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

<table>
<thead>
<tr>
<th>Journal:</th>
<th><em>Transactions on Visualization and Computer Graphics</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuscript ID</td>
<td>Draft</td>
</tr>
<tr>
<td>Manuscript Type:</td>
<td>Regular</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Real time rendering, Many lights, Visibility determination, Photorealism</td>
</tr>
</tbody>
</table>

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.

17125.mp4
Constant-Time 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 constant time approximation 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. An image resolution of one million pixels and one thousand point light sources require computing one billion ray-scene intersections. 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 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 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. A 2D array of voxelizations contains 90x90 voxelizations (for a two degree direction discretization), and each voxelization has a 128x128x128 resolution. This requires 333MB 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 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.

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
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%, respectively. Our frame rates are 26fps, 23fps, and 13fps, 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%. Our frame rates are 15fps, 3fps, and 2fps, which corresponds to speedups of $12\times$, $6\times$, and $35\times$ versus ray tracing.
Visibility to a light source are shadow mapping and ray tracing. However, these methods are too slow for scenes with a large 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 [1].

2.1 Shadow map 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.

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 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 [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 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 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 constant time 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 orthographic 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 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 off 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 with many applications, and it 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 of 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 CONSTANT-TIME RAY-SCENE INTERSECTION

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

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 90 with \( k \) degree increment do
3: \( i = \theta/k; j = \phi/k; j' = (\phi + 90)/k; \)
4: \((V_{ij}, V_{ij'}) = \text{SPVoxelization}(\theta, \phi)\)
5: end for
6: end for
7: return \( V \)

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 \((\theta, \phi)\) with a \( k \) degree increment. Given a pair \((\theta, \phi)\), 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 \((\theta, \phi)\), 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 in constant time. For each \((\theta, \phi)\) 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 the \( z \) direction.

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 \( 90 \times 90 \) 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 2GB. 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.

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: \( H = \text{Histogram of all rows in } V \)
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: \( D'[i] = \text{Simplify}(D[i], 2^c) \)
6: end for
7: for each unmapped row \( i \) do
8: \( r' = \text{Simplify}(V'[i], 2^c) \)
9: if \((j = \text{Find}(r', D'))\) then \( V'[i] = j \)
10: end if
11: end for
12: end for
13: for each remaining unmapped row \( i \) do
14: \( V'[i] = \text{Random}(0, n-1) \)
15: end for
16: return \((D, V')\)

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 the dictionary \( D \) (lines 3-15). The algorithm uses a sequence of \( s \) compression steps that are sorted 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.

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 \((Nb) / (nb + N\log_2 n) \approx b / \log_2 n \). 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 2GB to 333MB, and the compression losses translate to minimal shadow value 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 \((\theta, \phi)\) 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 \( \theta = 30^\circ \) and \( \phi = 42^\circ \), therefore the ray-scene intersection is approximated using the voxelization \( V'[30/2][42/2] \).

**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. \((\theta, \phi) = \text{DiscretizeDirection}(r.direction, k)\)
2. \(i = \theta/k; j = \phi/k;\)
3. \(row = D[V'_{ij}.LookupRow(r.midpoint)]\)
4. \((s, e) = V'_{ij}.\text{ComputeExtents}(r.endpoints)\)
5. \(clippedRow = row \ll s\)
6. \(clippedRow = clippedRow \gg (s + rowLength - e)\)
7. \(\text{return} \ (\text{clippedRow} \neq 0)\)

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% quality improvement for a 45% performance penalty).

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 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, 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 0000 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 4.

**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. \( I = \text{Render} \ S \) from \( C \) without lighting
2. for every pixel \( p \) in \( I \) do
3. \( P = \text{Unproject}(p, C)\)
4. \( \text{shadow} = 0 \)
5. for every light \( L_i \) in \( L \) do
6. \( r = (P, L_i)\)
7. \( \text{shadow} += \text{Intersect}((V', D), \text{ray})\)
8. end for
9. \( p.\text{outputColor} = \text{Shade}(\text{shadow}, n)\)
10. 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 2-10). 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 \( (P, L_i) \) intersects the scene (lines 5-8).

