# T-ReX: Interactive Global Illumination of Massive Models on Heterogeneous Computing Resources

Tae-Joon Kim, Xin Sun, and Sung-Eui Yoon, *Senior Member*, *IEEE*

**Abstract**—We propose several interactive global illumination techniques for a diverse set of massive models. We integrate these techniques within a progressive rendering framework that aims to achieve both a high rendering throughput and an interactive responsiveness. To achieve a high rendering throughput, we utilize heterogeneous computing resources consisting of CPU and GPU. To reduce expensive data transmission costs between CPU and GPU, we propose to use separate, decoupled data representations dedicated for each CPU and GPU. Our representations consist of geometric and volumetric parts, provide different levels of resolutions, and support progressive global illumination for massive models. We also propose a novel, augmented volumetric representation that provides additional geometric resolutions within our volumetric representation. In addition, we employ tile-based rendering and propose a tile ordering technique considering visual perception. We have tested our approach with a diverse set of large-scale models including CAD, scanned, simulation models that consist of more than 300 million triangles. By using our methods, we are able to achieve ray processing performances of 3 M∼20 M rays per second, while limiting response time to users within 15∼67 ms. We also allow dynamic modifications of light, and interactive setting of materials, while efficiently supporting novel view rendering.

**Index Terms**—Massive models, ray tracing, photon mapping, global illumination, heterogeneous parallel computing, voxels, compression

## 1 Introduction

The complexity of polygonal models has been increasing dramatically in both areas of computer-aided design (CAD) and entertainment. This continuing trend is mainly caused by the ever-growing demands of achieving higher accuracy for CAD and better realism for movies and games. This in turn causes significant challenges to high-quality visualization and rendering because of the heavy loads of computation and memory. The main bottleneck of rendering massive models that cannot fit into the memory of CPU or GPU is the data transmission time introduced by fetching data from external drives (e.g., HDD or SSD) or between CPU and GPU because of the limited bandwidth such as PCI-Express. The excessive data transmission costs hinder high rendering throughput and interactive responsiveness.

Most prior methods for rendering massive models mainly have been focused on providing basic visual effects such as local illumination and hard shadows [1]. Supporting global illumination requires significantly more computation than local illumination. More importantly, unlike coherent rays, such as the primary and shadow rays widely used in local illumination, secondary rays generated in global illumination, such as path tracing and photon mapping are incoherent and diverge into a wide area of a model, leading to excessive data loading given the limited available memory of CPU and GPU. As a result, the data transmission time of global illumination of massive models can take a larger portion compared to that of local illumination. Even though most prior techniques that are mainly designed for local illuminations show meaningful performance improvements, they show insufficient performance for interactive global illumination of massive models.

Recent GPUs provide high computational power and, thus, are capable of producing interactive high-quality global illumination. Nonetheless, because of the relatively limited video memory (e.g., 1∼2 GB), they can handle only small-scale models well with traditional in-core rendering methods.

In this paper, we propose novel techniques enabling interactive rendering of large-scale models consisting of hundreds of millions of primitives by highly utilizing the computational power of GPU and minimizing data transmission costs between CPU and GPU. The key idea is to use both geometric and volumetric representations for an input polygonal model to efficiently perform global illumination and utilize available heterogeneous computing resources of CPU and GPU.

Our hybrid representation named Tri-level Representations for eXpress rendering (T-ReX) consists of separate, three different levels of details (LOD) for the input model: the original polygonal representation, and coarse and fine volumetric representations (Section 4). We use the original, fully detailed geometric representation only at the CPU,

- *T.-J. Kim and S.-E. Yoon are with the Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea. E-mail: {tjkim.kaist, sungeui}@gmail.com.*
- *X. Sun is with Microsoft Research Asia, 131231, Building 2, Microsoft, Danling road #5, Haidian district, Beijing 100080, China. E-mail: sunxin@microsoft.com.*

*Manuscript received 30 Oct. 2012; revised 13 Apr. 2013; accepted 27 July 2013; published online 8 Aug. 2013.*
*Recommended for acceptance by P. Cignoni.*
*For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2012-10-0241.*
*Digital Object Identifier no. 10.1109/TVCG.2013.112.*

(a) Overview (b) Cockpit (c) Cabin (d) Engine

Fig. 1. These figures show photon mapping results of the Boeing 777 model consisting of 366 M triangles in different views. These results are progressively refined and are acquired after 40 k frames that take 8~12 minutes. More importantly, each rendering frame is provided to users with less than 100-ms latency time, while allowing dynamic changes on camera, light, and material setting.

while the two volumetric representations are used at GPU. The coarse volumetric representation is designed such that it can fit into the video memory of GPU, while the fine volumetric representation is stored in an external drive and fetched to the video memory asynchronously in an on-demand fashion.

We choose photon mapping as our global illumination rendering technique for massive models, since it has been known to handle a wide variety of rendering effects robustly; we extend our methods to another global illumination technique, ambient occlusion. We partition various types of rays required to perform photon mapping into two disjoint sets that do and do not require high geometric resolutions. For rays that generate high-frequency visual effects (e.g., primary rays), we use the geometric representation on the CPU side. For all the other rays that tend to generate low-frequency visual effects (e.g., gathering rays), we use our volumetric representation on the GPU side. This enables a significant reduction on the data transmission cost between CPU and GPU, leading to a lower requirement on the communication bandwidth. We then utilize the available communication bandwidth for asynchronously transmitting necessary portions of the fine volumetric representation to the video memory, and then progressively refine the rendering quality using the additionally loaded, finer volumetric representation. As a result, our system provides global illumination effects interactively for massive models, and then converges to a high-quality result quickly.

*Main contributions and results.* In summary, main contributions of this paper are as follows:

- Hybrid representation consisting of geometric and volumetric representations of massive polygonal models.
- Progressive rendering framework that utilizes heterogeneous computing resources and minimizes the data transmission costs.

The proposed techniques and system provide the following benefits:

- *High performance and interactive responsiveness.* By utilizing heterogeneous computing resources and minimizing data transmission costs, we are able to achieve ray processing performance of 3 M~20 M rays per second. More importantly, for various types of models with varying model complexity, our system provides photon mapping rendering results progressively within 15~67-ms response time, while allowing dynamic changes on camera, light, and material setting at runtime.
- *High complexity.* By using separate, decoupled multi-resolutions for CPU and GPU, we can achieve interactive responsiveness even for massive models (Fig. 1) consisting of up to 470 M triangles on commodity hardware. Also, our techniques mainly designed for massive models can handle small models robustly without much computation overhead over the state-of-the-art global illumination techniques specialized for small models.

To the best of our knowledge, the progressive rendering framework, integrated with our proposed techniques, is the first system that interactively performs photon mapping for massive models capable of dynamic changes on the camera, lights, and materials.

## 2 Related Work

In this section, we explain prior approaches of supporting global illumination for massive models.

### 2.1 Massive Model Rendering

There are orthogonal approaches for handling large-scale models: compact representation, cache-friendly, multiresolution, and so on.

#### 2.1.1 Compact Representations

Mesh-based representations [2], [3], [4] provide the most detailed representation for models including a spatial hierarchy for efficient ray tracing, but can require expensive memory space and I/O access time. As a mesh-based representation, we use a compressed version of it only for handling operations requiring high geometric resolutions at CPU.

A point-based approach [5], [6] like point clouds decouples illumination data from the geometry, and employs multiresolution techniques for efficient rendering. The irregular distribution of point samples enables high-quality indirect illumination effects, but also leads to heavy

computation costs. Gobbetti and Marton [7] showed interactive performance for local illumination of large-scale point clouds.

Recently, volume-based representations such as regular voxels are actively used for interactive performance. In this approach, the data of both geometry and estimated radiance are approximated as voxels in sparse voxel octrees [8], [9], [10], [11]. It is well suited to GPU architectures thanks to its compact storage and efficient traversal, and provides plausible rendering quality. Crassin [12] discussed some difficulties of sparse voxel octrees such as computing primary rays and detailed shadows that require a very high resolution of the voxels. We address these issues by using a separated geometric representation and our proposed augmented voxel representation for the shadow. VoxLOD [13] showed interactive color bleeding effects on massive models by using asynchronous voxel loading. We also use a similar asynchronous loading for out-of-core voxels for providing better quality in a progressive manner, when the data bandwidth is available. In addition, our rendering framework supports photon mapping and can generate more realistic outputs.

#### 2.1.2 Cache-Friendly Techniques

These techniques can be broken into out-of-core, i.e., cache-aware, and cache-oblivious techniques. Out-of-core techniques reduce the number of data fetching from disk [14] assuming a particular cache size. Cache-oblivious techniques were shown to improve the cache coherence across different cache sizes [15]. In the field of ray tracing, there are a few techniques that maximize cache utilization by reordering rays [16], [17], [18]. However, these techniques have not been widely applied to interactive global illumination because of their limited performance improvement; they can reduce, but not remove most of the expensive disk I/O accesses at runtime.

Wald et al. [19] demonstrated interactive visualization of a Boeing model consisting of 366 million triangles by using an out-of-core approach, but global illumination is not supported. In our method, we can provide a reasonable rendering quality efficiently based on the coarse volumetric representation that fits into the video memory, and then progressively refine it with other representations proving higher resolutions using CPU and GPU.

