GPU-based Out-of-Core Many-Lights Rendering

Rui Wang Yuchi Huo Yazhen Yuan Kun Zhou Wei Hua Hujun Bao*

State Key Lab of CAD&CG, Zhejiang University

Figure 1: *Example scenes rendered using our approach on an NVIDIA GTX 680 GPU with 2GB of memory. The left image is a museum scene, which consists of 117.1 million triangles and 32.4 million lights. The total storage sizes of geometry and lights are 14.3 GB and 3.75 GB respectively. The middle image shows an airport scene with two Boeing 777 models that has total 669.3 million (46.3 GB) triangles and 18 million (2.1 GB) lights. The right image is a carnival scene. There are 17.1 million (1.76 GB) triangles and 256 (29.6 GB) million lights. Our method takes 3m55s, 10m15s and 1m22s to shade the museum, the airport and the carnival scenes respectively, and requires an additional 1m20s, 7m25s and 1m14s to build acceleration structures on these lights and geometry.*

Abstract

In this paper, we present a GPU-based out-of-core rendering approach under the many-lights rendering framework. Many-lights rendering is an efficient and scalable rendering framework for a large number of lights. But when the data sizes of lights and geometry are both beyond the in-core memory storage size, the data management of these two out-of-core data becomes critical. In our approach, we formulate such a data management as a graph traversal optimization problem that first builds out-of-core lights and geometry data into a graph, and then guides shading computations by finding a shortest path to visit all vertices in the graph. Based on the proposed data management, we develop a GPU-based out-of-GPU-core rendering algorithm that manages data between the CPU host memory and the GPU device memory. Two main steps are taken in the algorithm: the out-of-core data preparation to pack data into optimal data layouts for the many-lights rendering, and the out-of-core shading using graph-based data management. We demonstrate our algorithm on scenes with out-of-core detailed geometry and out-of-core lights. Results show that our approach generates complex global illumination effects with increased data access coherence and has one order of magnitude performance gain over the CPU-based approach.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Rendering;

Keywords: out-of-core, global illumination, many-lights, GPU

Links: DL PDF WEB

*e-mail:bao@cad.zju.edu.cn

1 Introduction

Global illumination effects, such as soft shadows and interreflections, greatly affect the quality of computer synthesized images. These effects provide visual cues that enhance realism. Over the past three decades, many methods have been developed to achieve realistic, high-fidelity renderings: radiosity, ray-tracing, and many-lights approaches are a few. When applying these methods to extremely large scenes that do not fit in memory, the data management strategy arises as a critical problem. When these rendering methods run on modern graphics hardware, the problem is worse due to the limit of onboard GPU memory, and the data transfer overhead from the CPU to the GPU.

In this paper, we focus on extending the many-lights rendering framework to handle massive scenes with out-of-core geometry and complex lighting. In this framework, scenes with diffuse and low-gloss materials can be rendered by approximating direct and indirect illumination from a large number of virtual point lights (VPLs) [Keller 1997]. By viewing the integration of these lights as surface sample-light interactions of the light transport matrix, the rendering is formulated into a matrix sampling problem [Walter et al. 2005; Walter et al. 2006; Hašan et al. 2007], where elements of the matrix are first sampled to obtain representative lights and then these representative lights are integrated to samples (pixels). These two steps require different kinds of data: lights and geometry. At first, samples and lights are utilized to obtain representative lights and then geometry data are required in evaluating visibilities. Such a data access pattern is different from those in previous out-of-core rendering approaches, e.g., visualization or ray-tracing, which usually only need to manage out-of-core geometry data. When both of lights and geometry become too large to be stored in the memory, the data management of these two kinds of data in many-lights rendering may bring conflicts. Our results show that a simple or naive data management in out-of-core many-lights rendering will dominate the rendering time and result in one magnitude slower performance. Thus, a new and dedicated data management is necessitated for out-of-core many-lights rendering to not only handle the geometry but also lights .

To reduce the overall I/O overhead, we formulate the data man-

agement in out-of-core many-lights rendering as a graph traversal problem. First, the light transport matrix is partitioned into submatrices and geometry data is packed into blocks, each of which can be loaded into in-core memory at once. Then, we define graph vertices by pairing submatrices with their potentially intersected geometry blocks. The edge weight from one graph vertex to another is defined as the I/O cost to replace the corresponding in-core data. Therefore, the optimal loading order of lights and geometry can be derived from a specific route optimization on the graph that finds the shortest path to visit all vertices in the graph once, naming the traveling salesman problem (TSP)[1]. While TSP is a NP-hard problem, local or approximation algorithms are applied in this paper to guide the graph traversal.

Based on such a graph-based data management, we present an out-of-core rendering algorithm. It consists of two main steps. First, we solve for the optimal data layout. Second, we perform out-of-core shading using graph-based data management. To validate the algorithm and take advantage of GPU capabilities, we parallelize these computations and implement a GPU-based out-of-GPU-core solution that manages data between the CPU host memory and the GPU device memory. Unless mentioned otherwise, out-of-core in our context means out of device memory. Results demonstrate that our approach is capable of generating complex global illumination effects in scenes with out-of-core geometry and complex lighting, and of reducing render times and increasing data access coherence. The main contributions of this work are as follows:

- We formulate the out-of-core data management in many-lights rendering into a graph traversal problem, and propose different route optimization methods to obtain optimal rendering performance.
- We propose specific matrix adaptive partition and mesh packing methods to obtain optimal data layouts for the out-of-core many-lights rendering problem.
- We present the first GPU-based out-of-core rendering approach for the many-lights rendering problem and achieve high performance.