### 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.

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 \((V'_a, D_a)\) 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 \(\text{Intersect}((V'_s, D_s), k, r_j)\) then
2. \(\text{return true}\)
3. end if
4. for all dynamic object instances \(DO_j\) do
5. \(r_j = \text{Transform}(r, CS_j)\)
6. \(i = \text{DO}_j\text{.objectType}\)
7. if \(\text{Intersect}((V'_a, D_a), i, r_j)\) then
8. \(\text{return true}\)
9. end if
10. end for
11. \(\text{return false}\)

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 \(V_s\), 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. Instead of using single pass voxelization for every \((\theta, \phi)\) 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.

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 \((\theta, \phi)\) 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, Figure 1 left), \(Garden\) (416ktris, middle), \(Cathedral\) (456ktris, 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 \(Trees\), \(Garden\), 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 3.5GHz Intel(R) Core(TM) i7-4770 CPU, with 8GB of RAM, and with an NVIDIA GeForce GTX 1080 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, \(\epsilon_v\), 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, \(\epsilon_s\), is defined as the percentage shadow value error at a pixel. For the example used above, \(\epsilon_s = (10 - 5)/1,000 = 0.5\%). \(\epsilon_s\) is a stricter error measure since for \(\epsilon_s\) errors can cancel each other out. \(\epsilon_s\) 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]). Furthermore, we
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

<table>
<thead>
<tr>
<th>Scene</th>
<th>Trees</th>
<th>Garden</th>
<th>Cathedral</th>
<th>Planes</th>
<th>Bear</th>
<th>Park</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε_v [%]</td>
<td>10.5</td>
<td>6.0</td>
<td>4.8</td>
<td>4.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_s [%]</td>
<td>1.8</td>
<td>3.4</td>
<td>1.7</td>
<td>2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISM</td>
<td>14.4</td>
<td>10.6</td>
<td>6.4</td>
<td>5.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_v [%]</td>
<td>1.7</td>
<td>3.4</td>
<td>2.1</td>
<td>2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_s [%]</td>
<td>6.7</td>
<td>4.5</td>
<td>3.2</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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. 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 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). Figure 6 visualizes the approximation errors from Table 1, highlighting the smaller errors of our method compared to ISM.

### Table 2

<table>
<thead>
<tr>
<th>Lights</th>
<th>512</th>
<th>1,024</th>
<th>2,048</th>
<th>4,096</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε_v [%]</td>
<td>5.0</td>
<td>4.7</td>
<td>4.6</td>
<td>4.5</td>
<td>4.7</td>
</tr>
<tr>
<td>ε_s [%]</td>
<td>1.6</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
<td>1.7</td>
</tr>
</tbody>
</table>

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 × 128 × 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 × 64 × 64 could be used in applications where memory is at a premium.

### Table 3

<table>
<thead>
<tr>
<th>Voxellization resolution</th>
<th>32^3</th>
<th>64^3</th>
<th>128^3</th>
<th>256^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>ε_v [%]</td>
<td>10.5</td>
<td>6.0</td>
<td>4.8</td>
</tr>
<tr>
<td>ε_s [%]</td>
<td>1.8</td>
<td>3.4</td>
<td>1.7</td>
<td>2.1</td>
</tr>
<tr>
<td>Garden</td>
<td>ε_v [%]</td>
<td>14.4</td>
<td>10.6</td>
<td>6.4</td>
</tr>
<tr>
<td>ε_s [%]</td>
<td>6.7</td>
<td>4.5</td>
<td>3.2</td>
<td>3.0</td>
</tr>
</tbody>
</table>

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.

### Table 4

<table>
<thead>
<tr>
<th>Voxelization rotation resolution</th>
<th>30 × 30</th>
<th>60 × 60</th>
<th>90 × 90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>ε_v [%]</td>
<td>6.4</td>
<td>5.3</td>
</tr>
<tr>
<td>ε_s [%]</td>
<td>2.6</td>
<td>2.1</td>
<td>1.8</td>
</tr>
<tr>
<td>Garden</td>
<td>ε_v [%]</td>
<td>10.6</td>
<td>8.3</td>
</tr>
<tr>
<td>ε_s [%]</td>
<td>4.1</td>
<td>2.8</td>
<td>2.7</td>
</tr>
</tbody>
</table>

### 6.2 Speed

We have implemented our method using shaders. We compute our 2D array of voxelizations by extending a prior-art 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 BVH (bounding volume hierarchy) scene partitioning for acceleration, which yields the fastest Optix rendering times. The Optix rendering times do not include the BVH construction time for the static scenes, i.e. Trees, Garden, and Cathedral, and they do include the BVH construction time for the dynamic scenes, i.e. Planes, Bear, and Park.