#### 2.1.3 Multiresolution

Extensive research efforts have been put into designing various multiresolution techniques for geometry [20], spatial hierarchy [21], and lighting [22]. Sparse voxel octrees [11] provide a multiresolution scheme for all of them efficiently. In this paper, we extend this volumetric representation to provide interactive global illumination for massive models.

### 2.2 Global Illumination

High-quality rendering techniques have been long studied, and good books are available [23], [24].

The unbiased Monte Carlo ray tracing approach (e.g., bidirectional ray tracing [25]) based on the rendering equation is the standard solution of global illumination, but converges to the reference slowly. Many extensions have been made to improve its performance while introducing bias. Two notable techniques among them are virtual point lights (VPLs)-based radiosity [26] and photon mapping [27]. In this work, we adopt photon mapping because it has been known to provide various rendering effects.

Recently, photon mapping has been extended to efficiently support an infinite number of photons given available memory [28], stochastic rendering effects [29], and robust error estimation with a progressive rendering framework [30]. These techniques can be naturally combined with our method that focuses on handling massive models.

To improve the performance of global illumination, a class of global illumination techniques decomposes rays into different sets and uses a representation tailored for each set of those rays [31]. In addition, these representations have varying resolutions and, thus, provide multiresolutions. Another class of approaches use volumetric representations [32], [11] for improving the performance of global illuminations by introducing bias or visual artifacts to the final rendering results. Our method adopts similar concepts of these techniques for global illumination with large-scale models.

In addition to these approaches, many different interactive techniques (e.g., image-space techniques) have been proposed. See a recent survey on this topic [33]. As emphasized in its list of open problems, most global illumination techniques have been mainly designed and tested for small-scale models. Supporting scalability and large-scale models remains one of under-addressed topics in the rendering field.

### 2.3 Progressive and Adaptive Sampling

Progressive rendering techniques have been widely accepted especially for interactive global illumination. Unbiased Monte Carlo ray tracing techniques are intrinsically progressive [34], and photon mapping was extended to be progressive [28].

Various sampling methods have been extensively studied and can be integrated within a progressive rendering framework. Most sampling techniques are based on the variance of the previous samples [35]. In addition, human perception is also taken into account to guide sampling [36]. In this paper, we also use two *saliency* metrics [41], [37] that can be efficiently evaluated.

### 2.4 Heterogeneous Computing Resources

Recently, global illuminations have been accelerated by using multiple heterogeneous resources such as CPUs and GPUs. Budge et al. [17] proposed a generalized data management scheme on CPU/GPU hybrid resources for path tracing. Unlike the previous approaches, we separate data representations for CPU and GPU and minimize expensive data transmission overheads between them. While Budge et al. generate unbiased images in tens of minutes to hours, our framework aims to progressively produce biased images with high responsiveness.

## 3 Overview

We propose a novel framework to utilized the computation power of both CPU and GPU. As a result, the framework can compute global illumination of massive models with

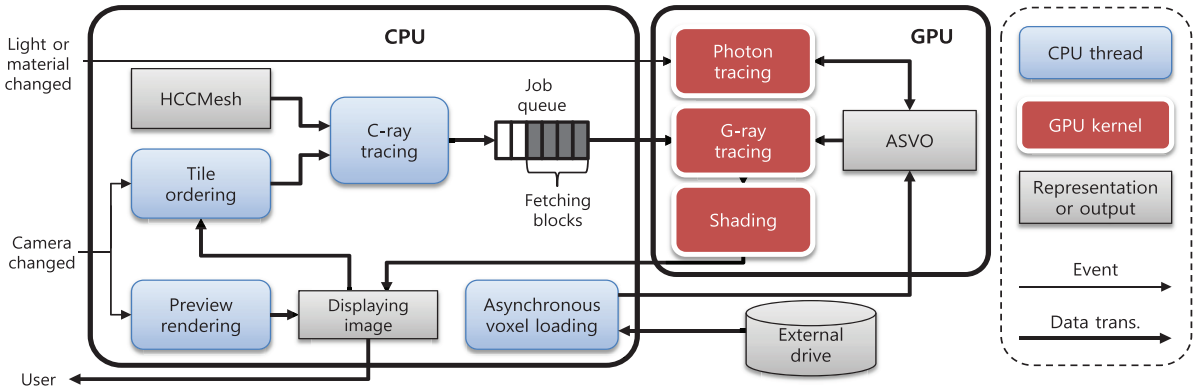



Fig. 2. This figure shows our rendering framework and data transitions between different modules.

high rendering throughput and responsiveness. In this section, we give an overview of our approach. We classify rays required to perform photon mapping into two disjoint sets called *C-rays* and *G-rays*, where C-rays and G-rays are rays that tend to create high-frequency and low-frequency rendering effects, respectively. We process C-rays on the CPU with a detailed, but compressed polygonal representation called HCCMesh [2], while process G-rays on the GPU with our volumetric representation, augmented sparse voxel octree (ASVO) (Fig. 2). At a high level, ASVO serves both as an approximated geometry for the input model and a volumetric representation of photons for indirect illumination; see Section 4.2 for more details.

We define C-rays to be the primary rays and their secondary rays that reflected on perfect specular materials, since they generate high-frequency rendering effects. All the other rays (e.g., gathering rays and shadow rays) are grouped together into G-rays, since they are likely to generate low-frequency effects, which can be constructed plausibly even with our approximated representation, ASVO.

We dedicate CPU to process C-rays, since CPU has a relatively large main memory that is required to hold the detailed polygonal models. On the other hand, most rays in G-rays are generated to produce low-frequency effects such as indirect illumination. In addition, the number of rays in G-rays is much higher (e.g., four to 12 times) than that in C-rays, leading to a higher computation load. As a result, we propose to use our volumetric representation ASVO and GPU to process those rays in G-rays, since the volumetric representation suits well to GPU.

*Runtime algorithm.* Fig. 2 shows an overall rendering framework that uses both CPU and GPU to compute direct and indirect illumination based on photon mapping. To compute indirect illumination, we perform a module of *Photon tracing* that generates and traces photons in the GPU side, and accumulate generated photons in our volume representation ASVO as discussed in [8].

We process rays tile-by-tile for better controlling the response time of our rendering framework. We, therefore, employ a *Tile ordering* module that orders tiles according to cache coherence and visual importance of tiles.

For each tile, we process C-rays associated with the tile in the CPU side by using the HCCMesh; this is conducted in a *C-ray tracing* module. Specifically, we perform intersection tests for C-rays against the mesh in the CPU side. If C-rays intersect with perfect specular materials, we recursively trace the rays until they do not intersected with any perfect specular materials or are not terminated based on the Russian roulette. We then send their last intersection results to the GPU side for processing G-rays, generated from those C-rays, with the ASVO representation in a *G-ray tracing* module and shading the final rendering output in a *Shading* module.

At the startup of our system, we first load the HCCMesh into main memory of the CPU. If the size of the HCCMesh is bigger than the main memory, we use an out-of-core version of the HCCMesh [2]. We then load a coarse version of our ASVO representation to the video memory of the GPU. Once these initial data loading operations are done, our system is ready to provide interactive response of a result to users.

At runtime, we run an *Asynchronous voxel loading module* that fetches necessary portions of the finer version of our ASVO asynchronously to provide better rendering results progressively; we do not send the original geometry to GPU at all. Such necessary portions are decided in a view-dependent and on-demand manner during a ray processing stage.

We also use a *Preview module*, which traces only primary rays in a reduced resolution (e.g., 100 by 100) that can be done quickly. This preview module let users can receive a new rendering result interactively, even when processing C-rays and G-rays in CPU and GPU takes much longer time.

## 4 Data Representations

In this section, we present our data representations for large-scale global illumination, followed by their preprocessing step.

### 4.1 Mesh Representation

As a detailed mesh representation for C-rays that produce high frequency effects, HCCMesh representation [2] is used. HCCMesh is a compact mesh representation that tightly integrates an input triangular mesh and its Bounding Volume Hierarchy (BVH) together. It reduces the size of the BVH by using connectivity templates of a hierarchy, and compactly encodes bounding volumes (BVs) based on vertices of the mesh. Furthermore, it provides random access on the compressed mesh and its BVH.

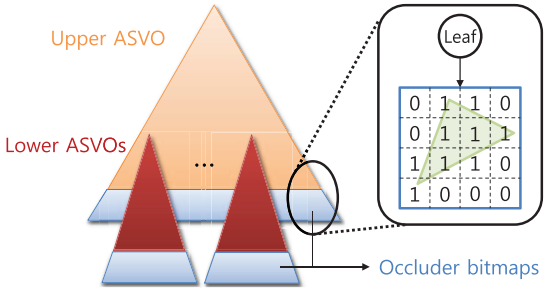
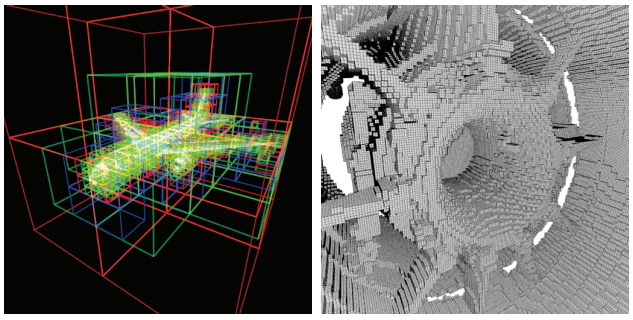



Fig. 3. *ASVO*: Our ASVO consists of upper and lower ASVOs, each of which is combined with occluder bitmaps for every leaf node. The occluder bitmap has bit values, each of which indicates whether its corresponding subvoxel overlaps with the original geometry.