2 Related work

Many-Lights Problem and Virtual Lights Approximation. Instant radiosity [Keller 1997] generates a number of virtual point lights (VPLs) from light sources. It replaces the computation of indirect diffuse illumination by direct diffuse illumination from these virtual point lights. This basic idea was then formulated as the many-lights problem and inspired a series of works. Walter et al. [2005] proposed the Lightcuts method that uses a hierarchy on lights to reduce the pixel-light computation cost from linear to sublinear. This was extended to multidimensional Lightcuts [Walter et al. 2006] for computing high dimensional rendering integrations, such as volume scattering, depth of field or motion blur. The latest work was to apply such a scalable framework to capture light transports in bidirectional ray paths [Walter et al. 2012]. An alternative formulation of the many-lights problem is the matrix representation, where each row represents an individual sample shaded by each of lights, while each column represents each of samples lit by an individual light [Hašan et al. 2007]. The final image is computed by summing each row in the matrix. Based on the observation that the matrix is low rank, Hasan et al. [2007] presented a matrix sampling method that samples a small number of rows and columns from the full matrix and uses these sparse samples to reconstruct the final image. This was then improved by using matrix slice sampling [Ou and Pellacini 2011]. The point-based representation of lights also appeals to the film industry. The point-based global illumination method (PBGI) [Christensen 2008] and its out-of-core version [Kontkanen et al. 2011] have been used in film production. Besides these offline rendering approaches, Ritschel et al. [2009] computed global illumination by rasterizing point-based lights on thousands of tiny micro-buffers and achieved interactive rendering rates. Recently, these point-based approximations have been extended to lines. Virtual ray lights (VRL) [Novák et al. 2012] is an efficient rendering method for participating media. For more information of many-light rendering, we refer readers to a recent survey [Carsten Dachsbacher 2013].

In this paper, our motivation is to handle massive scenes with complex lighting efficiently in the many-lights framework, where neither lights nor geometry data fit inside in-core memory. Such a motivation bears some similarity with the out-of-core PBGI [Kontkanen et al. 2011]. However, the surfels approximation and inaccurate visibility tests used in PBGI ease the challenges in data management and have less flexibility to tradeoff between quality and performance. To our knowledge, our work is the first one to address optimal data management both on out-of-core VPLs and geometry under the many-lights rendering framework and solve it in an error-driven fashion.

Out-of-core Rendering. Rendering massive models has long been a very challenging problem. Data management is the very first issue in handling massive models, where the out-of-core memory access can become a serious bottleneck in terms of rendering speed. A variety of out-of-core algorithms [Vitter 2001] have been developed to handle general computational problems by minimizing the I/O overhead. For more specific applications, e.g. visualization and rendering, methods were proposed to utilize more knowledge of scenes and computational patterns. Dietrich et al. [2007] and Gobbetti et al. [2008] provided good overviews and surveys on out-of-core rendering techniques and strategies. Frank and Kaufman [2009] proposed an out-of-core volume visualization method that also uses the graph to manage the out-of-core workload. However, their problem was formulated and solved differently in different applications with ours.

For out-of-core global illumination rendering, the computations for global lighting effects, such as shadows and inter-reflections, aggravate the challenges of data management. To efficiently compute these effects, out-of-core path tracing has attracted much attention. The main challenge in data management of methods based on path tracing is that hit points of paths are unknown before the tracing. This requires the data management to be based on some prediction heuristics. However, predictions are not always correct. This motivates many approaches to improve cache coherence so as to improve the performance. Beam [Heckbert and Hanrahan 1984] or packet ray tracing methods [Wald et al. 2001] exploit the coherence of rays. Ray reordering approaches reduce memory bandwidth by improving the ray traversal orders [Aila and Laine 2009], such as Z-curves, scheduling grids or coarse ray sampling. Instead of improving data coherence from rays, alternative methods [Yoon and Lindstrom 2007] have been developed to compact geometry or form acceleration hierarchies in special patterns in order to reduce the I/O overhead. Compared with these methods, our method addresses a great challenge of two kinds of data: lights and geometry. Furthermore, in many-light rendering, because all lights positions are available before visibility tests, we no longer need to predict, and instead focus on scheduling the optimal order of the data access.

Out-of-core radiosity is another important technique to achieve global illumination effects on massive models. Teller et al. [1994]

[1] In our problem, we do not require the path back to the origin point. But, it also can be formulated into the TSP by creating a dummy point whose distances to every other point is 0.

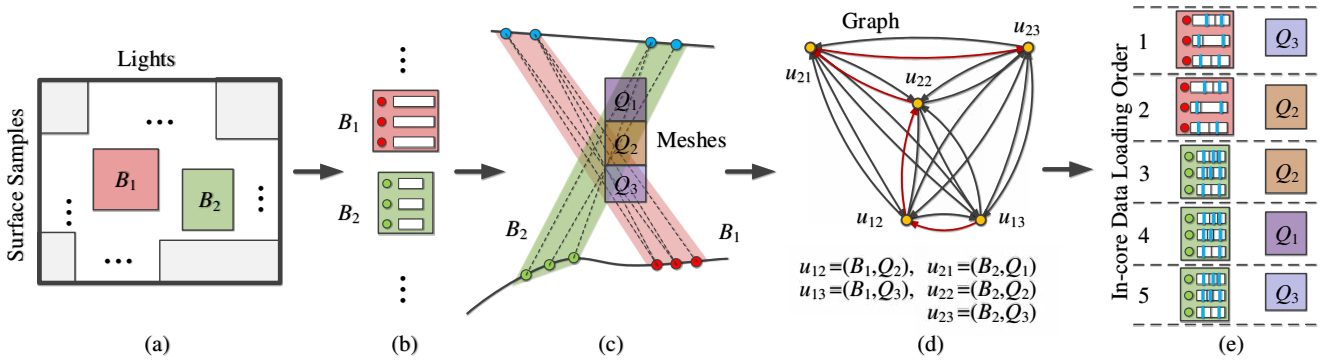


Figure 2: *Algorithm overview. (a) The light transport matrix A is divided into (b) submatrices, B_1, B_2, etc. (c) There are some floating occluders between lights and surface samples in submatrices, B1 and B2. The geometry of these occluders are packed into mesh blocks Q_1, Q_2 and Q_3. Rendering of submatrices, B_1 and B_2, require loading submatrices and mesh blocks into device memory. (d) To manage the I/O of these data, we formulate the data management into a graph traversal problem and find the optimal path with minimal I/O cost. (e) From the optimal path, data are loaded and computed in in-core memory accordingly. The I/O and computation order is shown as numbers.*