### Table 5

<table>
<thead>
<tr>
<th>Scene</th>
<th>Trees</th>
<th>Garden</th>
<th>Cathedral</th>
<th>Planes</th>
<th>Bear</th>
<th>Park</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours [ms]</td>
<td>38</td>
<td>43</td>
<td>59</td>
<td>66</td>
<td>312</td>
<td>441</td>
</tr>
<tr>
<td>Speedup vs. RT</td>
<td>43×</td>
<td>15×</td>
<td>21×</td>
<td>12×</td>
<td>6×</td>
<td>35×</td>
</tr>
<tr>
<td>Speedup vs. SM</td>
<td>52×</td>
<td>16×</td>
<td>89×</td>
<td>24×</td>
<td>10×</td>
<td>41×</td>
</tr>
<tr>
<td>Res. of SM</td>
<td>384</td>
<td>128</td>
<td>36</td>
<td>96</td>
<td>128</td>
<td>64</td>
</tr>
</tbody>
</table>
Fig. 5. Comparison between our method (left), ray tracing (middle), and imperfect shadow maps (right).
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 277ms and 210ms for the *Bear* and the *Park* scene (without compression), or 89% and 48% 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 $\epsilon_v$ to our method.

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 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 deforming geometry like the *Bear*, where ISM is five times faster than our method, which comes however at the cost of a shadow error $\epsilon_s$ that is twice as large (i.e. 4.9 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 1, and also of providing quality levels that cannot be matched by ISM, whereas ISM has a speed advantage for non-rigidly deforming scenes.

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.

<table>
<thead>
<tr>
<th>Lights</th>
<th>512</th>
<th>1,024</th>
<th>2,048</th>
<th>4,096</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trees</td>
<td>20</td>
<td>38</td>
<td>77</td>
<td>159</td>
<td>378</td>
</tr>
<tr>
<td>Garden</td>
<td>21</td>
<td>43</td>
<td>89</td>
<td>183</td>
<td>439</td>
</tr>
<tr>
<td>Cathedral</td>
<td>29</td>
<td>59</td>
<td>124</td>
<td>273</td>
<td>683</td>
</tr>
<tr>
<td>Planes</td>
<td>35</td>
<td>66</td>
<td>135</td>
<td>266</td>
<td>647</td>
</tr>
<tr>
<td>Bear</td>
<td>297</td>
<td>312</td>
<td>345</td>
<td>417</td>
<td>630</td>
</tr>
</tbody>
</table>

Fig. 6. Visualization of approximation errors $\epsilon_s$ for our method (top) and ISM (bottom), for the same frame rate, as reported in Table 1. 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.
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 pre-processing 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× to a manageable 333MB for the $128 \times 128 \times 128$ resolution.

#### 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 deform 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 Figure 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 333MB. The scene is rendered at 16Hz. 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. Figure 9 shows a Cornell box rendered with our approach with 1,024 real and virtual point light sources at 20fps ($128 \times 128 \times 128$ voxelization resolution, 333MB memory footprint).

#### 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 32x32x32 voxelizations for the airplanes.

We have demonstrated our technique with voxelization resolution of up to 256x256x256. This resolution might not be sufficient for large scenes with complex geometry. 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 90x90 discretization of ray directions and a 128x128x128 scene geometry discretization, which, with vector quantization compression totals a practical 333MB. Today’s GPUs also support 256x256x256 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 \( N\log_2 n \), 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 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 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 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 provide constant time 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


For Peer Review Only


Summary of Changes

The original submission is TVCG-2017-02-0050, "Constant-Time Ray-Scene Intersection for Interactive Rendering with Thousands of Dynamic Lights"
Manuscript Type: Regular

We thank the reviewers for their help with improving our manuscript. We have addressed every concern, through a very major revision that includes additional implementation, additional examples (TV scene to illustrate colored point light sources and Cornell Box to illustrate indirect illumination through virtual point light sources), and substantial rewriting. This major revision took longer than the time allotted by the reviewing system for a major revision, but we are resubmitting our work with the hope that it will be assigned to the same reviewers who have already generously invested their time towards improving our manuscript.

In this document we explain how each reviewer concern was addressed. We have opted for a side by side format that lists the reviewer concern followed by a summary of the concern was addressed in the revision.

