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

Tae-Joon Kim, Xin Sun, and Sung-Eui Yoon

Abstract—We propose 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 interactive responsiveness. In order to achieve a high rendering throughput we utilize heterogeneous computing resources 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, and 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 performance of 3 M∼20 M rays per second, while limiting response time to users within 15 ms∼67 ms. We also allow dynamic modifications on light, and material setting interactively, while efficiently supporting novel view rendering.

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

1 Introduction

The complexity of polygonal models has been increasing dramatically in both areas of computer-aided-design (CAD) and entertainments. 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 main memory of CPU or GPU is the data transmission time introduced by fetching data from external drives (e.g., HDD or SSD). The excessive data transmission costs hinder high rendering throughput and interactive responsiveness.

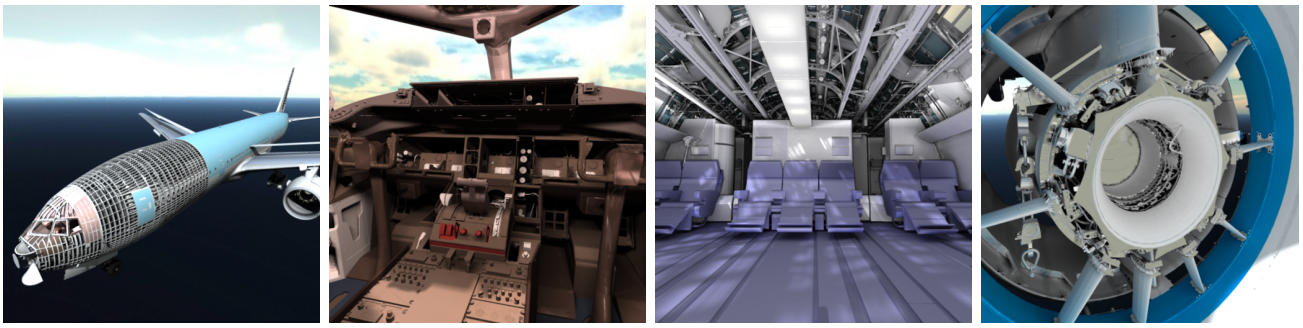
Most prior methods for rendering massive models mainly have been focused on providing basic visual effects such as local illumination and hard shadows [1]. Supporting global illumination requires significantly more computation than local illumination. More importantly, unlike coherent rays such as primary 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 in local illumination. Most prior techniques developed in local illuminations can show improvement, but insufficient performance for interactive global illumination of massive models.

Recent GPUs provide high computational power and thus realizes interactive, high quality global illumination mainly for small scale models that can fit into the video memory. Unfortunately, the video memory is generally more limited than main memory of CPU and it is even more challenging to support global illumination for massive models on GPUs.

In this paper we propose novel techniques enabling interactive rendering of large-scale models consisting of hundreds of millions of primitives by highly utilizing computation 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 representations named T-ReX (Tri-level Representations for eXpress rendering) consists of separate, three different levels-of-details (LoD) for the input model: the original polygonal representation, and coarse and fine volumetric representations (Sec. 4). We use the original, fully detailed geometric representation only at the CPU, while two volumetric representations are used at GPU. Especially, the coarse, volumetric representation is designed

- *Tae-Joon Kim and Sung-Eui Yoon are with Department of Computer Science, KAIST (Korea Advanced Institute of Science and Technology), Daejeon, South Korea.*
  E-mail: tjkim.kaist, sungeui@gmail.com
- *Xin Sun is with Microsoft Research Asia, Beijing, China.*
  E-mail: sunxin@microsoft.com



(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 which take 8∼12 minutes. More importanlty, each rendering frame is provided to users with less than 100 ms latency time, while allowing dynamic changes on camera, light, and material setting.

such that it can fit into the video memory of GPU, while the fine, volumetric representation is stored at main memory of CPU 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. 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 (e.g., primary rays) that generates high-frequency visual effects we use the geometric representation on the CPU side. For all the other rays (e.g., gathering rays) that tend to generate low-frequency visual effects, we use our volumetric representation on the GPU side.

Partitioning various rays of photon mapping to two sets, each of which can be supported well by either one of our representations, 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 available communication bandwidth for asynchronously transmitting necessary portions of the fine volumetric representation to the video memory, and then progressively refine the rendering quality with 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 CPU/GPU 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 minizing data transmission costs, we are able to achieve ray processing performance of 3 M∼20 M rays per second. More importanlty, for various types of models with variying 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 overheads over the state-of-the-art global illumination techniques specialized for small models.

According to our best knowledge, the progressive rendering framework integrated with our proposed techniques is the first system that interactively performs photon mapping for massive models with the ability 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, multi-resolution, etc.

2.1.1 Compact Representations

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

A point-based approach [4], [5], [6] like point clouds decouples illumination data from the geometry, and employs multi-resolution techniques for efficient rendering. The irregular distribution of point samples enables high quality indirect illumination effects, but also leads to heavy computation costs, hindering interactive applications.

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 [7], [8], [9]. It is well suited to GPU architectures thanks to its compact storage, and efficient traversal, and provides plausible rendering quality. Crassin [10] 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 [11] showed interactive color bleeding effects on massive models by using asynchronous voxel loading. Our rendering framework supports photon mapping and can generate more realistic outputs. 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.