(a) Bounding cubes of ASVO (b) Leaf voxels of ASVO

Fig. 4. The left figure visualizes bounding cubes of voxels that have a depth of six, while the right figure visualizes leaf nodes of the ASVO.

### 4.2 Augmented Sparse Voxel Octree

We use the ASVO for efficient handling of G-rays that produce indirect illuminations in the GPU side. ASVO consists of two different components that provide increasing higher resolutions: upper and lower ASVOs, and each of both has additional component called *occluder bitmaps* (Fig. 3). Unlike sparse voxel octrees [11], we store geometric information only in the leaf nodes of the upper and lower ASVOs, since storing the information in every node is not beneficial for our main application of handling large-scale models. The leaf node has the following structure:

```
struct LeafASVO {
  Material mat; // material information
  char phi, theta; // normal
  char occBitmap[8]; // occluder bitmap
  float r, g, b; // accumulated flux
  int num; // accumulated photon count
};
```

Listing 1: Our leaf node struture of the ASVO representation

The size of a leaf node can vary depending on the size of material information; we use 36 bytes in our implementation. The occluder bitmap is generated by subdividing a leaf voxel and assigning bit values to represent geometric information. This occluder bitmap can provide higher geometric resolutions for rays of G-rays that are sensitive to geometric resolutions (e.g., shadow rays).

We preload all the data of the upper sparse voxel octree and its corresponding occluder bitmaps to the video memory of the GPU and, thus, avoid their data transmission overhead between the CPU and the GPU at runtime. On the other hand, necessary portions of lower ASVOs (and their occluder bitmaps) are identified at runtime and asynchronously loaded to improve rendering quality progressively.

*Upper ASVO.* The upper ASVO is constructed such that it can fit into the video memory of GPU. In other words, its size is smaller than the available video memory of the GPU. As a result, the upper ASVO can be resident in the video memory and never swapped out at runtime. It has a $r_u^3$ resolution. In practice, we set $r_u$ to be in a range between 256 and 1k, resulting in a few hundred MBs (e.g., 300 MB).

We set the dimensions of the upper ASVO (hence its voxels) to be equal for efficient ray tracing (Fig. 4a). The bounding cube of the upper ASVO is recursively subdivided in the middle along each dimension to generate a sparse octree. All the nonempty nodes are stored in an array by the breadth-first search order. For each nonempty leaf node, we compute and record the representative normal and material information used for an illumination model (e.g., Phong illumination). The normal and material information are computed using triangles weighted by its intersected area with the voxel. This representative information in each leaf node serves as a LOD representation of geometry contained in each voxel, and is used for efficiently tracing multibounced photons and G-rays. Each internal node contains only pointers to its child nodes. Note that leaf nodes of the sparse octree do not contain the original geometry of the model nor any pointers to them; ASVO is totally a decoupled representation from the mesh.

*Lower ASVOs.* Conceptually, lower ASVOs have finer voxel resolutions than that of the upper ASVO. However, having lower ASVOs causes an increased memory requirement and more importantly, an increased data access time. In addition, there are potential overheads caused by synchronizations in the GPU side for connecting lower ASVOs to the upper ASVO as we discuss later.

To efficiently access lower ASVOs and reduce various synchronization operations, we create lower ASVOs for internal nodes in a particular depth, not for leaf nodes, of the upper ASVO, as shown in Fig. 3. Let us denote such internal nodes of the upper ASVO *linking nodes*. At runtime when we access a certain linking node of the upper ASVO, it is expected to access its subtree. We, therefore, prefetch its corresponding lower ASVO asynchronously [38] and connect it with the linking node of the upper ASVO. Since we cannot hold all the lower ASVOs in the video memory, we use a simple memory management method, clock algorithm, for unloading less-frequently used lower ASVOs.

The benefits of having lower ASVOs for internal nodes instead of leaf nodes come from the fact that the number of update operations drastically reduces, thus improves I/O throughputs. This is because the number of internal nodes is typically much smaller than the number of leaf nodes given the octree representation. In practice, we choose internal nodes whose depth is lower than leaf nodes in three levels for linking nodes, and thus, we reduce up to $8^3$ update operations. One may consider using a small size of the upper ASVO to avoid the overlap. We have found that this alternative provides low rendering quality before the lower ASVOs are loaded.

To use lower ASVOs at runtime, we need to connect them with linking nodes of the upper ASVO. To do so, we

Fig. 5. The left and right images show rendering results computed w/o and w/occlusion bitmaps with the resolution of $4^3$, respectively. When we do not use the occlusion bitmaps, shadow rays intersect with coarse voxels and produce false shadows, while bright spots are created since photons falsely interact with coarse voxels.

simply overwrite the child pointer of each linking node with the address of its corresponding lower ASVO after appropriate locking on data. Since we need only a single address update for each lower ASVO, this update can be done quite efficiently. We perform the connection and unloading process right after we process all the G-rays in the G-ray tracing module to reduce expensive synchronization. See the supplementary report for the analysis, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TVCG.2013.112.

*Occluder bitmaps.* The upper and lower ASVOs provide enough resolutions for various indirect illuminations (e.g., color bleeding) in our tested models, while providing an interactive rendering performance. Nonetheless, we found that it is necessary to have more detailed LOD representations of the geometry for visibility tests, especially for high-frequency shadows. To address this issue, we propose to use an occluder bitmap in each leaf node of the upper and lower ASVOs. The occluder bitmap of a leaf node provides additional visibility information for a voxel corresponding to the node. To construct the occlusion bitmap, we subdivide the voxel of the node into $r_o^3$ subvoxels, and check only whether each subvoxel is empty or not. We use this binary information of each subvoxel to provide higher geometry information for shadow rays (Fig. 5). In practice, we set $r_o$ to be 4, and thus requiring $4^3 = 64$ bits, or 8 bytes for each node.

In summary, our sparse octrees in ASVOs provide up to $(r_u * r_l)^3$ resolutions for the indirect illumination, while ASVOs with the occlusion bitmaps provide $(r_u * r_l * r_o)^3$ resolution for the geometry.

## 5 Rendering Algorithm

When we receive events of light or material changes, we trigger the photon tracing module (Fig. 2). For photon tracing, we use the common photon tracing method of the standard photon mapping approach [27]; we generate photons from light sources and bounce them with the model based on the Russian roulette. The difference between our method and the standard photon tracing is that we perform photon tracing in the GPU side and accumulate photons on leaf voxels [11] of both upper and lower ASVOs. Also, we progressively trace the photons when lights or materials are changed for better response. Once photon tracing is done, each leaf node of ASVOs maintains a radiance value of all the accumulated photons in its corresponding voxel.

When the user stays in a particular view point and, thus, gives time for our rendering system, we asynchronously perform photon tracing and gathering in the GPU side, and then show its result to the user in a progressive manner (See the supplementary report, available online). This is demonstrated in the accompanying video, available in the online supplemental material. Finally, shading is done with photon mapping, and we perform a joint bilateral filtering [39] using the G-buffer as its edge function, to reduce the variance level of indirect illuminations.

### 5.1 GPU-Based Photon Tracing

We generate and trace photons from light sources. When a photon hits a leaf voxel of an ASVO, its intensity are accumulated to the voxel. We simply average the intensities and store the single value into each leaf voxel.

We compute the outgoing direction of the intersected photon based on the normal and material information stored in the voxel. To decide whether a photon hits the geometry in a leaf voxel as its visibility test, we additionally use an occluder bitmap associated with the voxel. This effectively improves the quality of visibility tests and, thus, the overall rendering quality (Fig. 5). The traversal scheme of rays with occlusion bitmaps are same to that with ASVOs, since they are constructed based on regular grids as well.

When a user changes settings of lights or materials, we initialize the accumulated photon information stored in all the ASVOs and then restart the photon tracing module. Since photon tracing takes a lot of time in most cases, we generate new photons progressively and asynchronously in a background mode, then accumulate them in an additional, temporary buffer. While generating new photons with the updated settings in a background mode, we also process photon gathering performed in the G-ray tracing module for indirect illumination with the current ASVOs that are stored in the video memory. We then get radiance from both the current ASVO and temporary buffer, but with different weights. Initially, we give a higher weight to the current ASVOs, but to the temporary buffer. The weights are linearly increased or decreased as the rendering frame goes on; we finish the photon tracing module once it generates photons with a user-defined target number (e.g., 5 M photons for each light). Once all the photons are generated, the temporary buffers are swapped with the current ASVOs, and thus, we see rendering results only with the current ASVOs. In practice, it takes approximately 5 sec., i.e., around 300 frames, to trace 10 M photons for the cockpit viewpoint (Fig. 1b.)

