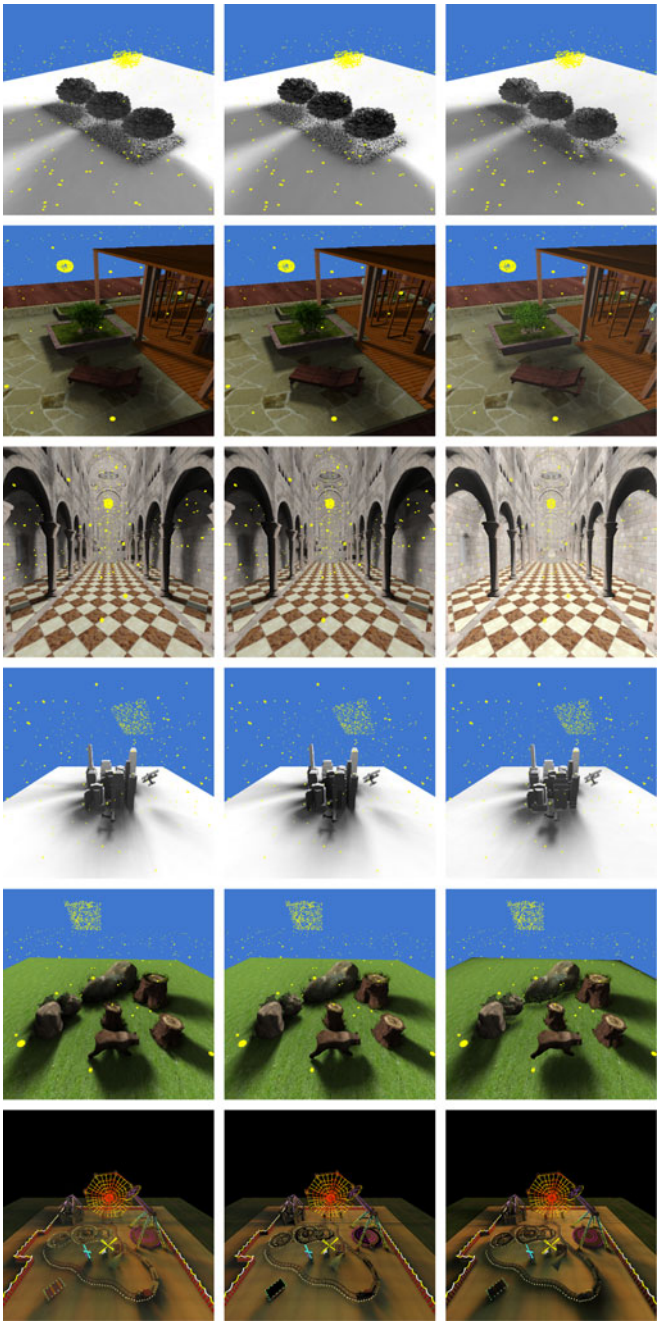


Fig. 5. Comparison between our method (left), ray tracing (middle), and imperfect shadow maps (right).

generate unacceptably large errors. We do provide an *equal-quality* comparison of our method to conventional shadow mapping in Section 6.2.

As can be seen in Table 1, the approximation errors produced by our method are consistently small, and they are consistently smaller than those produced by ISM. Fig. 5 shows the six images from Figs. 1 and 2 rendered with our method, with ray tracing, and with ISM. The approximation errors produced by ISM are salient: the tree canopies are too bright (row 1), the flower bed and column shadows are missing (row 2), the shadows of the pillars 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). Fig. 6 visualizes the approximation errors

TABLE 2
Errors as a Function of the Number of Lights

Lights		*512*	*1,024*	*2,048*	*4,096*	*10,000*
Trees	$\epsilon_v[\%]$	4.6	4.8	4.7	4.6	4.7
	$\epsilon_s[\%]$	1.8	1.8	1.7	1.7	1.7
Garden	$\epsilon_v[\%]$	7.8	7.8	7.9	7.7	7.8
	$\epsilon_s[\%]$	2.8	2.7	2.9	2.7	2.7