presented a spatial partition and ordering method to schedule the intermediate results for later, memory-coherent shading. Meneveaux et al. [1998] also utilized the knowledge of scenes in radiosity computation and provided an efficient memory management based on the precomputed visible order of scene nodes. The many-lights rendering framework inherits some aspects of these advantages in the radiosity method. However, compared with their work of only using scheduling geometry for shading computation, our approach employs two kinds of out-of-core data, lights and geometry, thus it requires different data management.

GPU-based Global Illumination Rendering. With the rapid development of graphics hardware, many methods focus on adapting CPU-based algorithm to the GPU. Recently, many GPU-based or GPU-friendly global illumination algorithms and systems [Zhou et al. 2008; Wang et al. 2009; Parker et al. 2010; Hachisuka et al. 2008; Kaplanyan and Dachsbacher 2013] have been presented to utilize the parallel computational power. Some of these works also target out-of-core data, e.g. the PantaRay system [Pantaleoni et al. 2010]. Compared with them, our method is the first GPU-based approach for the out-of-core many-lights rendering problem.

3 Algorithm Overview

The many-lights rendering problem has been formulated in matrix form [Hašan et al. 2007], where matrix A of size $M \times N$ represents the light transport from N lights to M surface samples. The shading of one surface sample is calculated by summing one column as

$$L_m = \sum_{n=1}^{N} A_{mn} \tag{1}$$

where L_m is the outgoing radiance of sample m and A_{mn} is the contribution of light n to sample m, which can be decomposed into

$$A_{mn} = M_{mn} G_{mn} V_{mn} I_n \tag{2}$$

where M_{mn}, G_{mn}, V_{mn} and I_n are terms of the material, geometry, visibility and intensity of light n respectively.

In this formulation, the computation of L_m can be mainly split into two steps: the matrix sampling and the shading. These two steps require three different kinds of data: surface samples, lights and scene geometry. At the first step, surface samples and lights are used in Eq. (1) to choose best representative lights for each sample. Then, at the second step, geometry is loaded in evaluating visibilities of sample-light rays in Eq. (2). These two different computation steps with different data make the computational patterns unfriendly to the data access patterns, especially in the case of rendering scenes with out-of-core data sizes of lights and geometry. We denote the I/O costs of loading the l-th data block of lights and k-th data block of geometry as t_l and t_k respectively. Thus, the overall goal of out-of-core data management to load these two kinds of data in many-lights rendering is represented as

$$\min_{n_l, n_k} \sum_{l} n_l t_l + \sum_{k} n_k t_k \tag{3}$$

where n_l and n_k are the number of times that the algorithm loads the l-th light block and k-th geometry block respectively. From Eq. (3), the optimal data management is to minimize n_l as well as n_k. However, due to the computational correlation of lights and geometry, there is a conflict in minimizing both of them at the same time. When a light block is loaded in in-core memory, the best data management strategy for that block is to take all computation with surface samples as well as all visibility tests before unloading it. However, these visibility tests may increase n_k, the I/O number on geometry blocks. On the other hand, when a geometry block is loaded, it prefers to test visibilities with as many sample-light rays as possible, which will incur high a n_l.

3.1 Graph-based Out-of-core Many-lights Rendering

To optimize Eq. (3), in this paper, we formulate the out-of-core data management of these two kinds of data, lights and geometry, into a graph traversal optimization problem. The optimized traversal route is used to guide the many-lights rendering. Fig. 2 gives a basic illustration. First, the computation on the whole light transport matrix, Fig. 2(a), is divided into smaller submatrices, B_l, Fig. 2(b), where each submatrix contains contributions from a portion of lights to some surface samples. The rendering in each submatrix requires different geometry blocks to be loaded into in-core memory for visibility tests, Fig. 2(c). We composite submatrices, B_l, with potentially intersected geometry blocks, Q_m, to make up a graph, which is named the submatrix-geometry graph, Fig. 2(d). We use $u_{lm} = \{B_l, Q_m\}$ to denote the graph vertex. The edge from one vertex to another, e.g. u_{ip} to u_{jq}, is defined by a cost function to swap data of these two graph vertices. Note that according to different block sizes, the edge is directional because the I/O overheads of swapping u_{ip} with u_{jq} and u_{jq} with u_{ip} might be different. The conservative visibility tests between all surface samples and lights require every vertex in the graph to be traversed. Thus, the optimal data management of the minimal I/O overhead, Eq. (3),

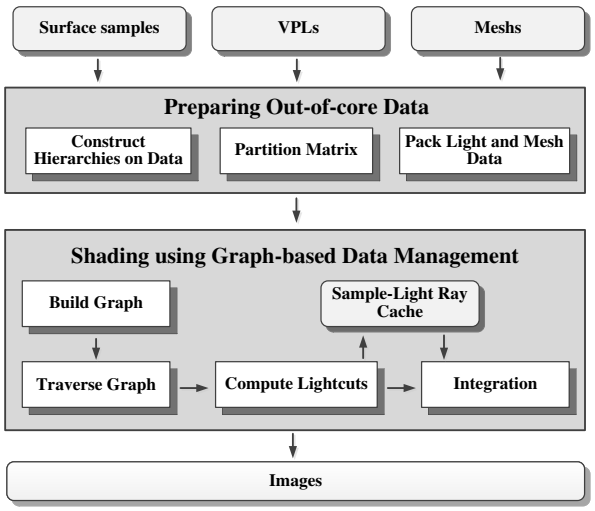


Figure 3: *Our algorithm is split up into two main steps: out-of-core data preparation (described in Section 4), and out-of-core shading using our graph-based data management (described in Section 5).*

is converted into a graph traversal optimization, the traveling salesman problem, that finds a shortest path to visit every vertex in the graph once. Once we have the optimal path, the red one shown in Fig. 2(d), lights and geometry data are loaded and computed accordingly to shade the final image, Fig. 2(e). In the shading computation, in each submatrix, representative lights are chosen from a light tree by a modified relative error bound from Lightcuts [Walter et al. 2005] and contribute to each surface sample.