If we do not allow users to modify the lighting/material settings, we can then perform the photon tracing step with our mesh representation, HCCMesh, for the best rendering quality, and bake its results in the ASVOs. At runtime, we need to perform G-ray tracing with the baked ASVOs and shading in the GPU side.

### 5.2 Tile-Based Rendering

We divide an image screen to tiles for better controlling the response time to users. We set each tile to have less than

100 pixels then process the tiles using SIMD-based packet tracing in CPU and GPU. Furthermore, we process multiple tiles simultaneously in C-ray and G-ray tracing modules with multiple working threads. We aim to provide a rendering result to a user within a user-specified threshold, $t_{max}$ for interactivity. $t_{max}$ is set to be 100 ms, which is the time we consider as the longest response time. As a result, it is possible to process only a portion of the tiles within a frame. If so, we process other tiles in its subsequent frames. If a user does not change the viewpoint, we can keep process tiles and provide progressively improved rendering results to the user at the viewpoint.

When we process a tile, we generate only a single primary ray for each pixel in a tile. We then generate $n_s$ and $n_g$, the shadow and final gathering rays spawned from each primary ray, respectively. Later, when we process the tile again after processing all the other tiles, we also apply the same procedure to the tile and, thus, improve its rendering quality in a progressive manner.

*Tile ordering.* There can be many options for ordering of processing tiles. The most common ordering methods for tiles include row-by-row or z-curve [40]. Z-curve is usually recommended for a higher performance, since it maximizes the cache coherence arising during processing tiles sequentially. We also identified that z-curve ordering shows the best rendering performance, but it does not accommodate users' preference on which regions he or she wants to see earlier than others.

To accommodate the users' preference, we propose a salience-based tile ordering. We estimate the users' preference by predicting important, i.e., salient, regions of the final reference image based on a salience metric. Since we cannot compute the final reference image, we use the prior rendering output. We adopt two saliency metrics proposed by Itti et al. [41] and Achanta et al. [37] because of their simplicity and efficiency (See the supplementary report, available online, for the visualization). Note that any efficient metrics can be adopted. For each tile, we evaluate the saliency metric for each pixel of the prior rendering output, and then compute an average saliency value for each tile. We then sort tiles based on its saliency values and process them sequentially.

We have tested with different tile orderings including our saliency-based, z-curve-based, random, and row-by-row ordering. We observed that z-curve-based ordering shows the best performance followed by row-by-row, ours, and random ordering. Nonetheless, the performance differences between our method and the z-curve are very small (e.g., up to 4 percent difference). As a result, we employ saliency-based tile ordering for our approach, since it achieves the best rendering quality in our progressive rendering framework with a reasonably high runtime performance.

Once processing a tile is done at the C-ray tracing module in the CPU side, we enqueue the tile and its associated information (e.g., hit points and material index of primary rays generated for the tile) to a *job queue* (Fig. 2), which contains tiles to be passed to the GPU side. Instead of sending an available tile to the GPU, we collect and send them in a block, called *fetching block*. Specifically, when the size of the job queue is bigger than a threshold, i.e., fetching block granularity, we dequeue all the tiles as a fetching block and send their information to the GPU side, followed by launching the G-ray tracing module that performs the final gathering and others in the GPU side. Once the G-ray tracing model is done, we perform the shading and then show its final result to the viewer.

The granularity of the fetching block is controllable based on a threshold set by a user. If the user prefers higher responsiveness, we need to use a smaller threshold (e.g., 64 tiles). On the other hand, when users target optimized rendering throughputs, larger fetching blocks (e.g., 2 k tiles) are recommended. A more detailed analysis is available in Section 6.2. We use the user-specified granularity of fetching blocks to respect the user's preference on the rendering throughputs. Nonetheless, if the response time of the current frame is larger than $t_{max}(= 100\ \mathrm{ms})$, then the size of fetching block is automatically reduced for the next frame to make the response time to be less than $t_{max}$. When the response time becomes within $t_{max}$, we gradually increase the fetching block size to the user-specified granularity.

### 5.3 G-Ray Tracing

From the hit points computed by processing primary rays of a tile in the CPU side, we generate shadow rays and final gathering rays in the GPU side and process them in the G-ray tracing module. We use an octree traversal algorithm [42] to trace both kinds of rays with both ASVOs and occlusion bitmaps in a similar manner that we trace photons in the photon tracing module. To maximize the utilization of the GPU, we process a bundle of rays simultaneously by considering the SIMT architecture of modern GPUs [43]; we map the bundle of rays to 32 threads, a *warp* in the recent NVIDIA GPU architecture. Since these threads for the bundle of rays in a warp execute one common instruction at a time, the utilization will be lowered when the threads have data-dependent conditional branches. To minimize such serializations, we perform a cache-oblivious ray reordering for rays [18]. In particular, we sort the rays based on their ray directions and then assign rays with similar directions to a warp.

### 5.4 Asynchronous Voxel Loading

When a ray traverses a linking node of the upper ASVO, we check whether its corresponding lower ASVO is loaded or not in the video memory. When the lower ASVO is loaded, it indicates that it is already linked to the linking node of the upper ASVO. As a result, we can keep traverse into the corresponding lower ASVO. On the other hand, when the lower ASVO is not loaded yet, we send a data loading request to the CPU side, and process the rays only with the information stored in the upper ASVO.

The voxel loading manager running asynchronously on the CPU side receives such requests. It then asynchronously loads the requested lower ASVOs in a separate CPU thread. Once a lower ASVO is loaded, it is then sent to the video memory asynchronously. As a final step, we connect it based on a simple pointer update, as discussed in Section 4.2.

## 6 Results and Comparisons

We have tested our method on a PC, which has 3.3-GHz Intel Core i7 CPU (hexa-core), 8-GB RAM, NVIDIA GTX 680

TABLE 1
This Table Shows Model Complexity, Size of Each Representation and Resolution of Voxels ($r_u$ and $r_l$) for Benchmark Models

| Model | Tri (M) | Size (MB) of HCCM. | u-ASVO | l-ASVO | $r_u$ | $r_u * r_l$ |
|---|---|---|---|---|---|---|
| Boeing 777 | 366 | 6708 | 243 | 5789 | 1024 | 4096 |
| Double Eagle Tanker | 82 | 1758 | 278 | 5549 | 1024 | 4096 |
| Power plant | 13 | 258 | 89.3 | 1898 | 1024 | 4096 |
| Sponza | 66k | 2.6 | 191 | 767 | 512 | 1024 |
| St. Matthew | 372 | 5612 | 150 | 629 | 512 | 1024 |
| Iso-surface | 469 | 7341 | 182 | 5268 | 256 | 1024 |

*HCCM. stands for the HCCMesh. u-ASVO and l-ASVO indicate upper and lower ASVOs, respectively.*

graphics card with 2-GB DRAM, and HDD. We have implemented our system on Windows7 and NVIDIA CUDA 4.2 toolkit. We allocate a certain portion of the available video memory to *an ASVO buffer* that permanently holds the upper ASVO, while the rest of the video memory is reserved for caching lower ASVOs. Specifically, 15 percent of the available video memory, which is 300 MB, is set for the ASVO buffer; a range of 10 to 50 percent works well without much performance and quality difference.

*Benchmarks.* We have tested our method with a diverse set of models (Table 1) that have different characteristics. Our main benchmark model is a Boeing 777 model (Fig. 1) consisting of 366 M triangles. The model takes 15.6 and 21.8 GB for its mesh and BVH, respectively. We encode its mesh and BVH compactly in a HCCMesh. The HCCMesh representation takes only 6.55 GB. We use 11 area lights for the model. In addition, we have tested our method with different CAD models including Double Eagle Tanker (82 M triangles), power plant (13 M triangles), and Sponza models (66 k triangles) (Table 3) with four, eight, and two area lights, respectively. The CAD models usually have irregular distributions of geometry and drastically varying triangle sizes. Other types of benchmark models include a St. Matthew model (372 M triangles) as a scanned model, and an isosurface model (469 M triangles) extracted from a scientific simulation (Table 3) with one and two area lights, respectively. Triangles in these models are distributed relatively regularly, but are highly tessellated.

### 6.1 Implementation Details

We elaborate implementation details that are important to be able to achieve a high performance of our method reported in this paper.

*Preprocessing.* Parameters $r_u$ and $r_l$ play a major role in terms of the overall performance and rendering quality. Since the upper ASVO should have the highest resolution while it fits within the ASVO buffer, we incrementally increase the value $r_u$ of the upper ASVO by a factor of 2 and use the maximum resolution value given the memory constraint. Our system allows that we can have a high-resolution $r_l$ for lower ASVOs, since they are fetched asynchronously on demand at runtime. As a result, we let users to set $r_l$ depending on the required resolution for a model. Detailed parameter values for each tested model are shown in Table 1.

*Runtime rendering.* We use the Russian roulette for tracing photons, but we set it such that the average number of bounces for photons is 3. We use the Phong illumination model for BRDF. To process primary rays efficiently in the C-ray tracing module, we adopt packet ray tracing [44]. Some of our benchmark models have many lights. Generating shadow rays for all the lights can be very time consuming, hindering interactive response to users. To efficiently consider many lights, we adopt a simple importance sampling scheme for lights. Whenever we need to generate shadow rays, we randomly select lights and generate shadow rays only for those selected lights. We use a simple heuristic of measuring the importance of lights; we set a probability of each light based on its light intensity and distance from the camera position. One can use more advanced techniques such as an adaptive technique proposed by Ward [45].