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 [12] assuming a particular cache size. Cache-oblivious techniques were shown to improve the cache coherence across different cache sizes [13]. In the field of ray tracing, there are a few techniques that maximize cache utilization by reordering rays [14], [15], [16]. 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. demonstrated interactive visualization of a Boeing model consisting of 366 million triangles by using an out-of-core approach [17], 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 Multi-Resolution

Extensive research efforts have been put into designing various multi-resolution techniques for geometry [18], spatial hierarchy [19], and lighting [20]. Sparse voxel octrees [9] provide a multi-resolution 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 [21], [22].

Unbiased Monte Carlo ray tracing approach (e.g., bidirectional ray tracing [23]) based on the rendering equation is the standard reference 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 [24] and photon mapping [25]. 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 an available memory [26], stochastic rendering effects [27], and robust error estimation with a progressive rendering framework [28]. These techniques can be naturally combined with our method that focuses on handling massive 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 [29]. 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 [30], and photon mapping was extended to be progressive [26].

Various sampling has been extensively studied and can be integrated within a progressive rendering framework. Most sampling techniques are based on variance of the previous samples [31]. In addition, human perception is also taken into account to guide sampling [32]. In this paper we also use a *saliency* metric [33] that can be efficiently evaluated.

2.4 Heterogeneous Computing Resources

Recently computation-intensive applications including global illumination have been accelerated by using multiple heterogeneous resources such as CPUs and GPUs. For the problem of collision detection, which is related to ray tracing, HPCCD [34] divided tasks to either CPU or GPU, according to how well characteristics of tasks suit well to either one of architectures. In the context of global illumination, Budge et al. proposed a generalized data management on CPU/GPU hybrid resources for path tracing [15]. Unlike the previous approaches, we separate data representations for CPU and GPU and minimize expensive data transmission overheads between them.

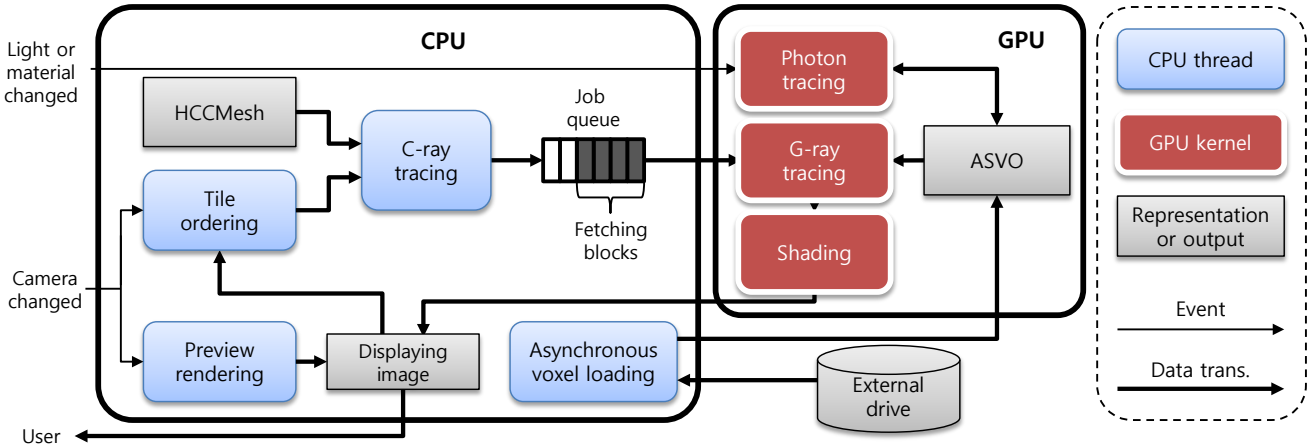


Fig. 2: This figure shows our rendering framework and data transitions between different modules.

3 Overview

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 then process C-rays on CPU with a detailed, but compressed polygonal representation called HCCMesh, while processing G-rays on GPU with our volumetric representation, augmented sparse voxel octree (ASVO) (Fig. 2).

We define C-rays to be primary rays and their secondary rays reflected on perfect specular materials, since they are likely to generate high-frequency rendering effects. All the other rays (e.g. gathering rays and shadow rays) are grouped together into G-rays, even though some of them (e.g. gather rays) produce low-frequency effects and others (e.g. shadow rays) high-frequency effects.

In order to provide high quality results for C-rays, we use detailed, but compressed polygonal models to process those rays. 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., 4 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. Furthermore, we subdivide leaf voxels of sparse voxel octrees and represent geometric information of the subdivided voxels with a compact *occluder bitmap*. As a result, we can provide higher geometric resolutions for rays (e.g., shadow rays) of G-rays that are sensitive to geometric resolutions.

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.

We process rays tile-by-tile for better controlling the response time of our rendering framework. We therefore employ a *Tile ordering* module that computes a tile ordering considering both 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 and then send their 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 CPU and then load a coarse version of our ASVO representation to the video memory of GPU. Once these initial data loading operations are done, our system is ready to provide interactive response 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 progressively better rendering results; we do not send the original geometry to GPU at all.