TABLE 3
Errors as a Function of Voxelization Resolution

Voxelization resolution		32^3	64^3	128^3	256^3
Trees	$\epsilon_v[\%]$	10.5	6.0	4.8	4.0
	$\epsilon_s[\%]$	7.1	3.1	1.8	1.6
Garden	$\epsilon_v[\%]$	12.3	8.8	7.8	5.6
	$\epsilon_s[\%]$	6.4	3.5	2.7	2.1
Cathedral	$\epsilon_v[\%]$	14.4	10	6.4	5.5
	$\epsilon_s[\%]$	6.7	4.5	3.2	3.0

TABLE 4
Errors as a Function of the Voxelization Rotation Resolution

Voxelization rotation resolution		30×30	60×60	90×90
Trees	$\epsilon_v[\%]$	6.4	5.3	4.8
	$\epsilon_s[\%]$	2.6	2.1	1.8
Garden	$\epsilon_v[\%]$	10.6	8.3	7.8
	$\epsilon_s[\%]$	4.1	2.8	2.7

from Table 1, highlighting the smaller errors of our method compared to ISM.

Table 2 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 3 shows the approximation errors of our method for lower voxelization resolutions. The errors are significantly larger for the lower resolutions, but the $64 \times 64 \times 64$ could be used in applications where memory is at a premium.

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 4 shows the approximation errors of our method for smaller voxelization arrays, i.e., for larger rotation angle increments. Compared to voxelization resolution, shadow quality is less dependent on voxelization rotation resolution. Using 60×60 voxelizations, i.e., a rotation angle increment of 3 degrees, produces a quality similar to using 90×90 voxelizations, while memory usage is reduced by a factor of 2.

6.2 Speed

We have implemented our method using shaders. We compute our 2D array of voxelizations by extending a prior-art

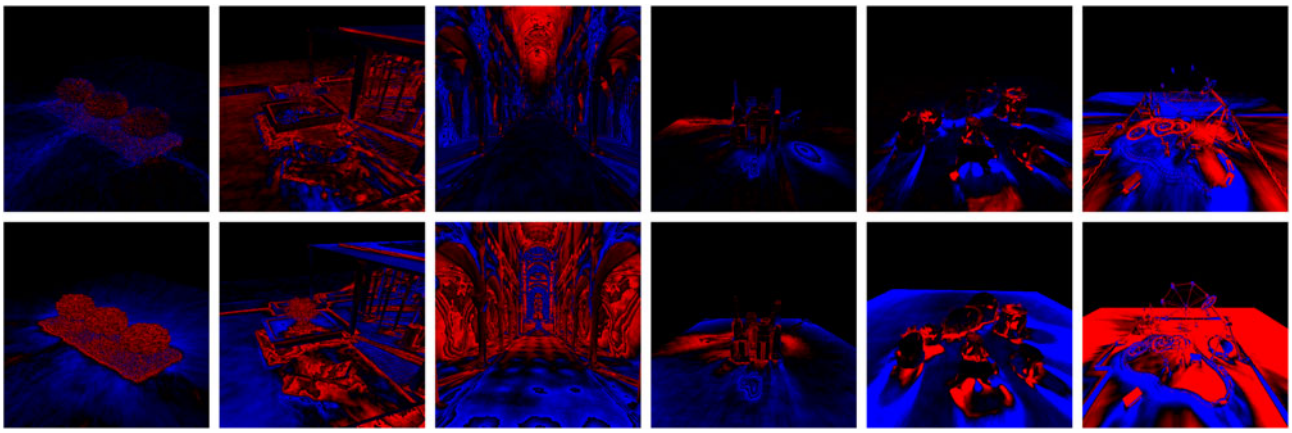


Fig. 6. Visualization of approximation errors ϵ_s for our method (top) and ISM (bottom), for the same frame rate, as reported in Table 1. The images correspond to Figs. 1 and 2. Darker and brighter regions are highlighted with blue and red. The error is scaled by a factor of 10 for illustration purposes.

single-pass voxelization method [28] as shown in Algorithm 6. The 2D array of voxelizations is stored as a 1D array of 3D textures. Then, for each frame, the fragment shader looks up the 2D array of voxelizations for each light to estimate visibility from the output image sample to the light.