3.2 Algorithm Steps

We split up our many-lights rendering algorithm into two steps: out-of-core data preparation and out-of-core shading using graph-based data management. The purpose of the preparation step is to pack and organize data into an improved data layout for the graph representation and the shading. In the shading step, the graph is traversed and optimal routes are exploited for the many-lights rendering. The steps of our algorithm are shown in Fig. 3. We first briefly introduce these steps and then provide details of each step in following sections.

1. In data preparation, unorganized surface samples, lights and geometry represented in meshes are first constructed into hierarchies. Then, the whole matrix is adaptively partitioned into submatrices. Finally, mesh data are packed into blocks according to these partitioned submatrices.

2. At shading, the submatrix-geometry graph is constructed and traversed to find the shortest path visiting all vertices. The submatrix and mesh data of these vertices on the path are sequentially loaded into in-core memory and visibility tests are performed. All contributions from visible representative lights are integrated into surface samples to generate the final image.

4 Out-of-core Data Preparation

This section describes our data preparation step. First, we introduce the hierarchy construction method to organize out-of-core input data, i.e., surface samples, lights and meshes. Then, we present a new matrix partition algorithm that divides the matrix into submatrices. Finally, we describe the mesh packing process to prepare mesh data into blocks for the out-of-core data management. After the data preparation step, data are specifically organized for the graph representation, graph traversal and graph-based shading.

4.1 Constructing Hierarchies

For different kinds of data (surface samples, lights and meshes), different hierarchies are constructed. For out-of-core lights and meshes, data are organized into two-level hierarchies. At the lower level, lights and meshes are grouped into chunks with approximately equal sizes, and within each chunk a local hierarchy is constructed and maintained. The low-level hierarchies are stored and loaded in-core or out-of-core with the chunk data. At the higher level, a global hierarchy is built on chunks. The high-level hierarchy is constructed among chunks and always kept in in-core memory.

For lights, point KD-trees [Zhou et al. 2008] are used for both levels. For meshes, at the higher level, we use the Hierarchical Linear Bounding Volume Hierarchy (HLBVH) [Pantaleoni and Luebke 2010]. At the lower level, we use the Split Bounding Volume Hierarchy (SBVH) [Stich et al. 2009]. The SBVH is able to provide tighter bounding volumes and is more efficient in intersection tests. But it requires multiple references to geometry, which is unsuitable for the high-level hierarchy. The construction of these two-level hierarchies takes three steps. First, elements of these two kinds of data, point lights or triangles, are sorted according to their spatial Morton code [Pantaleoni and Luebke 2010]. Next, we partition these sorted elements into chunks and build low-level hierarchies within each chunk. After the construction of low-level hierarchies, each chunk with its hierarchy is streamed out to the host memory. Only the bounding boxes of chunks are used to construct high-level hierarchies.

For surface samples, due to a relatively small data size, we store all surface samples in-core and use a clustering algorithm to construct the hierarchy. Specifically, a bottom-up clustering scheme is employed. The error metric utilized in the clustering is based on a screen-based irradiance caching approach [Wang et al. 2009] that measures geometry variations between surface samples,

$$\varepsilon = \alpha \, ||x_i - x_k|| + \sqrt{2 - 2(\vec{n}_i \cdot \vec{n}_k)} \tag{4}$$

where x_i is a surface sample to be classified, x_k is the center position of a cluster C_k, and $\vec{n}$ denotes a surface normal. α is a weighting factor that determines the relative importance of position and normal incurred changes.

4.2 Partitioning the Matrix Adaptively

The basic principle of matrix sampling in previous methods [Walter et al. 2006; Hašan et al. 2007; Ou and Pellacini 2011] is to find representative lights so as to reduce the rendering computation. In our method, we need to partition the matrix into submatrices satisfying our data management requirements. First, we need to divide the matrix into small parts so that each one can fully fit into device memory. Second, our graph-based data management prefers spatially coherent partitions on lights and surface samples to concentrate the computation of visibility tests. Finally, since the graph is made up of submatrices and their potentially intersected meshes, the sizes of submatrices should be appropriate for the graph representation and data management. Too many submatrices bring more challenges in computing optimal data management while too few submatrices will result in large I/O overhead. Thus, to satisfy these requirements, we adapt the simultaneous traversal in [Walter et al. 2006] to traverse hierarchies of both lights and surface samples but with different traversal metrics.

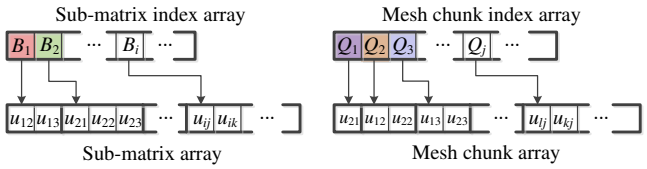


Figure 4: *Submatrix-geometry graph storage illustration of Fig. 2(d).*

In traversing these hierarchies, one important computation is to find potentially intersected mesh chunks for surface samples and lights in one submatrix. In order to find these mesh chunks quickly, we use bounding volumes to bound actual sample-light rays generated by these surface samples and lights. Specifically, we compute two bounding spheres that bound surface samples and lights respectively. The envelope bounding the space between these two spheres forms a shaft. A 2D example is shown in Fig. 2(c). The green and the red regions are two such shafts. If the bounding box of one mesh chunk potentially intersects one shaft, we regard it as a potentially intersected mesh chunk to these surface samples and lights. The intersection tests between the shaft and the bounding box can be carried out efficiently using distance queries between the median axis of the shaft and the bounding boxes in the mesh hierarchy [Schneider and Eberly 2002]. If the distance to one bounding box is larger than the distance to the median axis, there is no intersection. Otherwise, they are intersected.