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 guarantees that users can receive a new rendering result interactively, even when processing C-rays and G-rays in CPU and GPU takes much larger 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, we use a HCCMesh representation [2]. HCCMesh is a compact mesh representation that

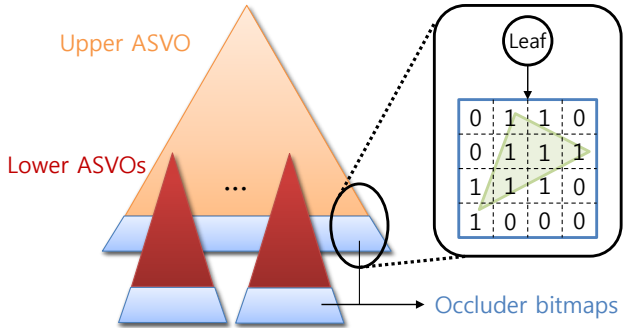
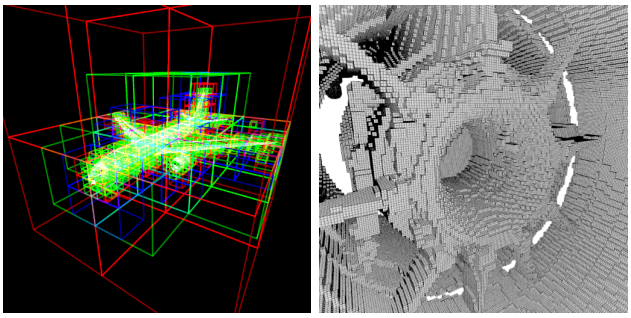


Fig. 3: **Augmented Sparse Voxel Octree (ASVO)**: Our ASVO consists of upper and lower ASVOs, each of which is combined with occluder bitmaps for every leaf node. The occluder bitmap has bit values, each of which indicates whether its corresponding sub-voxel overlaps with the original geometry.



(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.

tightly integrates an input triangular mesh and its BVH (Bounding Volume Hierarchy) together. It reduces the size of the BVH by using connectivity templates of a hierarchy, and compactly encodes of bounding volumes (BVs) based on vertices of the mesh. Furthermore, it provides random access on the compressed mesh and its BVH.

HCCMesh has in-core and out-of-core representations; i-HCCMesh and o-HCCMesh, respectively. i-HCCMesh supports high-performance decompression, but has relatively low compression ratio (about 7:1). On the other hand, o-HCCMesh provides high compression ratio (about 20:1), but low decompression performance because of its more complex encoding scheme. In our current approach, we use only i-HCCMesh, since i-HCCMeshes of all the tested models fit into the available memory of our tested system; we can use o-HCCMesh additionally for even bigger models.

4.2 Augmented Sparse Voxel Octree (ASVO)

We use ASVO for efficient handling of G-rays that produce indirect illuminations in the GPU side. ASVO serves both as an approximated geometry for the input model and a volumetric representation of photons for indirect illumination. ASVO consists of three different components that provide increasing higher resolutions: upper and lower ASVOs, and occluder bitmaps (Fig. 3). We pre-load all the data of the upper sparse voxel octree and its corresponding occluder bitmaps to the video memory of GPU and thus avoid their data transmission overhead between CPU and GPU at runtime. On the other hand, necessary portions of lower ASVOs (and their occluer 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. As a result, the upper ASVO is resident on 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 three dimensional sizes of the upper ASVO (and thus its voxels) to have the equal size for efficient ray tracing. Because of this constraint the bounding cube of the ASVO is bigger or equal to the bounding box of the model; we use the word of bounding cube to highlight the regularity of dimensions of ASVO and its voxels as shown in Fig. 4(a). The constraint on three dimensional sizes of the ASVO would generate many empty octree nodes. We store octrees as sparse octrees [9] to effectively represent such empty octree nodes.

The bounding cube of the upper ASVO is recursively subdivided in the middle along each dimension to generate a sparse octree. All the non-empty nodes are stored in an array by the breath-first-search order. For each non-empty leaf node, we compute and record representative normal and material for the corresponding voxel. The normal and material are computed using triangles weighted by its intersected area with the voxel. These representative material and normal in each leaf node serves as an Level-of-Detail (LOD) representation to geometry contained in each voxel, and are used for efficiently tracing multi-bounced 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 theirs; ASVO is totally a decoupled representation from the mesh. In addition we bake photons by accumulating their information at leaf voxels of the upper and lower ASVOs for indirect illumination at runtime in a similar manner to [9].

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

In order 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 sub-tree. We therefore prefetch

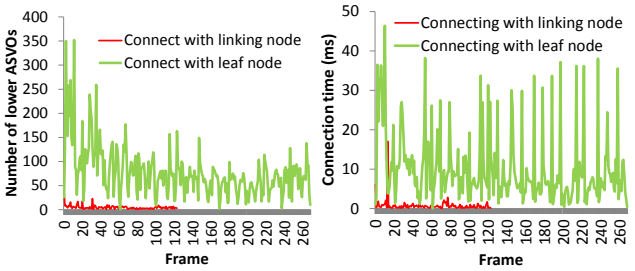


Fig. 5: The left figure shows a number of loaded lower ASVOs per each frame, and the right figure shows synchronization time used for connecting lower ASVOs to the upper ASVO.

Fig. 6: 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.

its corresponding lower ASVO asynchronously [35] 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 for unloading less-frequenly used lower ASVOs.

Benefits of having lower ASVOs for internal nodes instead of leaf nodes come from the fact that the number of update operations drastically reduces and thus I/O throughputs improve. This is because the number of internal nodes is typically much smaller than the number of leaf nodes given the octree representation, and accordingly the granularity of lower ASVOs increases. In practice we choose internal nodes that have three depth lower than leaf nodes for linking nodes and thus we reduce up to 8^3 update operations.