Table 5 shows the frame rendering times for our method and the speedup versus ray tracing and versus conventional shadow mapping. For ray tracing we used NVIDIA's Optix (version number 3.9.1) with bounding volume hierarchy (BVH) scene partitioning for acceleration, which yields the fastest Optix rendering times. The Optix times do not include BVH construction for the static scenes, i.e., *Trees*, *Garden*, and *Cathedral*, and they do include it for the dynamic scenes, i.e., *Planes*, *Bear*, and *Park*.

Our method is substantially faster than ray tracing. The *Planes* scene is rendered using Algorithm 5, 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. In the case of a few rigidly moving objects our method has the advantage of not having to recompute its acceleration data structure. The serial off-line precomputation and compression of the voxelizations takes 492s, 470s, 527s, and 726s for the *Trees*, *Garden*, *Cathedral*, and *Planes* scenes by computing each voxelization from scratch as shown in Algorithm 1. The smallest speedup of 6 is obtained for the Bear scene where the non-rigidly deforming bear model requires computing the voxelizations on the fly using Algorithm 6. Computing the voxelizations using Algorithm 6 takes 277 and 210 ms for the *Bear* and the *Park* scene (without compression), or 89 and 48 percent of the total frame rendering time.

Compared to conventional shadow mapping, our method achieves a substantial speedup (e.g., 89× for *Cathedral*). The conventional shadow maps were rendered at the resolutions given in the last row of Table 5, which achieve a similar pixel visibility error ϵ_v to our method.

TABLE 5
Rendering Times of our Method and Speedup versus Ray Tracing (RT) and versus Conventional Shadow Mapping (SM)

Scene	*Trees*	*Garden*	*Cathedral*	*Planes*	*Bear*	*Park*
Ours[ms]	38	43	59	66	312	441
Speedup versus RT	43×	15×	21×	12×	6×	35×
Speedup versus SM	52×	16×	89×	24×	10×	41×
Res. of SM	384	128	36	96	128	64

We have attempted to perform an equal quality comparison to ISM. However, 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 Fig. 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 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., 4.9 for ISM versus 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 1, and also of providing quality levels that cannot be matched by ISM, whereas ISM has a speed advantage for non-rigidly deforming scenes.

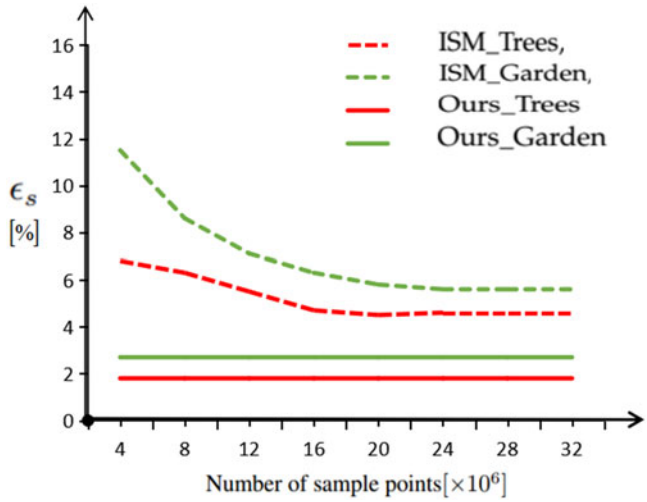


Fig. 7. Pixel shadow value errors for ISM as a function of the number of samples used, and for our method, for comparison. The ISM errors stop decreasing as the number of samples increases, and they do not reach the error values of our method.

TABLE 6
Rendering Times [ms] as a Function of the Number of Lights

Lights	*512*	*1,024*	*2,048*	*4,096*	*10,000*
Trees	20	38	77	159	378
Garden	21	43	89	183	459
Cathedral	29	59	124	273	683
Planes	35	66	135	266	647
Bear	297	312	345	417	630

TABLE 7
Rendering Time of our Method and Speedup versus Diagonal Tracing

Scene	*Trees*	*Garden*	*Cathedral*	*Planes*	*Bear*	*Park*
Ours [ms]	38	43	59	66	312	441
Diagonal tracing [ms]	2858	3401	2673	2825	3806	22482
Speedup	75×	79×	45×	43×	12×	51×

TABLE 8
Memory Reduction by Compression

Voxelization resolution	32^3	64^3	128^3	256^3
Uncompressed[MB]	32	254	2025	16200
compressed[MB]	36	95	333	1298

Table 6 shows the frame rendering times for our method as a function of the number of lights. As expected, for static scenes (i.e., *Trees*, *Garden*, and *Cathedral*) 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.