### 6.2 Performance

We show performance achievement mainly with the Boeing model, the most challenging benchmark model among our benchmark set. We also discuss performances with other models, if they show different results over those of the Boeing model.

Our unoptimized construction method for our representations processes 30 k triangles per second on average. For example, it takes about two and a half hours for the Boeing model.

*Runtime rendering.* A common method for evaluating performance of a rendering system is measuring its rendering performance with a predefined camera path. However, this evaluation protocol is not very meaningful to our case, since our system is progressive and focuses on delivering quick responsiveness to users (see the accompanying video, available in the online supplemental material). Instead, we have measured the average response times between a user event and its first result of our rendering system across various views. More specifically, we generate tiles for a new setting provided by a user and send tiles in a fetching block to GPU, followed by showing a result corresponding to those tiles. The *response time* is, thus, measured between the time when the user provided an event and the time that our system provides the initial result to the event. We also compute the *complete image time* that takes a time to process all the tiles of the final image. The complete image time is provided only for comparison with other nonprogressive rendering systems.

To measure response time in the Boeing 777 benchmark model, we choose views such as overview, cockpit, cabin, and engine, as shown in Fig. 1, following reference views listed by Wald et al. [19]. We use a 512-by-512 image resolution and Achanta et al.'s metric [37] as the saliency metric for all the tests. We test parameters $n_s$ and $n_g$ with two different values: $n_s = n_g = 2$, and $n_s = 4$ and $n_g = 8$. We generate 5 M photons for each light, since the rendering quality is almost converged with the number of photons [27].

As shown in Table 2, our approach shows the response time of 25.9~36.9 ms across different views when we use $n_s = n_g = 2$. These results directly indicate that users can get a feedback within this response time, even when they

TABLE 2
This Table Shows the Rendering Performance Including Response Time, *Resp. T*, and Ray Processing Throughput Measured in *M rays/s* at Different Views Shown in Fig. 1

| | | | $n_s$ | $n_g$ | Overview | Cockpit | Cabin | Engine |
|---|---|---|---|---|---|---|---|---|
| Ours | Resp. T | | 2 | 2 | 36.9 | 25.9 | 34.9 | 31.9 |
| | | | 4 | 8 | 36.3 | 36.0 | 67.3 | 60.3 |
| | M rays/s | | 2 | 2 | 3.4 | 10.4 | 9.3 | 9.5 |
| | | | 4 | 8 | 6.6 | 15.0 | 12.4 | 13.0 |
| | CIT | CPU | NA | NA | 141 | 89 | 64 | 64 |
| | | GPU | 2 | 2 | 43 | 81 | 123 | 111 |
| | | | 4 | 8 | 66 | 166 | 253 | 226 |
| | | Total | 2 | 2 | 152 | 106 | 141 | 130 |
| | | | 4 | 8 | 151 | 186 | 273 | 246 |
| CPU-GI | Resp. T | | 2 | 2 | 76.4 | 102.4 | 129.0 | 122.0 |
| | | | 4 | 8 | 132.0 | 206.6 | 279.6 | 273.8 |
| | M rays/s | | 2 | 2 | 1.6 | 2.6 | 2.4 | 2.4 |
| | | | 4 | 8 | 1.8 | 3.2 | 2.9 | 2.7 |
| | CIT | | 2 | 2 | 321 | 432 | 541 | 516 |
| | | | 4 | 8 | 560 | 877 | 1184 | 1170 |
| Full-GI | Resp. T | | 2 | 2 | 173 | 2407 | 6838 | 815 |
| | | | 4 | 8 | 404 | 7088 | 25895 | 2625 |
| | M rays/s | | 2 | 2 | 0.62 | 0.11 | 0.039 | 0.31 |
| | | | 4 | 8 | 0.45 | 0.080 | 0.027 | 0.25 |
| | CIT | | 2 | 2 | 822 | 10120 | 33701 | 4009 |
| | | | 4 | 8 | 2215 | 34991 | 126444 | 12568 |

*We also report complete image time, **CIT**, for comparison with other work. Time is reported in* ms *unit.*

modify camera, lighting, and materials. While providing this interactive responsiveness, our method also achieves 3.4 M~10.4 M rays/s across different views. In terms of complete image time, our method generates seven to nine complete images per second. When we use bigger $n_s$ and $n_g$ (i.e., $n_s = 4$ and $n_g = 8$), we can achieve higher ray throughputs (6.6 M~15.0 M rays/s), but longer response time (36.0~67.3 ms). Since the response time with $n_s = 4$ and $n_g = 8$ may not be preferred for interactive applications, the parameters $n_s = n_g = 2$ are chosen and used in the accompanying video, available in the online supplemental material.

To see the utilization of the CPU and GPU, we also measure the time spent on each computing resource when we process all the tiles in the screen space. Since the CPU and GPU run simultaneously, the total complete image time is slightly longer than the maximum of each time spent on CPU and GPU. The CPU is the main bottleneck for overview and cockpit viewpoints, but the GPU for cabin and engine viewpoints. In all the cases, our method shows response time around 30 ms.

*Fetching block granularity.* The ray processing throughput and responsiveness of our system depend heavily on the fetching block granularity. To find reasonable ranges for the parameter, we first tested various sizes of fetching blocks with the fixed setting of $n_s = n_g = 2$ (Fig. 6). We tested with the Boeing 777 model at the overview and cockpit viewpoints, and all the other parameters are same to the ones used for prior experiments. We observed the natural tradeoff between the ray processing throughput and response time, as we increase the fetching block size. We found that using block sizes from 128 to 512 is a good compromise in terms of both throughput and responsiveness. For the rest of tests, we use 512 as the default fetching block size. For the St. Matthew scene, the size is, however,

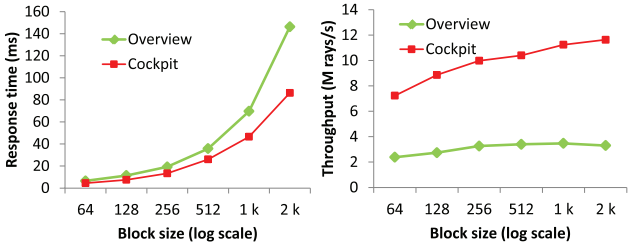



Fig. 6. These graphs show response time and ray processing throughputs as a function of the fetching block sizes.

automatically reduced to 128, to make the response time within $t_{max}$ as discussed in Section 5.2.

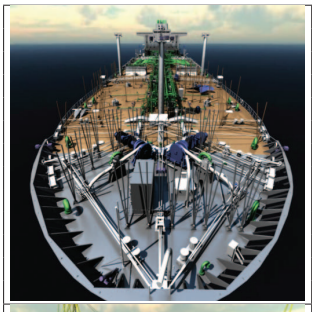
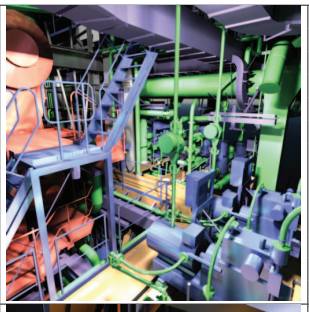
*Limited main memory.* Since HCCMeshes and upper ASVOs of all of the tested models fit into the main memory of our tested system, we can preload them. Therefore, at runtime, disk I/O occurs for only lower ASVOs. To see the performance and behavior of our framework with a smaller size of main memory, we have also tested the Boeing benchmark with 2 and 4 GB of main memory. This setting makes our rendering system run in an out-of-core manner in terms of HCCMeshes. To achieve high responsiveness, we set 128 for the block size. When we test at the cockpit viewpoint, we observed that our framework shows the response time of 7.1 and 6.8 ms, and ray processing throughput of 8.0 and 8.2 M rays/s on average, when we use 2 and 4 GB, respectively. Both results are similar to results achieved with 128 block size in Fig. 6, which is observed in an in-core case where all the HCCMeshes are preloaded. These results are achieved, mainly because we use the fixed viewpoint and the most necessary parts of the model are cached. We additionally performed a stress test that quickly alternates the overview and cockpit viewpoints to raise many cache misses. We achieve 5.7 and 6.1 M rays/s on average, and response times of 10.2 and 8.4 ms. The peak response time is 73.7 ms with 2-GB memory and 64.0 ms with 4-GB memory, both of which are still less than $t_{max}(= 100$ ms$)$.

*Other benchmark models.* We reported results mainly with the Boeing model so far. We also discuss results with other models among our benchmark set. We achieve 4.7 M~20.2 M rays/s and response time of 15.3~60.4 ms for other models. CAD models such as Double Eagle tanker, power plant, and Sponza models show similar performance trends to the Boeing 777 model, even though they have varying model complexity, i.e., more than three orders of magnitude difference in terms of triangle counts. From these results, we can conclude that our method shows a robust performance with a largely varying model complexity. This is mainly because the voxel-based representation of ASVOs is decoupled from the original geometry.