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

In order to verify benefits of connecting lower ASVOs with the linking nodes, we measured a number of loaded lower ASVOs and synchronization time in each frame between two methods of connecting lower ASVOs to linking and leaf nodes (Fig. 5). The average cost with the chosen method is 0.16 ms, while the cost of creating lower ASVOs for each leaf node is 4.4 ms, which is about 28 times slower than our method. This is mainly because of the drastically reduced number of update operations.

Since we create lower ASVOs for linking nodes, there are overlapping nodes between upper and lower ASVOs. When we allowed three depth overlaps between them the memory overhead of the data redundancy is 0.15% of the total size of the ASVO. Since this overhead is negligible, we do not adopt any compression techniques to remove or reduce this overhead. We denote r_l an effective subdivision level for leaf voxels of lower ASVOs that excludes the overlapping factor bewteen lower and upper ASVOs. In practice r_l is set to 2∼4. As a result, the total resolution considering the upper and lower ASVOs is $(r_u * r_l)^3$.

Occluder bitmaps. The upper and lower ASVOs provide enough resolutions for various indirect illuminations (e.g., color bleeding) to our tested models, while providing 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 sub-voxels, and check only whether each sub-voxel is empty or not. We use this binary information of each sub-voxel for providing higher geometry information for shadow rays (Fig. 6). In practice we set r_o to be 4, and thus we require $4^3 = 64$ bits, which require 8 bytes for each node.

In summary our sparse octrees in ASVOs provide up to $(r_u * r_l)^3$ resolutions to 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 [25]; 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 [9] of both upper and lower ASVOs. Once photon tracing is done each leaf node of ASVOs maintains a radiance value of all the accumulated photons in its corresponding voxel.

To quickly respond to user's modifications on the camera, lighting, and materials, we compute a low-resolution preview that contains only direct illumination of the model in the preview rendering module. Especially when a user quickly navigates the model, the user sees only the preview. The resolution of the preview is computed such that its

computation time is less than a user-specified threshold for rendering a frame, t_{max}. t_{max} is set to be 100 ms, which is the time we consider as a longest response time to users. Since we need to give time for other modules, we set that the preview is done within one third of t_{max}, which is about 30 ms. By providing the preview in this manner, we can avoid excessively long response time to any user input. 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 (Fig. 8). This is also demonstrated in the accompanying video.

Finally, shading is done with photon mapping, and we perform the well-known G-Buffer based filtering [36] to reduce a 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 a ASVO, its intensity and incident direction are accumulated to the voxel. We compute an outgoing direction of the intersected photon based on the normal and material information stored in the voxel. To decide whether a photon hits a 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. 6). The traversal scheme of rays with occlusion bitmaps are same to that of ASVOs, since they are constructed based on regular grids.

When a user changes settings of lights or materials, we initialize accumulated photon information stored in all the ASVOs and then re-start 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 and store them in additional, temporary ASVOs. 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. We then get radiance from both the current ASVO and temporary ASVOs, but with different weights. Initially we give a higher weight to the current ASVOs, but to the temporary ASVOs, as the photon tracing module generates more photons; we finish the photon tracing module once it generates photons with a user-defined target number (e.g. 5 M photons for each light). This idea is in the same spirit to *geomorphing* widely adopted in rasterization based multi-resolution rendering [37]. Once all the photons are generated, the temporary ASVOs 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. 1(b).)

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 ASVOs. At runtime we need to perform G-ray tracing with the backed ASVOs and shading in the GPU side.

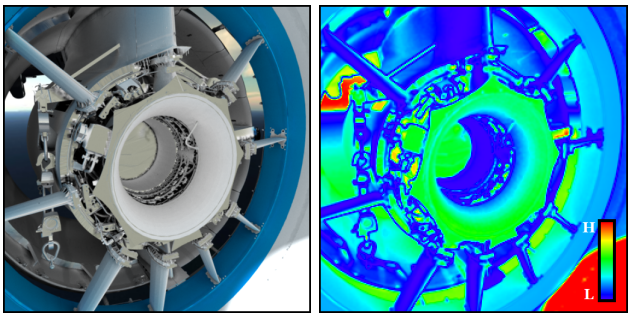


Fig. 7: The right image visualizes a saliency map of the left image. Red colors indicate high saliency, while blue colors low saliency.

5.2 Tile-based Rendering

We compute rendering results in a tile-based manner for better controlling the response time to users. We divide an image screen to tiles and process all the pixels stored in a tile in a parallel manner. We set each tile to have less than 100 pixels that can be processed in a parallel manner by SIMD based packet tracing in CPU and GPU. Furthermore we also 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 t_{max} for interactivity. As a result, it is possible to process only a portion of tiles within a frame. If so, we process other tiles in its subsequent frames. As 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 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 an ordering of processing tiles. The most common ordering methods for tiles include row-by-row or z-curve [38]. Z-curve is usually recommended for 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.

In order to accommodate the users' preference, we propose a salience-based tile ordering. We estimate the users' preference based 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 preview image computed with the current viewpoint for the estimation.

We adopt a saliency metric proposed by Achanta et al. [33] because of its simplicity and efficiency. For each tile we evaluate the saliency metric for each pixel of the preview image, and then compute an average saliency value for each tile. We then sort tiles based on its saliency values and process them sequentially; higher saliency values indicate more importance regions (Fig. 7).