Our method achieves performance by avoiding marching diagonally through the voxelization when computing the ray intersection. We have compared our performance to marching diagonally one voxel at the time, and the speedup brought by our method is substantial, see Table 7. We have also implemented a method that partitions the ray into subsegments based on the rows of the voxelization it traverses, and that steps one ray subsegment at the time. Stepping one subsegment at the time is made possible by storing in each voxel the distance to the next occupied voxel on the same row. Even compared to this fast diagonal marching approach, our speedups are 22× and 31× for the *Trees* and *Garden* scenes.

6.3 Compression

Compressing the array of voxelizations reduces memory footprint at the cost of a small shadow quality loss, of preprocessing computation, and of a small frame rate decrease.

6.3.1 Memory Reduction

Table 8 shows the memory reduction by compression. The overhead of compression is prohibitive for the $32 \times 32 \times 32$ resolution, but compression reduces memory consumption by a factor of $6\times$ to a manageable 333 MB for the $128 \times 128 \times 128$ resolution.

TABLE 9
Quality Loss Due to Voxelization Compression

Scenes.		*Trees*	*Garden*	*Cathedral*	*Planes*
Without compression	$\epsilon_v[\%]$	4.5	7.6	6.4	3.3
	$\epsilon_s[\%]$	1.7	2.7	3.2	1.6
With compression	$\epsilon_v[\%]$	4.8	7.8	6.4	3.4
	$\epsilon_s[\%]$	1.8	2.7	3.2	1.7

TABLE 10
Pre-Computation Time

Scenes.	*Trees*	*Garden*	*Cathedral*
Voxelization [s]	0.2	0.19	0.19
Compression [s]	492	470	527

TABLE 11
Frame Rate Loss Due to Compression

Scenes.	*Trees*	*Garden*	*Cathedral*
Without compression	28fps	25fps	20fps
With compression	26fps	23fps	17fps

6.3.2 Quality Loss

Table 9 shows the shadow approximation errors introduced by our lossy voxelization compression scheme for the frames shown in Fig. 5. Whereas the visibility errors go up, the quality loss in terms of shadow intensity is small, which indicates that the visibility errors introduced by compression are random and that they cancel out.

6.3.3 Performance

For static scenes, the array of voxelizations is pre-computed off-line. For dynamic scenes with deforming objects, the voxelizations are recomputed for every frame. Row 1 in Table 10 gives the times for computing the arrays of voxelizations on GPU (using Algorithm 1). Except for dynamic scenes with deforming objects, when the voxelizations are used as is, the voxelizations are compressed using Algorithm 2. Row 2 in Table 10 gives the times for compressing the arrays of voxelizations on a single CPU.

Using a compressed array of voxelizations to compute shadows adds a decoding step to the intersection lookup. Table 11 shows that the performance loss due to this additional step is small.

6.4 Extensions

We have extended our approach to handle colored light sources. In Fig. 8 the TV is modeled with 1,024 colored point light sources. The voxelization resolution is $128 \times 128 \times 128$. The compressed memory footprint is 333 MB. The scene is rendered at 16 Hz. We have also extended our method to support indirect illumination, where virtual point light sources are placed on scene surfaces to compute second order light rays. Fig. 9 shows a Cornell box rendered with our approach with 1,024 real and virtual point light sources at 20 fps ($128 \times 128 \times 128$ voxelization resolution, 333 MB memory footprint).