The simultaneous traversal is carried out in a top-down scheme. Given one parent light node and one parent sample node in the hierarchies, we split parent nodes into child nodes and pair each child node with the other parent node to generate candidate submatrices. For each candidate submatrix, we compute the bounding shaft and use the bounding shaft to find all potentially intersected mesh chunks from the high-level mesh hierarchy. The split of the parent node that produces a larger difference of potentially intersected mesh chunks is chosen as the next traversal direction. Recursive traversals will later be carried out on child nodes of the parent that is not split this time. Once the number of potentially intersected mesh chunks of both of the two splits are the same, we stop the traversal and output two nodes. For each submatrix, we maintain a bit array to indicate the intersection status of all mesh chunks. If one mesh chunk is potentially intersected, the corresponding bit is set to 1 or otherwise 0. By bit operations, we can easily find whether these two submatrices have the same potentially intersected mesh chunks or how different these potentially intersected mesh chunks of two submatrices are. Once all traversal on both hierarchies stop, outputted pairs are organized into submatrices.

We conduct the simultaneous traversal in a parallel manner for implementation on the GPU. In order to fully utilize parallel streaming processors, traversals are organized into batches. We start with a lower level of nodes for both hierarchies. Initial pairs of nodes are put into an active list. In each batch, node pairs in the active list are processed in parallel. To make sure that the active list does not grow too quickly, each time we only process $n_{\max}$ pairs and traverse hierarchies in a depth-first fashion by always processing the latest pairs generated from the last batch. Please refer to the supplementary document for more implementation details.

4.3 Packing Data

After obtaining all submatrices, we pack light and mesh chunks into blocks according to the partition of the matrix. Each block is one I/O unit in the shading step and is used to construct the graph. From our simultaneous traversal, lights in one submatrix are from a subtree of one light hierarchy node. Thus, we pack each light subtree into one block and further use the light subtree to compute lightcuts in shading. For mesh data, we reuse these bit arrays that indicate the intersection status between submatrices and mesh chunks in the aforementioned matrix partition process. For each pair of mesh chunks, we compare the intersection status with submatrices. If the status of several mesh chunks are exactly the same and the total size of these mesh chunks can be loaded into device memory at once, we combine these mesh chunks into one mesh block.

5 Out-of-core Shading using Graph-based Data Management

In this section, we introduce our out-of-core shading using the graph-based data management. At first, we describe the graph made up by submatrices and mesh blocks. Then, we present three different graph traversal solutions to obtain the optimal data management of lights and meshes. Finally, we describe our modifications on the Lightcuts method [Walter et al. 2005] to compute representative lights and the final integration.

5.1 Building the Graph

After all submatrices are generated and mesh blocks are packed, we use them to construct the submatrix-geometry graph. In the graph, vertices are paired by submatrices with their potentially intersected mesh blocks. The edge weight from one vertex, u_{ij}, to another vertex, u_{lm}, is defined by a cost function to replace u_{lm} by u_{ij} in device memory. It is computed as

$$w(u_{ij}, u_{lm}) = t(u_{ij}, u_{lm}) = t(B_i, B_l) + t(Q_j, Q_m) \quad (5)$$

where light blocks, B_l, and mesh blocks Q_m are required to be loaded in to replace B_i and Q_j.

The submatrix-geometry graph is a complete graph, where every two vertices are connected by an edge, because any vertices in the device memory can be replaced by any other vertices. To reduce the storage size of the complete graph, we store it in a compact form that only edges between vertices that share one data block are recorded and stored. Other edges are computed at runtime. The rationale is that the I/O cost to replace one vertex with another by only loading one light block or mesh block is less than that to load both of them. Thus, these edges between vertices that share one data block have low weights, and have a high probability to be traversed in graph traversal optimization. In this way, the storage size is reduced by only storing low weight edges but still retains efficiency in graph traversal optimization.

Fig. 4 illustrates the storage layout of the graph. Graph vertices are stored in two data arrays with two index arrays: the submatrix data array stores graph vertices in the order of submatrix indices and the mesh data array in the order of mesh block indices. Index arrays are used to record the start addresses of each submatrix or mesh block in data arrays.

5.2 Traversing the Graph

The data management of lights and geometry is formulated into a TSP that finds the shortest path to visit all vertices in the graph once. Compared to the standard TSP problem, our formulation introduces new challenges. First, because the device memory buffer size is usually larger than the packed data block size, to maximize the computational power, we need to load more than one data block in the device memory buffer for computations. This will make the edge weight non-deterministic. Here is an example. If vertices, u_{12} and u_{13} in Fig. 2, are loaded in the device memory, the cost function

of $t(u_{12}, u_{23})$ is no longer $t(B_1, B_2) + t(Q_2, Q_3)$ but $t(B_1, B_2)$, because Q_3 block has already been loaded in device memory. In these cases, the edge weight does not only depend on two vertices, but is associated with all vertices currently in the device memory buffer. Based on the classification in [Ghiani et al. 2003], our problem can be classified into a static but non-deterministic route optimization problem. Second, since the time of graph traversal optimization counts towards the total rendering time, we cannot spend a large amount of time on the optimization even if we would be able to compute a best route. A fast optimization is required to avoid canceling the benefits gained from optimal data management.