We have tested with different tile ordering 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% difference). As a result, we employ saliency-based tile ordering for our approach, since it achieves 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 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, 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 a viewer.

The granularity of the fetching block is controllable based on a threshold by users. 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. More detailed analysis is available in Sec. 6.2. We use the user-specified granularity of fetching blocks to respect his or her preference on the rendering throughputs. Nonetheless, the response time of the current frame is larger than $t_{max}(= 100\ ms)$ 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 [39] to trace both kinds of rays with both ASVOs and occlution bitmaps in a similar manner that we trace photon in the photon tracing module. In order to maximize utilization of GPU, we process a bundle of rays simultaneously by considering the SIMT architecture of the modern GPUs [40]; 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 executes one common instruction at a time, the utilization is lowered when the threads have data-dependent conditional branches. To minimize such serializations, we perform a cache-oblivious ray reordering for rays [16]. In particular, we sort the rays based on their ray directions and then assign rays with similar directions to a warp.

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

TABLE 1: This table shows model complexity, size of each representation and resolution of voxels (r_u and r_l) for benchmark models. HCCM. stands for the HCCMesh. u-ASVO and l-ASVO indicate upper and lower ASVOs, respectively.

5.4 Asynchronous Voxel Loading

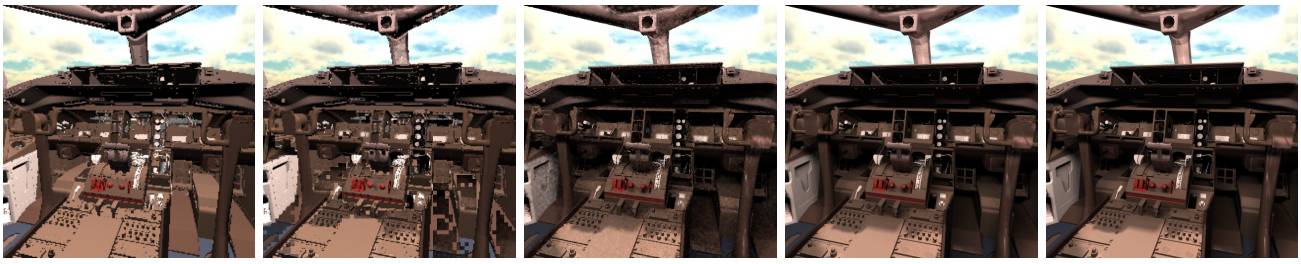
When a ray traverses a linking node of the upper ASVO, we check whether its corresponding lower ASVO is loaded or not in the video memory. When the lower ASVO is loaded, it indicates that it is already linked to the linking node of the upper ASVO. As a result, we can keep traverse into the corresponding 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 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 Sec. 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 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 *a ASVO buffer* that permanently holds the upper ASVO, while the rest of the video memory is reserved for caching lower ASVOs. Specifically, 15% of the available video memory, which is 300 MB, is set for the ASVO buffer; a range of 10% to 50% 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 Boeing 777 model (Fig. 1) consisting of 366 M triangles. The model takes 15.6 GB 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



(a) Preview$< 30\ ms$ (b) 1 frame$< 60\ ms$ (c) Compl. I.$< 0.2\ s$ (d) 800 frames$< 10\ s$ (e) Converged I.

Fig. 8: These images show progressive results after finishing preview rendering, a fetching block to generate a frame, all the tiles for the complete image (Compl. I.), and 800 frames. The converged image is computed after 40 k frames.

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 4, 8, and 2 area lights, respectively. The CAD models usually have irregular distributions of geometry and drastically varying triangle sizes. Other types of benchmark models include St. Matthew model (372 M triangles) as a scanned model, and iso-surface model (469 M triangles) extracted from a scientific simulation (Table 3) with 1 and 2 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 achieve 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 two and use the maximume resolution value given the 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 a required resolution for a model. Nonetheless, four times higher resolution over r_u for r_l shows a good balance in terms of quality and performance. 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 three. We use the Phong illumination model for BRDF. To process primary rays efficiently in the C-ray tracing module we adopt packet ray tracing [41]. 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 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 simply 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 et al. [42].

6.2 Performance

We show performance achieved 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 a 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 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). Instead, we have measured 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 a *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 non-progressive 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. [17]. We use a 512 by 512 image resolution 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 [25].

As shown in Table 2, our approach shows the response time of 25.9 ms$\sim$36.9 ms across different views when

			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

TABLE 2: This table shows rendering performance including response time, **Resp. T**, and ray processing throughput measured in **M rays/s** at different views shown in Fig. 1. We also report complete image time, **CIT**, for comparison with other work. Time is reported in ms unit.

we use $n_s = n_g = 2$. These results directly indicate that users can get a feedback within this response time, even when they 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 7∼9 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 ms∼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. Progressive results for the cockpit viewpoint are shown in Fig. 8.