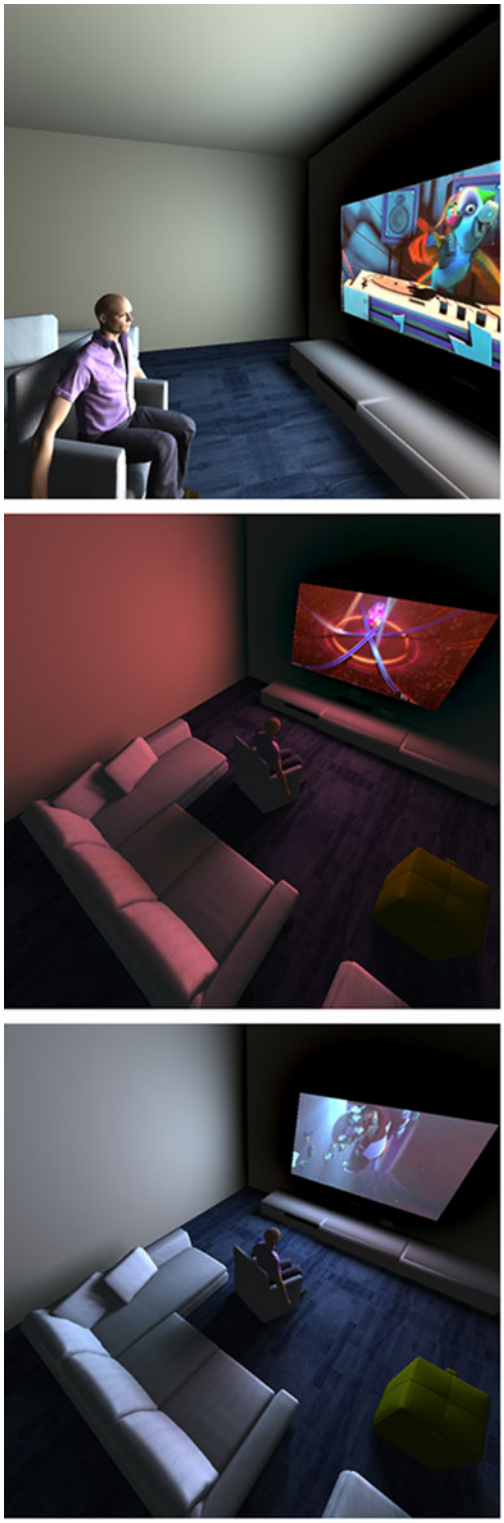


Fig. 8. Scene with TV modeled with 1,024 light sources, rendered with our method at 16 fps.

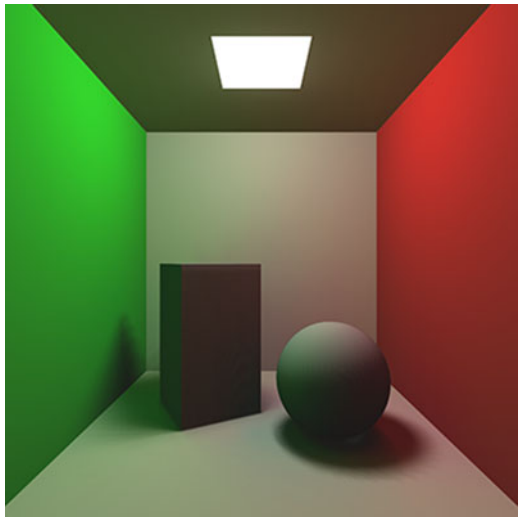


Fig. 9. Cornell box with lighting modeled with with 1,024 real and virtual point light sources, rendered with our method at 20 fps.

6.5 Limitations

Our method reduces the complexity of the per-ray computation at the cost of storage, resorting on several approximations. First, the scene geometry is approximated by voxelization. Second, the light ray direction is discretized based on angle increments. Third, the array of voxelizations are compressed using a lossy vector quantization scheme. These approximation errors are easily controlled and our method will be able to leverage any advances in GPU storage and computing capability.

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. For scenes where the number of dynamic objects is small, and the dynamic objects are rigid, it is feasible to pre-compute a 2D array of voxelizations for each object and to transform the ray to the local coordinate system of the moving object. The spatial resolution of the per-object voxelizations can be smaller than for those used for the entire scene, e.g., we have used $32 \times 32 \times 32$ voxelizations for the airplanes.

We have demonstrated our technique with voxelization resolution of up to $256 \times 256 \times 256$. This resolution might not be sufficient for large scenes with complex geometry, where discretization "banding" like artifacts might remain. Future work will investigate extending cascaded shadow map approaches [41] to our method, by computing a 2D array of voxelizations for the part of the scene located in the near light frustum.

Shadow intensity errors are small because occasional visibility errors cancel out. Like for many methods for rendering soft shadows, the approximation error of our method increases with the hardness of the shadows. This limitation could be addressed in future work by detecting the occurrence of hard shadows and by building conventional shadow maps for the lights that cast them.