Sincerely,

The Authors
<table>
<thead>
<tr>
<th>Reviewer number</th>
<th>Concern</th>
<th>How the concern was addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer #1</td>
<td>The biggest limitation is still the memory cost, even with the compression scheme. It should be made clear at the limitation section and at beginning of the paper.</td>
<td>We have added a clarification at the end of the introduction that our method achieves a constant time ray-scene intersection by trading off storage space, that the storage requirements are substantial, but already practical in the context of today’s GPUs (i.e. ~333MB for 128x128x128 voxelizations, and ~1.2GB for 256x256x256), and that the accuracy of ray-scene intersection is likely to go up as the storage capacity of GPUs increases allowing for finer ray direction and scene geometry discretizations. We have added a new subsection 6.3 that discusses our compression by vector quantization, showing the substantial memory reduction achieved (Table 8), the small shadow quality loss (Table 9), the reasonable pre-processing times (Table 10), as well as the small run-time performance penalty (Table 11). We pioneer a method that completely replaces the unbounded-cost ray-scene intersections with a data-intensive lookup method. We show that the 5D space of 2D ray directions x 3D scene geometry can be sampled adequately with practical storage requirements, demonstrating the feasibility of this approach located at the memory-intensive end of the storage versus computation trade off continuum. In addition to the its present practical value, practical value that is likely to increase in the future, the method makes an important theoretical contribution, which is likely to inspire additional applications and extensions, like other similar methods have done in the past, e.g. in light field rendering.</td>
</tr>
<tr>
<td></td>
<td>The temporal flickering is also a critical problem, which makes the method quite limited for large scenes, such as the sponza scene in the video.</td>
<td>The temporal flickering has been alleviated, see revised video. The temporal flickering that was visible in the initial video was due to incorrect texture filtering and to insufficient voxelization resolution. The texture filtering issue is unrelated to our method, we have now corrected it by using mipmapping for the texture lookups. We have also addressed the insufficient voxelization resolution, by demonstrating the user of 256x256x256 voxelizations (as opposed to the original 128x128x128).</td>
</tr>
<tr>
<td></td>
<td>This is a shadow test algorithm and all images are rendered showing shadow effects. So the current title is misleading.</td>
<td>We have modified the title to indicate the only application of constant-time ray-scene intersection that our paper investigates. Extensions to other applications are only mentioned in the future work section. The new</td>
</tr>
</tbody>
</table>