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

Fetching block granularity. Ray processing throughput and responsiveness of our system depend heavily on the fetching block granularity. In order 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. 9). We tested with the Boeing 777 model at the overview and cockpit, and all the other parameters are same to ones used for prior experiments. We observed the natural trade-off between the ray processing throughput and response time, as we increase the fetching block size. We found that using 128 to 512 fetching block sizes

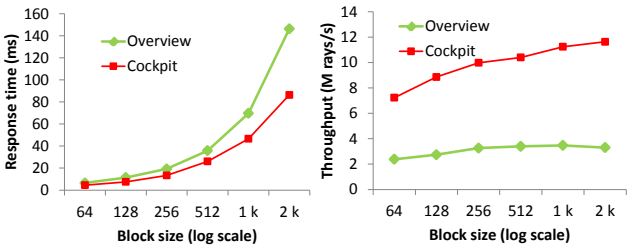


Fig. 9: These graphs show response time and ray processing throughputs as a function of the fetching block sizes.

is a good compromise in terms of both throughput and responsiveness. For the rest of tests, we use 512 block sizes as the default fetching block size. For St. Matthew scene the size is, however, automatically reduced to 128, to make the response time within t_{max} as discussed in Sec. 5.2.

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 iso-surface models show different results over CAD models. In these models, especially the St. Matthew model, the main computational bottleneck is on operations performed at 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 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, 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 [43]. 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 (Fig. 10).

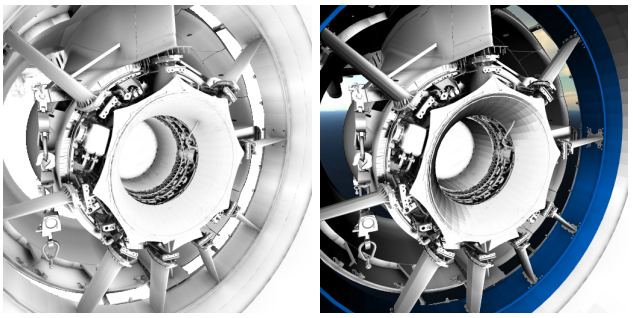
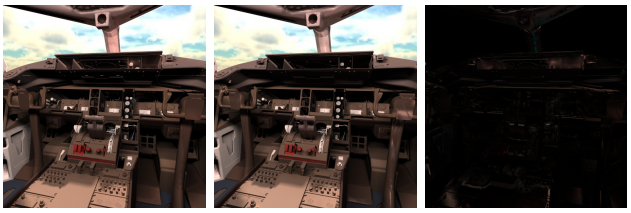


Fig. 10: Ambient occlusion (AO) result on the left and the final image (on the right) combining AO and direct illumination.



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

Fig. 11: 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.

6.3 Comparisons

To highlight 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, we do not use ASVOs and 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.

In order 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, our method running on the CPU, 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 very higher than others, and hence KNN search becomes takes much larger 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. 11); 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, and makes our method a biased technique.

Comparison with coupled representations. Several LOD based approaches [11], [19] 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 main memory and video memory, which are one of major bottlenecks of rendering large-scale models [15]. Departing from this coupled approach, we decouple the original mesh and its LOD representation into HCCMesh and ASVOs. This decoupling requires additional memory space. For example we use 89% more space over 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. [8] proposed efficient voxel octrees as an volumetric LOD representation, and used the same representation for filtered (i.e. low-frequency effects) global illumination with small models that can fit into main memory [9]. 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 better and thus test our method with a diverse set of massive models including CAD models that have irregular distribution of geometry. In addition, we minimize the expensive data transmission costs for effective handling massive models on heterogeneous resources by decoupling the mesh representation and its volumetric representations, followed by having the upper (coarse, but small) and lower (fine, but large) augmented voxel octrees.

Comparisons with small models. Our techniques are mainly designed for handling massive models. Nonetheless our results indicate that our method can handle small

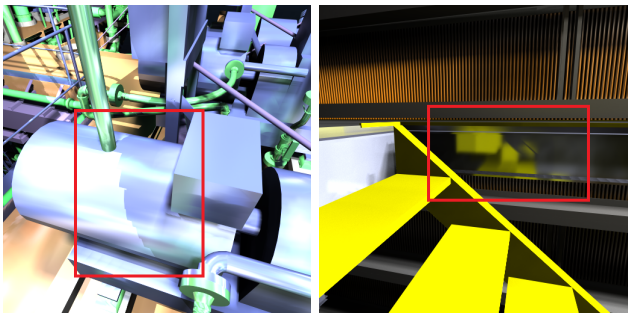
models robustly without much computation overheads in terms of ray processing performance, even when compared with the state-of-the-art global illumination techniques specialized for small models [44]. This is mainly because our voxel representation drastically reduces the computation of global illumination. The technique proposed by Wang et al. [44] 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's 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 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 CPU and GPU. We also augmented ASVOs 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 bias and not even consistent. Also, our volumetric representation spans more spaces 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. 12). This artifact becomes more noticeable when voxels are closed to shadow or gathering rays. We tried an approach to detect such cases, but it required too much computations, lowering ray throughputs. We leave this issue as one of our future work. Also, ASVOs may have storage overheads for small models such as Sponza model because the ASVOs do not depend on number of primitives. Note that these are common drawbacks of voxel based ray tracing.

In our current rendering framework we manually assigned each type of rays to CPU and GPU depending on its characteristics. A better approach is to measure ray footprints based on ray differentials [18] and assign rays with small footprints to CPU using HCCMesh, while the rest of rays with wider footprints are processed on 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 camera, geometry, and materials. This can result in a low utilization of either 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.



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

Fig. 12: These images show artifacts caused by a limited resolution of our volumetric representation.