Our method requires substantial amount of storage to anticipate all possible rays that have to be intersected with the scene. Good shadows are obtained for a 90×90 discretization of ray directions and a $128 \times 128 \times 128$ scene geometry discretization, which, with vector quantization compression totals a practical 333 MB. Today's GPUs also support $256 \times 256 \times 256$ voxelizations, and the accuracy of the ray-scene intersection approximation will go up as the storage capacity of GPUs continues to improve.

We did not investigate parallelizing the compression of the array of voxelizations since compression is performed

off-line. From Algorithm 2, one can see that the running time of compression is dominated by the search for the simplified row r' in the simplified dictionary D' (line 8). This yields an asymptotic running time of $Nlog_2n$, where N is the total number of rows, and n is the number of dictionary entries. If pre-processing time is important, future work could examine parallelizing compression by compressing each row in parallel. Rows are independent and processing them in parallel does not result in concurrent writes, which promises good speedup scalability with the number of processors.

7 Conclusions and Future Work

We have presented a method for interactive rendering with thousands of dynamic lights based on an 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 acceptable. Compared to imperfect shadow maps, our method produces 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.

Our method handles scenes of medium complexity very quickly and it could be extended to high complexity scenes when used in conjunction with prior art methods. For example, our method is compatible with prior work that relies on hierarchical scene subdivisions. For example, one could use an octree where a leaf is modeled with one of our 2D array of voxelizations. Such a hybrid approach will not guarantee a fixed, small number of texture look-ups per ray-scene intersections, but the powerful leafs will reduce the depth of the hierarchical subdivision, and will aid with balancing it.

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). Once the location of the intersection is found, the location can be mapped to a main, unrotated voxelization that stores all ingredients for shading, i.e., color for first order reflections, or normal for the spawning of the second order reflected ray.

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 two lookups, one to lookup the row index in the dictionary, and one to lookup the ray. 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 61772051, by the National High Technology Research and Development Program of China through 863 Program No.2013AA01A604.

References

[1] C. Dachsbacher, J. Křivánek, M. Hašan, A. Arbree, B. Walter, and J. Novák, "Scalable realistic rendering with many-light methods," *Comput. Graph. Forum*, vol. 33, no. 1, pp. 88–104, 2014.

[2] T. Ritschel, T. Grosch, J. Kautz, and S. Müeller, "Interactive illumination with coherent shadow maps," in *Proc. 18th Eurographics Conf. Rendering Techn.*, 2007, pp. 61–72.

[3] T. Ritschel, T. Grosch, J. Kautz, and H.-P. Seidel, "Interactive global illumination based on coherent surface shadow maps," in *Proc. Graph. Interface*, 2008a, pp. 185–192.

[4] Z. Dong, T. Grosch, T. Ritschel, J. Kautz, and H.-P. Seidel, "Real-time indirect illumination with clustered visibility," in *Proc. Vis. Model. Vis. Workshop*, 2009, pp. 187–196.

[5] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz, "Imperfect shadow maps for efficient computation of indirect illumination," *ACM Trans. Graph.*, vol. 27, no. 5, 2008b, Art. no. 129.

[6] Z. Dong, J. Kautz, C. Theobalt, and H. P. Seidel, "Interactive global illumination using implicit visibility," in *Proc. Conf. Comput. Graph. Appl.*, 2007, pp. 77–86.

[7] M. Hollander, T. Ritschel, E. Eisemann, and T. Boubekeur, "ManyLoDs: Parallel many-view level-of-detail selection for real-time global illumination," *Comput. Graph. Forum*, vol. 30, no. 4, pp. 1233–1240, 2011.

[8] O. Olsson, E. Sintorn, V. Kämpe, M. Billeter, and U. Assarsson, "Efficient virtual shadow maps for many lights," in *Proc. 18th Meet. ACM SIGGRAPH Symp. Interactive 3D Graph. Games*, 2014, pp. 87–96.

[9] M. Hašan, F. Pellacini, and K. Bala, "Matrix row-column sampling for the many-light problem," *ACM Trans. Graph.*, vol. 26, no. 3, 2007, Art. no. 26.