To solve the specific graph traversal problem in our method, we present three solutions. The first solution is an approach based on the minimum spanning tree (MST) that first generates a MST and then uses the MST to build a route. The MST is a good approximation for a static and deterministic TSP, because the cost of MST is at most twice of the most optimal route. The second solution is a local nearest neighbors search algorithm with online update of edge weights. The third one is an ant colony graph optimization solution that utilizes the parallel computational power provided by the GPU.

MST-based graph traversal. The MST is built by Prim's algorithm [PRIM 1957] that starts with a set containing a randomly selected vertex, and then iteratively inserts new vertices with the least weight edge one at a time, until the set spans all vertices. In the construction of the MST, the edge weights are regarded as static. Once we have the MST, we traverse the MST in a depth-first order to obtain a path to visit all vertices in the graph. These vertices on the path are sequentially loaded into in-core memory to compute the shading.

Local graph traversal. We employ a local optimization heuristic, nearest neighbors (NN), to guide the graph traversal [Johnson and McGeoch 1995]. Under such a heuristic, a feasible path is always constructed by taking the decision that is mostly advantageous at each step. Initially, we choose a set of vertices as active vertices to fill the in-core memory buffer, $\{u_{ij}\}$. Then, for these active vertices, the NN traversal strategy is always to select the next nearest as-yet-unvisited vertices.

Ant colony-based graph traversal. Ant colony optimization [Dorigo and Stüzle 2010] is a population-based method and has been widely used in many combinatorial optimization problems, such as TSP. In this paper, the optimization is executed in iterations. At the beginning of each iteration, a number of artificial ants are generated in parallel to select vertices to be visited according to a function based on pheromone stored at the edges. Pheromone is accumulated from other ants and previous iterations. At the end of an iteration, graph edges are parallelized to collect pheromone values generated by the ants. These pheromone values are used to bias ants in subsequent iterations and guide ants to utilize the best paths previously constructed and find new paths. In order to avoid taking a long time in optimization, we approximate the optimal route by several suboptimal routes by dividing the entire graph into subgraphs, then using ants to traverse each subgraph independently, and finally connecting routes in each subgraph. We use clustering to divide graphs into spatially coherent graphs and the number of vertices in each cluster is set to approximately 5000 in our method.

More implementation details on these graph traversal strategies are provided in the supplementary document.

5.3 Computing Lightcuts and Integration

After we have the optimized graph traversal path, we load the light and mesh data of graph vertices in device memory to compute these representative lights in each submatrix and integrate their contributions for individual surface samples. For each surface sample in one submatrix, we take a top-down light hierarchy traversal scheme like that in [Walter et al. 2005] to compute representative lights. For each traversed light node, the maximum error bound is computed as that in [Walter et al. 2005], where the visibility term is one. If the error bound is less than the relative error threshold, no traversal will be performed in the subtree, and this light node is chosen as a representative light. Otherwise, the traversal continues down the light hierarchy. However, different from the original Lightcuts method that sequentially refines a coarse cut to final one, the cut in our method is refined in parallel in each submatrix and computed relatively independently among submatrices. Thus, the way to estimate the total radiance for the relative error criterion in Lightcuts cannot be applied in our method. To solve this difficulty, we make two modifications. First, our method starts by computing a coarse estimation from the high-level light hierarchy to obtain a coarse estimate of total radiance. The estimate of total radiance is not update until one submatrix is fully processed. Second, in processing one submatrix, we use an estimate of relative error as the relative error threshold to drive the cut refinement. The estimate of relative error, e_r, for a surface sample in submatrix, i, is defined as

$$e_r = r\frac{I_i}{I_{all}}\widetilde{L}_{sample} + \frac{L_a}{10}, \tag{6}$$

where r is a constant (2% in our experiments), I_i is the total cluster intensity of the light subtree in submatrix i, I_{all} is the total intensity of all lights, $\widetilde{L}_{sample}$ is the currently estimated total radiance for the sample and L_a is the perceptual term like that in [Walter et al. 2006]. The rationale behind Eq. (6) is that before we actually sample lights in the light subtree of one submatrix, we assume that the error produced in this light subtree of the total error is proportional to the light subtree intensity of the total intensity. By applying these two modifications, we are able to construct cuts for each submatrix in parallel and process submatrices in an independent order.

After all representative lights are chosen, sample-light rays are generated and stored in a sample-light rays cache for further visibility tests. We parallelize visibility tests of rays with mesh blocks in device memory. We directly perform the ray intersection tests on low-level hierarchies, the SBVHs. In each visibility test, an accurate intersection point is unnecessary. Instead, the intersection search stops as soon as one object is intersected. After all in-core meshes have been tested, rays that are occluded are removed from the sample-light rays cache. Rays that have been tested with all potential mesh chunks are compacted into the buffer of visible sample-light rays. In integration, these lights with visible rays in the visible sample-light ray are accumulated to pixels.

To further reduce the latency of data transfer from the CPU to GPU, we use asynchronous data transfer. Two streams are created and used in the out-of-core shading to overlap the transfers with computation. We launch kernels in one stream to compute lightcuts, take visibility tests and do the integration, and transfer data asynchronously from the CPU to GPU in the other stream. In the shading process, when one vertex in the optimal path is processed, we check if there is buffer space available for the upcoming computation of the next vertex. If so, we launch the asynchronous data transfer to fill these buffers. More implementation details can be found in the supplementary document.

6 Results and Discussion

We have implemented all our algorithms using CUDA. In the following, we present our results computed on a PC with an Intel Core$^{\text{TM}}$ i7 3770 CPU, 64GB RAM, and an NVIDIA GeForce GTX

680 graphics card with 2GB RAM. Unless mentioned otherwise, all images reported are rendered at a resolution of 800×600 with 4 supersamples. Surface samples are generated by an out-of-core ray tracer that traces rays from supersamples of pixels. There are several ways to generate VPLs fast and efficiently, such as a standard VPLs generation [Keller 1997] using an out-of-core ray tracer or sampling mesh surfaces by randomly distributing samples in triangles [Kontkanen et al. 2011]. We generated surface samples and VPLs in a preprocess and fed them with geometry as input for our algorithm.