On the other hand, the St. Matthew and isosurface models show different results over CAD models. In these models, especially the St. Matthew model, the main computational bottleneck is on operations performed at the CPU, since many triangles are mapped to a single tile (i.e., 300~400 triangles per a pixel), leading to ineffective utilization for the packet tracing in the CPU side. To verify this, we disabled packet tracing and measured the performance again with these models. We found that the

TABLE 3
Rendering Performance with Our Benchmark Models

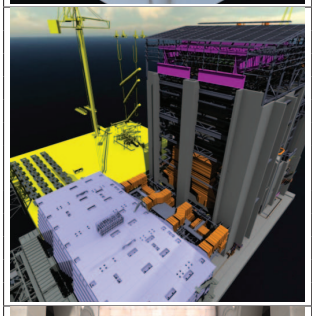
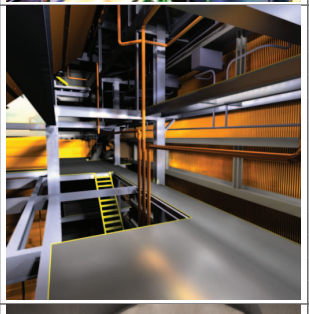
| View1 | View2 | |
|---|---|---|

Double Eagle Tanker (82 M triangles)

| Size (MB) of HCCMesh | 1758 |
|---|---|
| Size (MB) of u-ASVO | 278 |

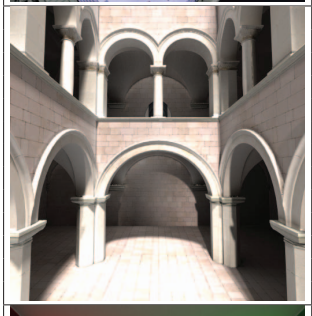
| | View1 | | View2 | |
|---|---|---|---|---|
| $n_s/n_g$ | 2 / 2 | 4 / 8 | 2 / 2 | 4 / 8 |
| Resp. T | 26.1 | 32.7 | 28.1 | 54.0 |
| M rays/s | 7.9 | 14.8 | 11.2 | 15.1 |
| CIT | 105 | 133 | 115 | 219 |
| CPU T | 94 | 93 | 94 | 90 |
| GPU T | 61 | 114 | 97 | 202 |

Power plant (13 M triangles)

| Size (MB) of HCCMesh | 258 |
|---|---|
| Size (MB) of u-ASVO | 89.3 |

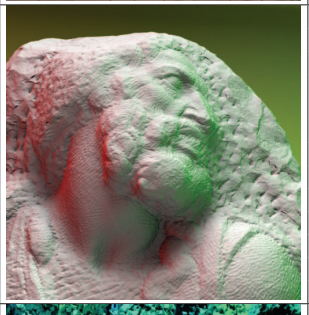
| | View1 | | View2 | |
|---|---|---|---|---|
| $n_s/n_g$ | 2 / 2 | 4 / 8 | 2 / 2 | 4 / 8 |
| Resp. T | 18.2 | 30.3 | 21.8 | 41.4 |
| M rays/s | 9.5 | 13.1 | 11.9 | 15.8 |
| CIT | 104 | 187 | 109 | 213 |
| CPU T | 43 | 42 | 30 | 36 |
| GPU T | 88 | 170 | 94 | 196 |




Sponza (66 k triangles)

| Size (MB) of HCCMesh | 2.6 |
|---|---|
| Size (MB) of u-ASVO | 191 |

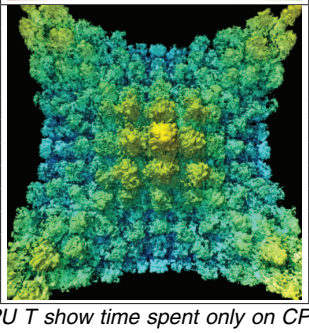
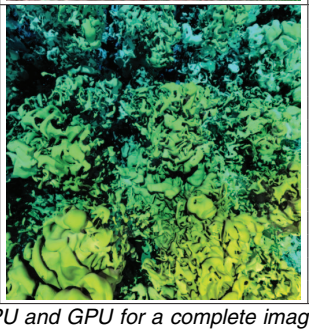
| | View1 | | View2 | |
|---|---|---|---|---|
| $n_s/n_g$ | 2 / 2 | 4 / 8 | 2 / 2 | 4 / 8 |
| Resp. T | 20.0 | 37.4 | 15.3 | 27.4 |
| M rays/s | 12.6 | 16.8 | 14.7 | 20.2 |
| CIT | 104 | 203 | 88 | 166 |
| CPU T | 15 | 15 | 11 | 11 |
| GPU T | 89 | 186 | 72 | 147 |

St. Matthew (372 M triangles)

| Size (MB) of HCCMesh | 5612 |
|---|---|
| Size (MB) of u-ASVO | 150 |

| | View1 | | View2 | |
|---|---|---|---|---|
| $n_s/n_g$ | 2 / 2 | 4 / 8 | 2 / 2 | 4 / 8 |
| Resp. T | 26.1 | 50.1 | 51.9 (40.0) | 52.3 (40.5) |
| M rays/s | 10.5 | 13.4 | 1.5 (7.6) | 3.9 (16.0) |
| CIT | 125 | 255 | 848 (172) | 863 (213) |
| CPU T | 64 | 63 | 840 (161) | 852 (164) |
| GPU T | 111 | 237 | 91 | 198 |

Iso-surface (469 M triangles)

| Size (MB) of HCCMesh | 7341 |
|---|---|
| Size (MB) of u-ASVO | 182 |

| | View1 | | View2 | |
|---|---|---|---|---|
| $n_s/n_g$ | 2 / 2 | 4 / 8 | 2 / 2 | 4 / 8 |
| Resp. T | 51.8 | 53.2 | 60.4 | 61.5 |
| M rays/s | 4.7 | 11.4 | 4.8 | 12.1 |
| CIT | 224 | 229 | 276 | 281 |
| CPU T | 215 | 220 | 269 | 272 |
| GPU T | 68 | 115 | 88 | 160 |

*CPU T and GPU T show time spent only on CPU and GPU for a complete image time, CIT, respectively.*

rendering system without packet tracing shows higher performance (about four times) than using packet tracing; parenthesized results in Table 3 are achieved without packet tracing. Therefore, we found that it is not a good choice to use packet tracing for these kinds of models. Data structures for improving the performance even for such incoherent rays were proposed [46]. Even though it is not investigated further, it is straightforward to adopt this scheme in our method.

*Extensions to other global illumination techniques.* Even though we demonstrated our method mainly with photon mapping, our method can be easily extended to support

other kinds of global illumination. For example, we can adopt ambient occlusion by tracing random rays using our ASVOs from each visible point, in addition to using the HCCMesh for primary rays. We tested progressive ambient occlusion tracing 10 rays per a frame for each visible point and observed 16.6 M rays/s and 40.9-ms response time for the cockpit viewpoint (See the supplementary report, available online, for the rendering results).

### 6.3 Comparisons

To highlight the benefits of our approach, we compare our CPU/GPU hybrid rendering algorithm with a framework that runs entirely on CPU. We call this CPU-based framework *CPU-GI*. In addition, we compare our method with a framework that uses the original, full detailed model, HCCMesh, even for shadow and gathering rays; in other words, no geometry approximations are used for this framework at all. We call this framework *Full-GI*. For Full-GI, because we do not use ASVOs, photons are recorded in a separate kd-tree as the usual photon mapping method. Full-GI runs entirely on CPU because the full detailed model cannot be loaded into GPU.

*Comparisons with CPU-GI and Full-GI.* We achieve 3.9 times improvements on average compared with CPU-GI. The major difference between ours and CPU-GI is that modules of photon tracing and G-ray tracing are performed in the CPU side for CPU-GI. Therefore, this result indicates that these modules are more efficiently performed in the GPU side. This is mainly because traversal algorithms are performed on ASVOs, which are defined on a regular grid and, thus, are well suited for various GPU operations.

To see benefits of using only the ASVOs, we compare CPU-GI with Full-GI, since the CPU-GI uses our representation in the CPU side, while the Full-GI running also in the CPU side does not use it. CPU-GI achieves 3.3, 32, 84, and 9.3 times performance improvement on average over Full-GI for the overview, cockpit, cabin, and engine viewpoints, respectively. Complete image times at the cockpit and cabin viewpoints using Full-GI are much longer than those measured in other viewpoints because photon densities needed for these viewpoints are much higher than others, and hence, K-Nearest Neighbor (KNN) search takes a much longer time. On the other hand, using ASVOs for photon gathering is independent to the density of traced photons because the photons are accumulated to voxels. As a result, our method shows steady performance across different regions and viewpoints.

Overall our method utilizing CPU and GPU achieves 135 times improvement on average over Full-GI. Nonetheless, results of our method are approximations to those of Full-GI (Fig. 7); results computed by Full-GI are reference images computed by photon mapping. The major difference comes from the fact that our volumetric representation conservatively covers more space than the original mesh. As a result, this conservativeness causes false-positive ray intersections.

*Comparison with GPU only framework.* We have tested a GPU only framework that also processes C-rays on GPU. To analyze the performance of the processing C-rays on GPU, we have implemented a GPU ray tracer, which is two times faster than NVIDIA Optix [47]. We trace primary rays with the Stanford Bunny model consisting of 70 k tris., and

(a) Full-GI (b) Ours (c) Diff.