For Peer Review Only
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The computation time and approximation error of the compression scheme is not well analyzed.</td>
<td>We have added a new subsection 6.3 that deals with the compression scheme performance exclusively. The new subsection reports the substantial memory reduction achieved (Table 8), the small quality loss (Table 9), the pre-processing times (Table 10), and the small online rendering performance penalty (Table 11).</td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Generally, the method is clear and practical. The limitation still comes from the heavy memory cost and the temporary flickering artifacts.</td>
<td>Please refer to Concerns 1, 2, and 4.</td>
</tr>
<tr>
<td>Reviewer #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>The method produces more error and is slower now (caption of Fig. 1)?</td>
<td>Indeed, the lossy compression scheme lowers quality and increases lookup cost. However, as noted in the new Section 6.3, the quality loss and the shadow rendering performance loss are minimal, whereas memory consumption is reduced substantially, make the vector quantization scheme an excellent trade off, which reduces memory consumption to very practical levels. Please also see Concerns 1 and 4.</td>
</tr>
<tr>
<td>7.</td>
<td>There is one additional scene (the Sibenik cathedral, labeled `Sponza' here).</td>
<td>The additional scene is indeed the Sibenik cathedral, we are now calling the scene Cathedral in the paper to avoid any confusion.</td>
</tr>
<tr>
<td>8.</td>
<td>Supplemental video that shows quantisation error, especially for cases where the lights are close by/cast distinct, almost sharp shadows.</td>
<td>We discuss in the limitations section the unsuitability of our method for rendering hard shadows, which could be addressed in future work by detecting the lights that cast hard shadows and by handling them with their own conventional shadow map. The hard shadow limitation is common to many soft shadow / many light rendering methods.</td>
</tr>
<tr>
<td>9.</td>
<td>Added tables 3+4 about memory (300MB compressed vs 2G before, for a moderate resolution of 128^3, probably responsible for the artifacts)</td>
<td>The video artifacts were alleviated by filtering the textures and by increasing the voxelization resolution, see Concern 2.</td>
</tr>
<tr>
<td>10.</td>
<td>I was surprised to see the vector quantization applied to rows of the volume, instead of to blocks or something in Morton order.</td>
<td>We have to apply vector quantization to rows of the volume in order to preserve our constant time ray scene intersection approximation. Attempting to leverage data locality by partitioning the voxelizations into blocks results in having to perform several lookups to reconstruc the row that contains a given ray.</td>
</tr>
<tr>
<td>Concern</td>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>I think the idea is interesting but the technique will not be very relevant in practise: the high storage and precomputation costs seem too much for the given quality that can be achieved. Given that the 5D lookup table will scale with $N^5$ when refining the resolution, it is likely that the storage cost will stay prohibitive even on future hardware. We have demonstrated that current GPUs can support a 256x256x256 voxelization. The data structure is 4D as the lookup unit is a voxelization row. Please refer to Concerns 1 and 4.</td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>The introduction is very long and hard to read. Especially the expression '2D array of voxelisations' is used very often without revealing what exactly it means. This makes it particularly hard to understand the paper since rays are more likely 5D and lines 4D. We have greatly condensed and clarified the introduction. The 2D array of voxelizations is introduced early and clearly.</td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>It took me until the end of Sec. 2.1 where redundancy is the topic to be able to guess that this means actually storing a 3D voxelisation per 2D ray direction.</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>Algorithm 1: this is a very non-uniform sampling pattern. Are there any artifacts due to that? We disagree with the reviewer, we voxelize with an orthographic projection, and scene geometry is sampled with the same density, no matter whether it is at the center or at the periphery of the scene.</td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>Sec 3: The symmetric voxelisation cases are computed only once but then stored twice? Isn't that wasting memory? We have clarified that we cannot recreate a row from columns in constant time, hence we need to store both north-south and east-west voxelizations. Of course, we do not compute and we do not store both north-south and south-north voxelizations.</td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>A voxel resolution of $128^3$ may not be enough for scenes with more interesting depth complexity. I think this is the cause of the artifacts seen in the video when light sources move together closely, and in the shadows of the pillars in the Sibenik cathedral. We have demonstrated the practicality of 256x256x256 voxelizations, and we have added proper texture filtering, which alleviate the flickering, please also see Concern 2.</td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>The method is limited to shadow queries. The argument drafted in Sec. 7 to extend it to first intersection queries is incomplete, you need to acquire a precise intersection point with material and normal and other geometric properties. We have extended the argument to explain that once the intersection point is located in a rotated voxelization, one can map in constant time to an original, unrotated voxelization that stores all ingredients needed for shading, such as color for first order reflections, or scene surface normal for higher order reflections.</td>
<td></td>
</tr>
</tbody>
</table>
18. When a voxelisation is performed already, standard voxel (cone) tracing should be compared (i.e. removing some memory concerns by only requiring one 128^3 voxelisation and ray tracing through this). At these small resolutions it is unclear to me how much speedup there is to be gained.

We have added a table (i.e. Table 7) at the end of Section 6.2, which compares our performance to tracing a ray diagonally through a main, unrotated voxelization. The table confirms that the speedup gained by rotating the voxelization is substantial.

19. I would also add these additional references:
Fast Approximate Visibility on the GPU using precomputed 4D Visibility Fields
Athanasios Gaitatzes Anthousis Andreadis
Georgios Papaioannou Yiorgos Chrysanthou
WSCG 2010
Adaptive Global Visibility Sampling
Jiri Bittner Oliver Mattausch Peter Wonka
Vlastimil Havran Michael Wimmer
Siggraph 2009

We have added a paragraph in Section 2.4 to discuss these two related papers.

20. small typos (note that these are still the same)
with the help of -> help of
epipolr -> epipolar
and 61109125, by the -> add space

We have proofread the paper and addressed all small typos.

21. These fast ray queries come at a significant storage (even with compression) and precomputation time cost. Since also the quality of the queries is not very good I am still not too enthusiastic about this submission. It is stressed in the resubmission that the fast voxelisation is done by previous work and not part of this project. Still the choice to use such a voxelisation is very intrinsic to this work, and I at least don't expect orders of magnitude improvement in voxelisation techniques.