Our host buffer consists of a light buffer and a mesh buffer. The light buffer and mesh buffer are allocated to contain all lights and meshes along with their low-level hierarchies. Each light chunk and mesh chunk size is set to approximately 50 MB, but after adaptive matrix partitioning and mesh data packing, the light and mesh blocks have variable sizes from 50 MB to 400MB. The device buffer consists of a light buffer, a mesh buffer, and the sample-light ray cache buffer. By default, we allocate two 500 MB buffers for meshes and lights and 400 MB for the sample-light ray cache. The rest of device memory is used for temporary storage of different kernels.

We tested our algorithm on different scenes: the Museum, the Airport, the Chinese Town and the Carnival. For each scene, we generated images from at least two views. Table 1 summarizes statistics of these scenes.

Museum. The left image of Fig. 1 and Fig. 7(a) shows a scene containing several statues and models in one museum, including David, Lucy, Neptune, Satva, the asian dragon and two dinosaur skeletons. This scene consists of 117.1 M triangles in total. The triangles and the two-level hierarchies of the scene take about 14.3 GB. The number of lights is 32.4 million and these lights with hierarchies occupy 3.75 GB of host memory.

Airport. The middle image of Fig. 1 and Fig. 7(b) shows the Airport scene with two Boeing 777s and a hangar. This scene consists of 669.3 million triangles. The triangles and the two-level hierarchies of the scene take 46.3 GB. 64 area lights, shown in spheres, are distributed in the hangar. The environmental lighting is partitioned into 20 thousand point lights. The total number of lights is 18 million and these lights with hierarchies occupy 2.1 GB of host memory.

Carnival. The right image of Fig. 1 and Fig. 7(c) shows the Carnival scene. The original scene is only approximately 200 thousand triangles but has complex lighting. To demonstrate our approach, we added some tree models and subdivided the scene into 17.1 million triangles. The triangles and the two-level hierarchies of the scene take 1.76 GB. The number of lights is 256 M and these lights with hierarchies occupy 29.6 GB of host memory.

Chinese Town. Fig. 8 shows the Chinese town scene. This scene consists of 88.1 million triangles. The triangles and the two-level hierarchies of the scene take 8.1 GB. A directional light illuminates the scene from right to left. The total number of lights is 21.4 million, and these lights with hierarchies occupy 2.5 GB of host memory.

6.1 Errors

The relative error bound in this paper is set to 2%. In Fig. 5, we show a comparison with the ground truth result generated by bruteforcedly accumulating all light contributions to pixels. It can be observed that our approach is able to capture the main visual effects. As we decrease the relative error bound, the global image error decreases. Though we have a different relative error criterion in choosing representative lights, our method still inherits the stochastic error-bound merit of Lightcuts [Walter et al. 2005; Walter et al. 2006]. We also visualize per pixel cut sizes of our method and Lightcuts [Walter et al. 2006] in Fig. 6. It can be seen that our per pixel cut size is larger than that in Lightcuts [Walter et al. 2006], indicating that our algorithm conservatively estimates the relative errors and produces less image error.

		Museum	Airport	Chinese Town	Carnival
Meshes	tris. (m)	117.12	669.3	88.1	17.1
	size (GB)	14.3	46.3	8.1	1.76
Texture	size (MB)	186.3	0	274.8	182
Lights	num. (m)	32.4	18	21.42	256
	size (GB)	3.75	2.1	2.51	29.6
Samples	num. (m)	1.9	1.8	1.6	0.99
Mesh	chunks	875	1804	367	183
	blocks	167	820	121	35
Lights	chunks	154	87	105	1172
	blocks	43	16	32	110
Submatrices	num. (k)	6.1	6.4	3.6	10.5
Graph vertices	num. (k)	58.2	132.2	18.3	30.4

Table 1: *Data statistics of view #1 in the test scenes. For each scene, we list the numbers and sizes of triangles and lights, the sizes of texture, the number of samples, numbers of chunks and blocks of meshes and lights, numbers of submatrices and the graph vertices. m and k indicate million and thousand respectively. MB and GB are Megabytes and Gigabytes respectively.*

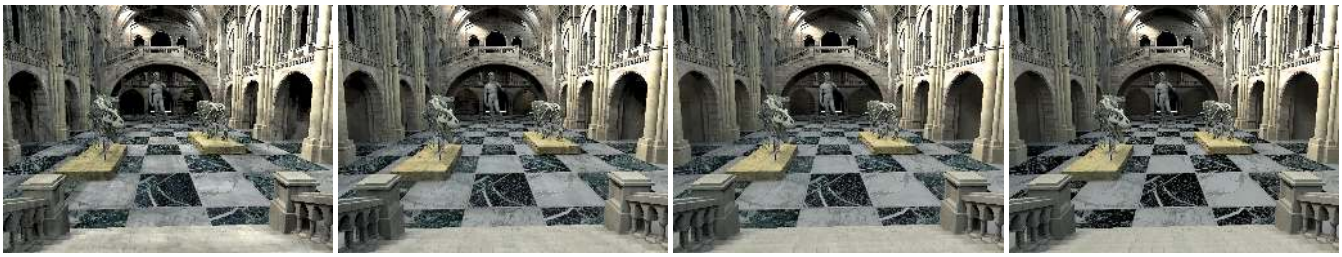
6.2 Data Preparation

In Table 1, we list the number of chunks, blocks, submatrices, and graph vertices produced by our out-of-core data preparation step. To validate our data preparation step, when we used two graphs, with and without data packing to do data management, under the local search strategy, the total rendering times using the graph with data preparation were 64.2, 456.1, 16.9 and 32.6 seconds, compared with 113.5, 605.6, 51 and 60.9 seconds without the data preparation. From the performance improvement, our data preparation step clearly provides better data layout and data management.