Fig. 7. Converged rendering images of our method are similar to the reference image generated by Full-GI, photon mapping with full detailed geometry and photon kd-tree.

original uncompressed mesh and HCCMesh representations are tested. We use a 512-by-512 image resolution and trace one primary ray per pixel for the comparison. Note that both representations can fit in the video memory of GPU thanks to its small data size of the tested model. We have observed that the GPU ray tracer is 8.3 times faster than the CPU version with the uncompressed mesh, while 1.3 times faster with the HCCMesh.

Although using GPU shows higher performance than CPU with both representations of the small model, we have concluded that the GPU is not the ideal computing resource for the HCCMesh, since its improvement is relatively minor, i.e., only 1.3 times. The main reason is that decompression of a HCCMesh requires a large amount of the working set and a high number of conditional branches, both of which are not well supported by current GPU architectures. Furthermore, if the size of a model is bigger than the video memory, data should be continuously transferred from main memory to the video memory, which significantly negates the advantages of using GPUs. This excessive amount of data transfer has been known as one of major bottlenecks of rendering large-scale models [17].

*Comparison with coupled representations.* Several LOD-based approaches [48], [21], [13] are coupled representations that consist both of a hierarchical LOD representation and primitives (i.e., triangles of the original model) that are spatially grouped and assigned to leaves of the hierarchical representation. Although these coupled representations can be more compact than our representation, they were not mainly designed for rendering with heterogeneous computing resources such as CPU and GPU. As a result, they can cause frequent, but unnecessary data transfers between the main memory and the video memory. Departing from this coupled approach, we decouple the original mesh and its LOD representation into the HCCMesh and ASVOs. This decoupling requires additional memory space. For example, we use 89 percent more space over the HCCMesh by having ASVOs for the Boeing model. We found that even though we have such additional memory requirements, it effectively reduces data transfer costs by fitting our volumetric representation, especially the upper ASVO, in the GPU video memory, and thus achieves a high throughput and low response time.

*Comparisons with prior voxel octrees.* Crassin et al. [9] proposed efficient voxel octrees as a volumetric LOD representation, and used the same representation for filtered (i.e., low-frequency effects) global illumination with small models that can fit into the main memory [11]. At a high level there are two main differences between our

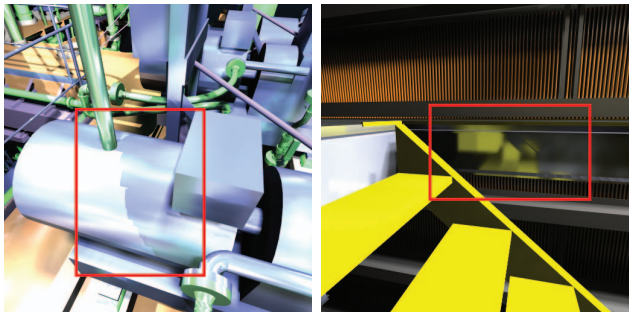
representation and theirs. We use the compact HCCMesh to process C-rays in the CPU side and augment voxel octrees with occlusion bitmaps. As a result, we are able to support high-frequency effects from geometry details better, and thus, test our method with a diverse set of massive models including CAD models that have irregular distribution of geometry. Also, the voxel octrees can be directly applied to our framework for G-rays instead of ASVOs, and it might show similar rendering results if a proper resolution for the prior voxel octree is used. However, it could be less efficient because voxel octrees require more space than our ASVOs, especially for visibility tests; the most detailed representation of the voxel octrees has its own color values unlike occluder bitmaps. Note that we use a higher resolution for the visibility tests by using the compact, occluder bitmaps.

*Comparisons with small models.* Our techniques are mainly designed for handling massive models. Nonetheless, our results indicate that our method can handle small models robustly without much computation overheads in terms of ray processing performance, even when compared with the state-of-the-art global illumination techniques specialized for small models [49]. This is mainly because our voxel representation drastically reduces the computation of global illumination. The technique proposed by Wang et al. [49] processes 5.0 M∼6.9 M rays per second (107 k photon rays, 250∼500 gathering rays for 5 k sample point, and 2 M rays for local illumination per frame) on NVIDIA GTX 280, and showed 1.5 FPS for a kitchen scene containing 21 k triangles. Since our graphics hardware outperforms about 2∼3 times over GTX 280, the performance of Want et al. approach is expected to be 10 M∼20 M rays per second on our test machine. Even though the Sponza model consisting of 66k triangles may have different characteristics to the kitchen model, our method for the Sponza model shows 12.6 M∼20.2 M rays per second.

## 7 Conclusion and Future Work

We have presented various techniques and their integrated progressive rendering framework to achieve a low response time to users and high throughputs for global illumination of massive models. In particular, we proposed to use a decoupled representation consisting of polygonal and volumetric representations, HCCMesh and ASVOs, to reduce expensive transmission costs and achieve high utilizations for the CPU and GPU. We also augmented sparse voxel octrees with occlusion bitmaps to provide higher geometric resolutions from our volumetric representation. We also proposed saliency-based tile ordering within our progressive rendering framework.

*Limitations and future work.* As other prior techniques employing volumetric representations, our method is biased and not even consistent. Also, our volumetric representation spans more space compared to its original polygonal model, causing false-positive intersections and wider shadow regions. In scenes with point light sources and highly glossy materials, our method can generate box-like artifacts even when we use occlusion bitmaps (Fig. 8). This artifact becomes more noticeable when voxels are close to shadow or gathering rays. We tried an approach to detect such cases, but it required too much computation, lowering ray throughputs. We leave this issue as one of our future work; adopting prefiltering [50] can be used to reduce such artifacts. Also, ASVOs may have storage overheads for small models such as Sponza model because the ASVOs do not depend on the number of primitives. Note that these are common drawbacks of voxel-based ray tracing.



(a) Point light sources (b) Highly glossy materials

Fig. 8. These images show artifacts caused by a limited resolution of our volumetric representation.

In our current rendering framework, we manually assigned each type of rays to the CPU and GPU depending on its characteristics. A better approach is to measure ray footprints based on ray differentials [20] and assign rays with small footprints to the CPU using HCCMesh, while process the rest of rays with wider footprints on the GPU with ASVOs. Also, even though our approach provided interactive rendering results within our progressive framework, the workload of CPU and GPU can vary a lot depending on the camera, geometry, and materials. This can result in a low utilization of either the CPU or GPU. To address this issue, we would like to extend our approach to off-load jobs of a busy resource to another resource.

There are many other interesting avenues for future work. Because our framework requires preprocessing, it is not easy to support fully deforming models. It can be extended to support local modifications such as geometry additions or deletions, which are useful operations for CAD designers. These operations can be supported by tracing rays with separate BVHs of modified meshes for immediate response, but with a low rendering throughput. We can then perform various precomputation in a background mode with the modified meshes for a higher rendering throughput later on. Also, the visual quality of our photon mapping can be improved by having a list of photons in each voxel instead of the accumulated photon intensity. The radiance estimation can be done by accessing neighboring voxels and then performing KNN search with photons stored in those voxels. Nonetheless, this improvement still leaves our approach to be biased. Also, our approach aimed to both a high rendering throughput and a low responsive time to users. We would like to design an optimization process considering our two goals and use it as a principle to redesign various components of our progressive rendering framework.

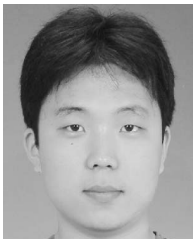
## Acknowledgments

The authors would like to thank anonymous reviewers for their constructive feedbacks. They also thank members of KAIST SGLab for their supports. They thank model donors: the Boeing 777 model for David Kasik at the Boeing Company, the Double Eagle model for NNS, the power plant environment for an anonymous donor,

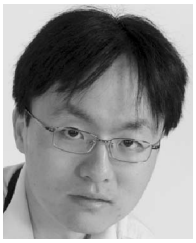
Sponza for Marko Dabrovic, St. Matthew models for Stanford Univ., and the isosurface model for LLNL. This project was supported in part by MCST/KOCCA/CT/R&D 2011, DAPA/ADD (UD110006MD), MEST/NRF/WCU (R31-30007), BK, KMCC, MSRA, ExoBrain, MEST/NRF (2012-0009228).