The method is practical with today's GPU's. The method produces better shadows faster when compared to prior art. Please refer to Concern 1 and 4.
<table>
<thead>
<tr>
<th>Reviewer #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>22. However, I wish the paper was more honest about the complexity of the algorithm. What happens as the number of dynamic objects increases? What if every dynamic object is unique? Now you need N x 2 GB voxel data structures and your costs are O(N). How does the data structure scale to open worlds common in today's games? (Or is it only designed for scenes that can be fit in a limited regions that can be spanned by a 128^3 grid?) Can scale be increased, say using &quot;cascades&quot; similar to the cascades widely used by shadow algorithms today?</td>
</tr>
<tr>
<td>We are describing how we handle dynamic scenes, including the cost for doing so, in the introduction, in the specially dedicated Section 5, and in the results and discussion Section 6. We have now further clarified in the Limitations subsection that, like for any ray tracing method, dynamic scenes require updating the geometry dependent acceleration data structure for every frame. The local arrays of voxelizations attached to rigid dynamic objects are of lower resolution than those used for the entire scene (e.g. 32x32x32). Indeed, the approach of associating local arrays of voxelizations to individual dynamic objects breaks down if there is a large number of types of objects, or if the objects deform in a complex way. We have demonstrated computing the entire scene voxelization from scratch for every frame, whose cost is independent from the number of types of dynamic objects. We have now demonstrated voxelizations of 256x256x256 resolution, which can handle accurately scenes with higher complexity. Also in the limitations subsection we are acknowledging the fact that even our now higher voxelization resolution of 256x256x256 might prove insufficient for large scenes with complex geometry, and we are describing the future work direction of extending cascaded shadow map approaches to our technique. For every frame, a near light frustum is computed by intersecting the near part of the output view frustum with the union of the light frusta, and the near light frustum is handled with its own 2D array of voxelizations.</td>
</tr>
<tr>
<td>23. I think what will end up happening in a realistic application is that a hierarchy of voxelizations will be necessary, and this will move the proposed work out of &quot;constant time&quot; into the domain of more typical logarithmic time object hierarchies. This does not mean the proposed approach does not have advantages, but rather than exploring how the work will be used, it has been analyzed under fairly simplistic use cases.</td>
</tr>
<tr>
<td>We have expanded the Conclusions and Future Work Section to clarify that our method is compatible with prior hierarchical scene partitioning schemes. Our 2D array of voxelization can be used to model the leafs of hierarchical data structures, bringing the gain of reducing the depth of the hierarchy, and of better balancing. The challenge with hierarchical scene subdivisions is not the O(log n) cost, but rather balancing the data structure to avoid prohibitive worst case O(n) costs, and our 2D arrays of voxelizations will help address this challenge.</td>
</tr>
</tbody>
</table>
24. On prior work, I think the related work and background sections are largely sufficient. However, there has been a number of papers on compressed voxel representations for shadows (and, recently colors) by the group at Chalmers University. While not really overlapping with the proposed work, these algorithms explore similar issues in voxel visibility, and it would be useful to put your work in context. See recent papers by Erik Sintorn and Viktor Kampe, one example of which is "Fast, Memory-Efficient Construction of Voxelized Shadows" from TVCG. You may also want to consider the compressed shadow representations that are already shipping in games, e.g., this recent talk: http://dl.acm.org/citation.cfm?id=2927418. Additionally, a number of game developers are actively improving upon "tiled lighting," which is another way to handle environments with a massive number of light sources. These techniques neither use geometry approximations nor light clustering (in the traditional sense), which the authors propose as the two frequently used techniques for simplifying. Instead, by splitting the environment into tiles, each tile is typically affected by a limited number of lights, e.g., see here for one recent paper: dl.acm.org/citation.cfm?id=3023376 (and here for one using Russian Roulette style stochastic to do this in an unbiased way: http://jcgt.org/published/0005/01/02/). We thank the reviewer for the additional references. They have all been added and discussed in our Prior Work Section.

25. Overall, I think the work is well written. The algorithm in Sections 3 and 4 are somewhat cryptic, but this is partially due to the limited space. We have edited the pseudocode algorithm descriptions to decrease density (at the cost of adding a few lines), which makes the algorithms more accessible.
On evaluation, my main remaining complaint on the evaluation is the somewhat unrealistic selection of point lights -- in almost all the environments (except the theme park) the point lights seem randomly scattered around the scene in a fairly unrealistic manner. In these scenes, it is unlikely lights would be located in random locations, and as far as I can tell, they contribute little to the final image (except maybe a sense of a global ambient term).