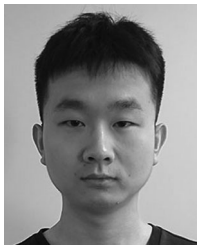
[10] T. Davidovič, J. Křivánek, M. Hašan, P. Slusallek, and K. Bala, "Combining global and local virtual lights for detailed glossy illumination," *ACM Trans. Graph.*, vol. 29, no. 6, 2010, Art. no. 143.

[11] R. Wang, Y. Huo, Y. Yuan, K. Zhou, W. Hua, and H. Bao, "GPU-based out-of-core many-lights rendering," *ACM Trans. Graph.*, vol. 32, no. 6, 2013, Art. no. 210.

[12] Y. Huo, R. Wang, S. Jin, X. Liu, and H. Bao, "A matrix sampling-and-recovery approach for many-lights rendering," *ACM Trans. Graph.*, vol. 34, no. 6, 2015, Art. no. 210.

[13] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher, "Micro-rendering for scalable, parallel final gathering," *ACM Trans. Graph.*, vol. 28, no. 5, 2009, Art. no. 132.
[14] G. Nichols, R. Penmatsa, and C. Wyman, "Interactive, multiresolution image-space rendering for dynamic area lighting," *Comput. Graph. Forum*, vol. 29, no. 4, pp. 1279–1288, 2010.
[15] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing," in *Proc. Symp. Interactive 3D Graph. Games*, 2011, pp. 207–207.
[16] E. Paquette, P. Poulin, and G. Drettakis, "A light hierarchy for fast rendering of scenes with many lights," *Comput. Graph. Forum*, vol. 17, no. 3, pp. 63–74, 1998.
[17] A. W. Kristensen, T. Akenine-Möller, and H. W. Jensen, "Precomputed local radiance transfer for real-time lighting design," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1208–1215, 2005.
[18] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg, "Lightcuts: A scalable approach to illumination," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1098–1107, 2005.
[19] O. Akerlund, M. Unger, and R. Wang, "Precomputed visibility cuts for interactive relighting with dynamic BRDFs," in *Proc. 15th Pacific Conf. Comput. Graph. Appl.*, 2007, pp. 161–170.
[20] E. Cheslack-Postava, R. Wang, O. Akerlund, and F. Pellacini, "Fast, realistic lighting and material design using nonlinear cut approximation," *ACM Trans. Graph.*, vol. 27, no. 5, 2008, Art. no. 128.
[21] B. Walter, P. Khungurn, and K. Bala, "Bidirectional lightcuts," *ACM Trans. Graph.*, vol. 31, no. 4, 2012, Art. no. 59.
[22] T. Hachisuka, "High-quality global illumination rendering using rasterization," *GPU Gems*, vol. 2, pp. 615–633, 2005.
[23] L. Szirmay-Kalos and W. Purgathofer, "Global ray-bundle tracing with hardware acceleration," in *Proc. Eurographics Workshop Rendering Techn.*, 1998, pp. 247–258.
[24] Y. O'Donnell and M. G. Chajdas, "Tiled light trees," in *Proc. 21st ACM SIGGRAPH Symp. Interactive 3D Graph. Games*, 2017, pp. 1:1–1:7. [Online]. Available: http://doi.acm.org/10.1145/3023368.3023376
[25] Y. Tokuyoshi and T. Harada, "Stochastic light culling for VPLs on GGX microsurfaces," *Comput. Graph. Forum*, vol. 36, no. 4, pp. 55–63, 2017. [Online]. Available: https://doi.org/10.1111/cgf.13224
[26] Y.-T. Wu and Y.-Y. Chuang, "VisibilityCluster: Average directional visibility for many-light rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 9, pp. 1566–1578, Sep. 2013.
[27] Z. Dong, W. Chen, H. Bao, and H. Zhang, "Real-time voxelization for complex polygonal models," in *Proc. Pacific Conf. Comput. Graph. Appl.*, 2004, pp. 43–50.
[28] E. Eisemann, "Fast scene voxelization and applications," in *Proc. Symp. Interactive 3D Graph. Games*, 2006, pp. 71–78.
[29] E. Eisemann and X. Décoret, "Single-pass GPU solid voxelization for real-time applications," in *Proc. Graph. Interface*, 2008, pp. 73–80. [Online]. Available: http://dl.acm.org/citation.cfm?id=1375714.1375728
[30] J. Pantaleoni, "VoxelPipe: A programmable pipeline for 3D voxelization," in *Proc. ACM SIGGRAPH/Eurographics Conf. High Perform. Graph.*, 2011, pp. 99–106.