## REFERENCES

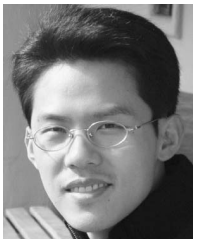
[1] S.-E. Yoon, E. Gobbetti, D. Kasik, and D. Manocha, *Real-Time Massive Model Rendering*. Morgan & Claypool Publisher, 2008.
[2] T.-J. Kim, Y. Byun, Y. Kim, B. Moon, S. Lee, and S.-E. Yoon, "HCCMeshes: Hierarchical-Culling Oriented Compact Meshes," *Computer Graphics Forum,* vol. 29, no. 2, pp. 299-308, 2010.
[3] C. Lauterbach, S.-E. Yoon, M. Tang, and D. Manocha, "ReduceM: Interactive and Memory Efficient Ray Tracing of Large Models," *Computer Graphics Forum,* vol. 27, no. 4, pp. 1313-1321, 2008.
[4] B. Segovia and M. Ernst, "Memory Efficient Ray Tracing with Hierarchical Mesh Quantization," *Proc. Graphics Interface,* pp. 153-160, 2010.
[5] P.H. Christensen, "Point-Based Approximate Color Bleeding," technical report, Pixar Animation Studios, 2008.
[6] J. Kontkanen, E. Tabellion, and R.S. Overbeck, "Coherent Out-of-Core Point-Based Global Illumination," *Computer Graphics Forum,* vol. 30, no. 4, pp. 1353-1360, 2011.
[7] E. Gobbetti and F. Marton, "Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-Sampled Models," *Computer Graphics,* vol. 28, no. 6, pp. 815-826, Dec. 2004.
[8] P.H. Christensen and D. Batali, "An Irradiance Atlas for Global Illumination in Complex Production Scenes," *Proc. Eurographics Symp. Rendering (EGSR),* pp. 133-141, 2004.
[9] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games (I3D),* pp. 15-22, 2009.
[10] S. Laine and T. Karras, "Efficient Sparse Voxel Octrees," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games,* pp. 55-63, 2010.
[11] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive Indirect Illumination Using Voxel Cone Tracing," *Computer Graphics Forum,* vol. 30, no. 7, pp. 1921-1930, 2011.
[12] C. Crassin, "Beyond Programmable Shading: Dynamic Sparse Voxel Octrees for Next-Gen Real-Time Rendering," *Proc. ACM SIGGRAPH Courses,* 2012.
[13] A.T. Afra, "Interactive Ray Tracing of Large Models Using Voxel Hierarchies," *Computer Graphics Forum,* vol. 31, no. 1, pp. 75-88, 2012.
[14] Y.-J. Chiang, J. El-Sana, P. Lindstrom, R. Pajarola, and C.T. Silva, "Out-of-Core Algorithms for Scientific Visualization and Computer Graphics," *Proc. IEEE Visualization Course Notes,* 2003.
[15] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-Oblivious Mesh Layouts," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 886-893, 2005.
[16] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering Complex Scenes with Memory-Coherent Ray Tracing," *Proc. ACM SIGGRAPH,* pp. 101-108, 1997.
[17] B. Budge, T. Bernardin, J.A. Stuart, S. Sengupta, K.I. Joy, and J.D. Owens, "Out-of-Core Data Management for Path Tracing on Hybrid Resources," *Computer Graphics Forum,* vol. 28, no. 2, pp. 385-396, 2009.
[18] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S.W. Nam, and S.-E. Yoon, "Cache-Oblivious Ray Reordering," *ACM Trans. Graphics,* vol. 29, no. 3, pp. 1-10, 2010.
[19] I. Wald, A. Dietrich, and P. Slusallek, "An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models," *Proc. Eurographics (EG) Symp. Rendering,* pp. 82-91, 2004.
[20] H. Igehy, "Tracing Ray Differentials," *Proc. ACM SIGGRAPH,* pp. 179-186, 1999.
[21] S.-E. Yoon, C. Lauterbach, and D. Manocha, "R-LODs: Interactive LOD-Based Ray Tracing of Massive Models," *The Visual Computer,* vol. 22, nos. 9-11, pp. 772-784, 2006.
[22] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D.P. Greenberg, "Lightcuts: A Scalable Approach to Illumination," *Proc. ACM SIGGRAPH,* pp. 1098-1107, 2005.
[23] P. Shirley and R.K. Morley, *Realistic Ray Tracing,* second ed. AK Peters, 2003.
[24] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann, 2004.
[25] E. Veach and L.J. Guibas, "Metropolis Light Transport," *Proc. ACM SIGGRAPH,* pp. 65-76, 1997.
[26] A. Keller, "Instant Radiosity," *Proc. ACM SIGGRAPH,* pp. 49-56, 1997.
[27] H.W. Jensen, *Realistic Image Synthesis Using Photon Mapping.* AK Peters, 2001.
[28] T. Hachisuka, S. Ogaki, and H.W. Jensen, "Progressive Photon Mapping," *Proc. ACM SIGGRAPH Asia,* pp. 1-8, 2008.
[29] T. Hachisuka and H. Jensen, "Stochastic Progressive Photon Mapping," *ACM Trans. Graphics,* vol. 28, no. 5, pp. 1-8, 2009.
[30] T. Hachisuka, W. Jarosz, and H.W. Jensen, "A Progressive Error Estimation Framework for Photon Density Estimation," *Proc. ACM SIGGRAPH Asia,* pp. 144:1-144:12, 2010.
[31] E. Tabellion and A. Lamorlette, "An Approximate Global Illumination System for Computer Generated Films," *ACM Trans. Graphics,* vol. 23, no. 3, pp. 469-476, 2004.
[32] S. Thiedemann, N. Henrich, T. Grosch, and S. Müller, "Voxel-Based Global Illumination," *Proc. Symp. Interactive 3D Graphics and Games,* pp. 103-110, 2011.
[33] T. Ritschel, C. Dachsbacher, T. Grosch, and J. Kautz, "The State of the Art in Interactive Global Illumination," *Computer Graphics Forum,* vol. 31, no. 1, pp. 160-188, 2012.
[34] J.-L. Maillot, L. Carraro, and B. Peroche, "Progressive Ray Tracing," *Proc. Third Eurographics Workshop Rendering,* pp. 9-20, May 1992.
[35] F. Rousselle, C. Knaus, and M. Zwicker, "Adaptive Sampling and Reconstruction Using Greedy Error Minimization," *Proc. ACM SIGGRAPH Asia,* pp. 159:1-159:12, 2011.
[36] M. Bolin and G. Meyer, "A Perceptually Based Adaptive Sampling Algorithm," *Proc. ACM SIGGRAPH,* pp. 299-309, 1998.
[37] R. Achanta, S. Hemami, F. Estrada, and S. Ssstrunk, "Frequency-Tuned Salient Region Detection," *Proc. IEEE Int'l Conf. Computer Vision and Pattern Recognition (CVPR),* 2009.
[38] G. Varadhan and D. Manocha, "Out-of-Core Rendering of Massive Geometric Environments," *Proc. IEEE Visualization,* 2002.
[39] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama, "Digital Photography with Flash and No-Flash Image Pairs," *ACM Trans. Graphics,* vol. 23, no. 3, pp. 664-672, 2004.
[40] H. Sagan, *Space-Filling Curves.* Springer-Verlag, 1994.
[41] L. Itti, C. Koch, and E. Niebur, "A Model of Saliency-Based Visual Attention for Rapid Scene Analysis," *IEEE Trans. Pattern Analysis & Machine Intelligence,* vol. 20, no. 11, pp. 1254-1259, Nov. 1998.
[42] J. Revelles, C. Urea, and M. Lastra, "An Efficient Parametric Algorithm for Octree Traversal," *J. WSCG,* vol. 8, pp. 212-219, 2000.
[43] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, Mar. 2008.
[44] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha, "RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes Using BVHs," *Proc. IEEE Symp. Interactive Ray Tracing,* pp. 39-46, 2006.
[45] G. Ward, "Adaptive Shadow Testing for Ray Tracing," *Proc. Eurographics Workshop Rendering,* 1991.
[46] H. Dammertz, J. Hanika, and A. Keller, "Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays," *Computer Graphics Forum,* vol. 27, no. 4, pp. 1225-1234, 2008.
[47] S.G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: A General Purpose Ray Tracing Engine," *ACM Trans. Graphics,* vol. 29, pp. 66:1-66:13, July 2010.
[48] E. Gobbetti and F. Marton, "Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 878-885, 2005.
[49] R. Wang, R. Wang, K. Zhou, M. Pan, and H. Bao, "An Efficient GPU-Based Approach for Interactive Global Illumination," *ACM Tran. Graphics,* vol. 28, article 91, 2009.
[50] E. Heitz and F. Neyret, "Representing Appearance and Pre-filtering Subpixel Data in Sparse Voxel Octrees," *Proc. Fourth ACM SIGGRAPH/Eurographics Conf. High Performance Graphics,* June 2012.



**Tae-Joon Kim** received the BS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2007. He is currently working toward the PhD degree at KAIST. His research interests include visualization, interactive rendering, and data compression.



**Xin Sun** graduated from Zhejiang University, Hangzhou, China, and received the bachelor's and PhD degrees in computer science in 2002 and 2008, respectively, under the supervision of Professor Jiaoying Shi and Professor Kun Zhou. After that, he joined Internet Graphics Group in Microsoft Research Asia. His research interests lie in global illumination rendering, real-time rendering and general purpose GPU computing.



**Sung-Eui Yoon** received the BS and MS degrees in computer science from Seoul National University in 1999 and 2001, respectively, and the PhD degree in computer science from the University of North Carolina at Chapel Hill in 2005. He is currently an associate professor at Korea Advanced Institute of Science and Technology (KAIST). He was a postdoctoral scholar at Lawrence Livermore National Laboratory. His main research interest is on designing scalable graphics and geometric algorithms. He wrote a monograph on real-time massive model rendering with other three coauthors. Some of his work received a distinguished Paper Award at Pacific Graphics, invitations to IEEE TVCG, an ACM student research competition award, and other domestic research-related awards. He is a senior member of the IEEE.