There are many other interesting avenues for future work. Our method can be extended to adopt progressive photon mapping [45] for better quality and better handling of dynamic changes. Also, our approach aimed to both a high rendering throughput and low responsive time to users. We would like to design an optimization process considering our two goals and use it as a principle to re-design various components of our progressive rendering framework.

Acknowledgments

Omitted for the review.

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 (Eurographics)*, 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," *Comput. Graph. Forum (EGSR)*, vol. 27, no. 4, pp. 1313–1321, 2008.
[4] P. H. Christensen, "Point-based approximate color bleeding," Pixar Animation Studios, Tech. Rep., 2008.
[5] J. Kontkanen, E. Tabellion, and R. S. Overbeck, "Coherent out-of-core point-based global illumination," *Comput. Graph. Forum*, vol. 30, no. 4, pp. 1353–1360, 2011.
[6] J. Pantaleoni, L. Fascione, M. Hill, and T. Aila, "Pantaray: fast ray-traced occlusion caching of massive scenes," in *ACM SIGGRAPH*, 2010, pp. 37:1–37:10.
[7] E. Gobbetti and F. Marton, "Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 878–885, 2005.
[8] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering," in *ACM SIGGRAPH Symp.on Interactive 3D Graph. and Games (I3D)*, 2009, pp. 15–22.
[9] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing," *Comput. Graph. Forum (Pacific Graphics)*, vol. 30, no. 7, 2011.
[10] C. Crassin, "Beyond programmable shading: Dynamic sparse voxel octrees for next-gen real-time rendering," in *ACM SIGGRAPH 2012 courses*, 2012.
[11] A. T. Afra, "Interactive ray tracing of large models using voxel hierarchies," *Comput. Graph. Forum*, vol. 31, no. 1, pp. 75–88, 2012.

[12] Y.-J. Chiang, J. El-Sana, P. Lindstrom, R. Pajarola, and C. T. Silva, "Out-of-core algorithms for scientific visualization and computer graphics," *IEEE Visualization 2003 Course Notes*, 2003.

[13] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-Oblivious Mesh Layouts," *ACM Transactions on Graphics (SIGGRAPH)*, vol. 24, no. 3, pp. 886–893, 2005.

[14] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering complex scenes with memory-coherent ray tracing," in *ACM SIGGRAPH*, 1997, pp. 101–108.

[15] 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," *Comput. Graph. Forum (EG)*, vol. 28, no. 2, pp. 385–396, 2009.

[16] 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. Graph.*, vol. 29, no. 3, pp. 1–10, 2010.

[17] I. Wald, A. Dietrich, and P. Slusallek, "An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models," in *EG Symp. on Rendering*, 2004, pp. 82–91.

[18] H. Igehy, "Tracing ray differentials," in *ACM SIGGRAPH*, 1999, pp. 179–186.

[19] S.-E. Yoon, C. Lauterbach, and D. Manocha, "R-LODs: Interactive LOD-based Ray Tracing of Massive Models," *The Visual Computer (Pacific Graphics)*, vol. 22, no. 9–11, pp. 772–784, 2006.

[20] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg, "Lightcuts: a scalable approach to illumination," in *ACM SIGGRAPH*, 2005, pp. 1098–1107.

[21] P. Shirley and R. K. Morley, *Realistic Ray Tracing*, 2nd ed. AK Peters, 2003.

[22] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 2004.

[23] E. Veach and L. J. Guibas, "Metropolis light transport," in *ACM SIGGRAPH*, 1997, pp. 65–76.

[24] A. Keller, "Instant radiosity," in *ACM SIGGRAPH*, 1997, pp. 49–56.

[25] H. W. Jensen, *Realistic image synthesis using photon mapping*. Natick, MA, USA: A. K. Peters, Ltd., 2001.

[26] T. Hachisuka, S. Ogaki, and H. W. Jensen, "Progressive photon mapping," in *ACM SIGGRAPH Asia*, 2008, pp. 1–8.

[27] T. Hachisuka and H. Jensen, "Stochastic progressive photon mapping," *ACM Trans. Graph. (TOG)*, vol. 28, no. 5, pp. 1–8, 2009.

[28] T. Hachisuka, W. Jarosz, and H. W. Jensen, "A progressive error estimation framework for photon density estimation," in *ACM SIGGRAPH Asia*, 2010, pp. 144:1–144:12.

[29] T. Ritschel, C. Dachsbacher, T. Grosch, and J. Kautz, "The state of the art in interactive global illumination," *Comput. Graph. Forum*, vol. 31, no. 1, pp. 160–188, 2012.

[30] J.-L. Maillot, L. Carraro, and B. Peroche, "Progressive ray tracing," *Third Eurographics Workshop on Rendering*, pp. 9–20, May 1992.

[31] F. Rousselle, C. Knaus, and M. Zwicker, "Adaptive sampling and reconstruction using greedy error minimization," in *SIGGRAPH Asia*, 2011, pp. 159:1–159:12.

[32] M. Bolin and G. Meyer, "A perceptually based adaptive sampling algorithm," in *ACM SIGGRAPH*, 1998, pp. 299–309.

[33] R. Achanta, S. Hemami, F. Estrada, and S. Ssstrunk, "Frequency-tuned Salient Region Detection," in *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, 2009.