[31] M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on GPUs," in *Proc. ACM SIGGRAPH Asia Papers*, 2010, pp. 179:1–179:10.
[32] V. Kämpe, E. Sintorn, D. Dolonius, and U. Assarsson, "Fast, memory-efficient construction of voxelized shadows," *IEEE Trans. Vis. Comput. Graph.*, vol. 22, no. 10, pp. 2239–2248, 2016. [Online]. Available: https://doi.org/10.1109/TVCG.2016.2539955
[33] K. Myers, "Sparse shadow tree," in *Proc. Special Interest Group Comput. Graph. Interactive Techn. Conf.*, 2016, pp. 14:1–14:2. [Online]. Available: http://doi.acm.org/10.1145/2897839.2927418
[34] C. Wyman, "Voxelized shadow volumes," in *Proc. ACM SIGGRAPH Symp. High Perform. Graph.*, 2011, pp. 33–40.
[35] P. Voicu, C. Mei, D. Jordan, and S. Elisha, "Reflected scene impostors for realistic reflections at interactive rates," *Comput. Graph. Forum*, vol. 25, no. 3, pp. 313–322, 2006.
[36] A. Gaitatzes, A. Andreadis, G. Papaioannou, and Y. Chrysanthou, "Fast approximate visibility on the GPU using pre-computed 4D visibility fields," in *Proc. 18th Int. Conf. Central Eur. Comput. Graph. Vis. Comput. Vis.*, 2010, pp. 131–138.
[37] D. Kauker, M. Falk, G. Reina, A. Ynnerman, and T. Ertl, "VoxLink—Combining sparse volumetric data and geometry for efficient rendering," *Comput. Visual Media*, vol. 2, no. 1, pp. 45–56, 2016. [Online]. Available: https://doi.org/10.1007/s41095-016-0034-8
[38] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gunther, and P. A. Navrátil, "Ospray—A CPU ray tracing framework for scientific visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 931–940, 2017. [Online]. Available: https://doi.org/10.1109/TVCG.2016.2599041
[39] J. Bittner, O. Mattausch, P. Wonka, V. Havran, and M. Wimmer, "Adaptive global visibility sampling," in *Proc. ACM SIGGRAPH Papers*, 2009, pp. 94:1–94:10. [Online]. Available: http://doi.acm.org/10.1145/1576246.1531400
[40] Nvidia, "NVIDIA OptiX ray tracing engine," 2016. [Online]. Available: http://developer.nvidia.com/optix
[41] W. Engel, "Cascaded shadow maps," in *ShaderX5: Advanced Rendering Techniques*, vol. 1. Hingham, MA, USA: Charles River Media, 2006.

Lili Wang received the PhD degree from the Beihang University, Beijing, China. She is a professor with the School of Computer Science and Engineering, Beihang University, and a researcher with the State Key Laboratory of Virtual Reality Technology and Systems. Her interests include real-time rendering, realistic rendering, global illumination, soft shadow, and texture synthesis.



Xinglun Liang received the BS degree in computer science from North China Electric Power University, in 2016. He is currently working toward the master's degree in the State Key Laboratory of Virtual Reality Technology and Systems, Beihang University. His research interests include many lights, real-time rendering.

Chunlei Meng received the BS degree in computer science from North China Electric Power University, in 2014. He is currently working toward the master's degree in the State Key Laboratory of Virtual Reality Technology and Systems, Beihang University. His research interests include global illumination, real-time rendering.

Voicu Popescu received the BS degree in computer science from the Technical University of ClujNapoca, Romania, in 1995, and the PhD degree in computer science from the University of North Carolina at Chapel Hill, in 2001. He is an associate professor with the Computer Science Department, Purdue University. His research interests lie in the areas of computer graphics, computer vision, and visualization. His current projects include camera model design, visibility, augmented reality for surgery telementoring, and the use of computer graphics to advance education.