I would much prefer if the thousands of lights were selected in a more realistic fashion. For instance, a common use of many thousands of lights is in indirect global illumination where thousands of virtual point lights are used. Typically these do not use visibility, so the proposed work would be directly applicable. This sort of environment would make much more realistic and understandable use cases, since many readers will understand the properties of these scenes. Additionally, use of colored point lights could allow this approach to simulate a television or other textured area light.

As suggested by the reviewer, we are now demonstrating our method with two additional examples, a TV scene where the TV is modeled with 1,024 real and virtual light sources, and a Cornell box, where the indirect illumination is approximated with 1,024 lights. See Extensions subsection 6.4, Figures 8 and 9 (images also included below), as well as additional video segment showing the two scenes.
27. The biggest weakness of the work is the somewhat naive claim that this approach is "constant time." While the intersection between a ray and voxelization is indeed a constant cost, given a fixed voxelization, this assumes a lot about the scene. Voxel grids proposed in this work top out at $128^3$, which is insufficient for accurate intersection in many larger environments. Additionally, real dynamic content (e.g., game content) can contain many instanced pieces of geometry. The proposed work deals with instanced geometry by testing one ray-voxel intersection per instance. This means, the algorithmic complexity is $O(N)$ for $N$ instances. Additionally, many of today's scenes contain many dynamic models, including trees and grass that sway in the leaves. It is unclear how these sorts of models fit into the proposed work.

Please refer to Concerns 1, 22, and 23.

28. ** About "Constant-Time Ray-Scene Intersection". I don't think it is a true "constant-time". When the voxelization resolution is small (e.g., $32^3$ or $64^3$), I agree that only one look-up is enough by using bit operations, however, when a large resolution is used (e.g. $1024^3$), multiple look-ups are needed. So the timing still depends on the resolution of the voxelization.

We are now clarifying in the Conclusions and Future Work Section that we use four channel, 32bit per channel textures, so a row is obtained with a single lookup. We have extended the resolution to 256bit rows, which requires two lookups. Constant time doesn't mean a single lookup, but rather a constant number of lookups. We have demonstrated quality shadows in large and complex scenes with a fixed ray-scene intersection const.

We agree that even larger and even more complex scenes might require voxelization resolutions that are not yet practical in the context of today's GPUs. Please also refer to concerns 22 and 23.

29. ** Timing of the precomputation stage. I didn't find the timing of voxelization construction and voxelization compressions. I am not sure whether I missed something.

We have provided the times for voxelization and compression, see subsection 6.3.3 and Table 10. The times are for a serial CPU implementation.

30. ** Voxelization Resolution and Rotation Resolution. The current resolutions used are $128^3$ and $90^2$, it is somehow low. Such resolution might be enough for low frequency indirect lighting, but I am not sure whether it is good for high frequency direct lighting. Is it possible to use resolutions of $512^3$ and $512^2$, respectively?

We have now demonstrated our technique in the context of $256x256x256$ voxelization resolutions. As described above (Concern 28) this implies two lookups to fetch the row data, as well as a larger memory footprint, i.e. 1.3GB. It is likely that future GPUs will be able to handle $512x512x512$ resolutions. The question is what percentage of the GPU memory is an application willing to devote to lighting. We speculate that this percentage might go up once the method is extended to handle additional effects (see Future Work section, and Concern 17 discussing extension to reflections).
We agree that hard shadows are a challenge for our method, as they are for other methods, please refer to Concern 8.

** "Zigzag" artifacts.**
Imagine a scene is illuminated using only one light, and the light is moving. It seems that the shadow may “jump” when the light is moving across a voxel row boundary. If that is true, I suggest adding more discussions on this artifacts and discuss possible solutions for alleviating it.

The biggest challenge for a single light is that the shadow becomes hard and it will reveal the blocker geometry approximation, please refer to Concern 8.

** The following papers are suggested to cite:**
Daniel Kauker, Martin Falk, Guido Reina, Anders Ynnerman, Thomas Ertl. VoxLink - Combining sparse volumetric data and geometry for efficient rendering. 45-56. CVM 2016.

We thank the reviewer for the additional reference suggestions. They were added and discussed in our prior work section. We compared our method with the voxel lists method in the last paragraph of Section 6.2, and our method is much faster.