[34] D. Kim, J.-P. Heo, J. Huh, J. Kim, and S.-E. Yoon, "HPCCD: Hybrid parallel continuous collision detection," *Comput. Graph. Forum (Pacific Graphics)*, vol. 28, no. 7, 2009.

[35] G. Varadhan and D. Manocha, "Out-of-core rendering of massive geometric environments," in *IEEE Visualization*, 2002.

[36] H. Dammertz, D. Sewtz, J. Hanika, and H. P. A. Lensch, "Edge-avoiding A-Trous wavelet transform for fast global illumination filtering," in *High Performance Graphics*, 2010, pp. 67–75.

[37] H. Hoppe, "View-dependent refinement of progressive meshes," in *ACM Trans. Graph.* ACM SIGGRAPH, Aug. 1997, pp. 189–198.

[38] H. Sagan, *Space-Filling Curves*. Springer-Verlag, 1994.

[39] J. Revelles, C. Urea, and M. Lastra, "An efficient parametric algorithm for octree traversal," in *Journal of WSCG*, 2000, pp. 212–219.

[40] 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.

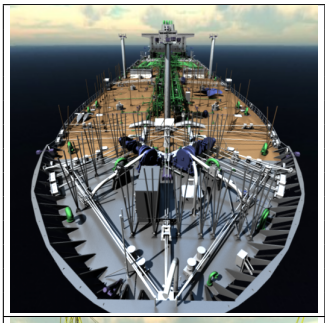
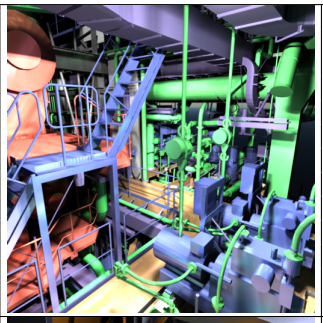
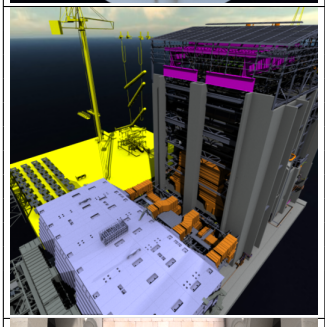
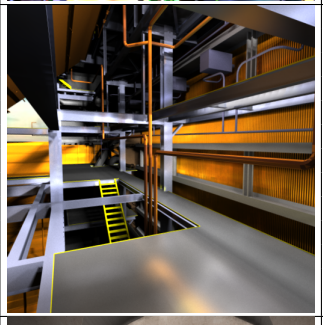
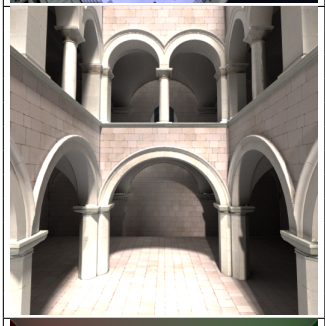
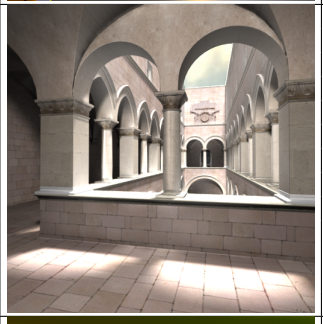
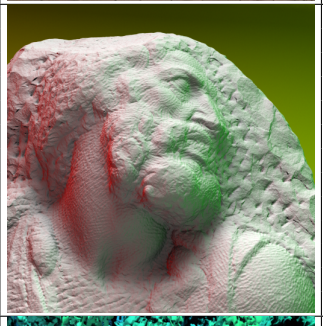
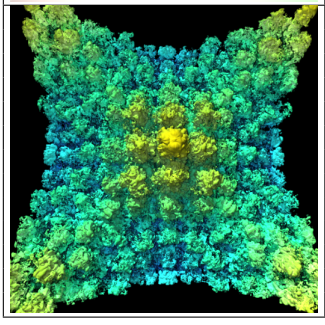
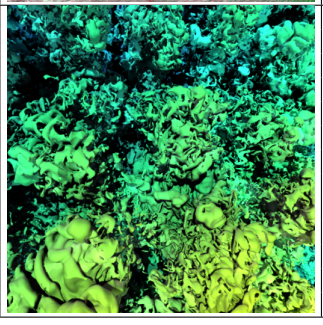
[41] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha, "RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs," in *IEEE Symp. on Interactive Ray Tracing*, 2006, pp. 39–46.

[42] G. Ward, "Adaptive shadow testing for ray tracing," in *Eurographics Workshop on Rendering*, 1991.

[43] 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.

[44] R. Wang, R. Wang, K. Zhou, M. Pan, and H. Bao, "An efficient gpu-based approach for interactive global illumination," *ACM Tran. on Graphics (SIGGRAPH)*, 2009.

[45] T. Hachisuka, S. Ogaki, and H. W. Jensen, "Progressive photon mapping," *ACM Trans. Graph.*, vol. 27, pp. 130:1–130:8, 2008.

View1	View2	
		Double Eagle Tanker (82 M triangles)
		Power plant (13 M triangles)
		Sponza (66 k triangles)
		St. Matthew (372 M triangles)
		Iso-surface (469 M triangles)

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

TABLE 3: Rendering performance with our benchmark models. **CPU T** and **GPU T** show time spent only on CPU and GPU for a complete image time, **CIT**, respectively.