6.3 Performance

The timings spent in each step of our approach are listed in Table 2. The hierarchy construction takes tens to hundreds of seconds. In shading, compared with the time to compute lightcuts and do integrations, the time to perform visibility tests and manage data between CPU and GPU dominate the entire computation. To assess the performance of our graph traversal strategies, we introduce two naive traversal strategies. The first one is to minimize n_l in Eq. (3) such that, once one light block is loaded in, it is swapped out only after all surface samples have been processed with lights in such a block. The second naive strategy is to minimize n_k in Eq. (3) such that, once one mesh chunk is loaded in, it is swapped out only after all submatrices that it potentially intersects have been tested for visibility. These two naive strategies are compared with three graph traversal solutions, the MST-based traversal, the local search traversal and the ant colony optimization traversal. The reported timings of I/O are are determined after applying the asynchronous data transfer. Even if some latency has been hidden, in all scenes, the I/O costs of naive strategies are more than the computational time. After applying optimal data management, in some scenes, such as the Airport and Carnival, it still takes longer to load data than to perform visibility tests. It demonstrates that without proper data management, the I/O overhead becomes the main bottleneck.

Comparison of scenes. The Airport scene is an outdoor scene and



(a) $e_{bound} = 10\%$ (b) $e_{bound} = 5\%$ (c) $e_{bound} = 2\%$ (d) Ground truth

(e) Diff. image of (a) (f) 8× diff. image of (a) (g) Diff. image of (b) (h) 8× diff. image of (b) (i) Diff. image of (c) (j) 8× diff. image of (c)

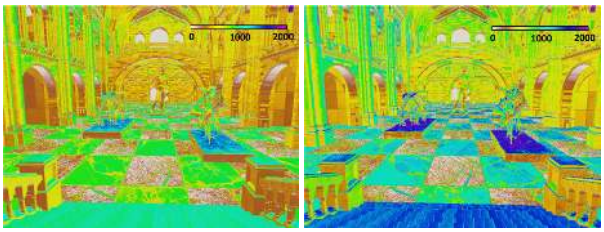
Figure 5: *Comparison under different error bounds.*

		Museum	Airport	Chinese Town	Carnival
Build VPL hierarchies (s)		8.5	4.4	5.9	63.5
Build Mesh hierarchies (s)		71	441	48	12.4
Partition matrix (s)		1.8	1.0	1.6	2.2
Pack data (s)		2.1	5.8	1.3	1.5
Compute lightcuts (s)		26	19.6	17.9	36.5
Test visibility (s)		141	170.1	27.4	16.5
Integration (s)		3.3	1.8	1.0	2.8
Naive traversal 1	#1 time (s)	182.9	862.4.4	65.4.9	48.7
	#2 time (s)	176.2	1015.2	62.1	52.5
Naive traversal 2	#1 time (s)	136.2	1734.1	76.5	63.3
	#2 time (s)	123.5	1826.5	73.1	69.1
MST traversal	#1 time (s)	106.1	663.1	52.8	46.4
	#2 time (s)	71.2	688.3	51	49.1
Local search traversal	#1 time (s)	64.2	456.1	16.9	32.6
	#2 time (s)	59.5	470.8	15.3	36.1
Ant colony traversal	#1 time (s)	60.6	416.3	14.3	28.2
	#2 time (s)	46.1	425	13.8	30.4

Table 2: *Timings in different scenes. Statistics of view #1 of each scene include the numbers of surface samples, submatrices, nodes in the matrix-mesh graph and the average cut size per surface sample are given. The times to preprocess the meshes and lights, to partition the matrix, to sample submatrices, to test visibilities and to do integration of view #1 of each scene are listed. Finally, to compare the performance of different strategies on traversing the matrix-mesh graph, the time spent on data management of both view #1 and #2 are included.*

has the largest number of triangles but with a relatively small number of lights. Thus, it takes less time to partition the matrix and compute lightcuts, but takes the longest time to perform visibility tests and load mesh blocks in the graph traversal. The Museum is an indoor scene with a large number of lights and triangles. The complexity of visibility makes the shading take much more time than that of the Chinese town, even though they have similar numbers of lights and triangles. The Carnival scene has the largest number of lights but the least number of triangles. Having a small number of triangles makes the data management of lights and meshes more efficient than that of the Museum, even though the Carnival has much more light data to manage.

Comparison of different graph traversal strategies. In Table 2, the time to compute the graph route is included in the traversal time. For view #1 of the Airport scene, the time to compute the optimal path is 15.2s, 4.1s and 21.2s for the MST-based traversal, the local search traversal and the ant colony traversal. Compared with the



(a) Lightcuts [Walter et al. 2006] (b) Our method (Fig. 5(c))

Figure 6: *Cut size comparison.*

	Museum	Airport	Chinese Town	Carnival
Naive 1(s)	266.9 (84.0)	1027.0 (164.6)	94.8 (29.4)	71.3 (22.6)
Naive 2(s)	247.5 (111.3)	1899.8 (165.7)	113.9 (37.4)	91.3 (28.0)
MST	151.8 (45.7)	770.0 (106.9)	79.5 (26.7)	65.7 (19.3)
Local search	97.3 (33.1)	525.7 (69.6)	29.0 (12.1)	43.5 (10.9)
Ant colony	90.3 (29.7)	469.3 (53.0)	25.3 (11.0)	36.2 (8.0)

Table 3: *I/O timings in seconds for four scenes (view #1) using different strategies without asynchronous data transfer. The hidden latency is shown in brackets.*

reduced time obtained by our graph-based data management, the costs on optimizing pathes are satisfactory. Additionally, for the Museum scene, we record the I/O data transfer per 32×32 pixels using different strategies and plot them in Fig. 9. These comparisons demonstrate the high data coherence of our data management strategies, especially the one with the ant colony optimization. In all scenes, it outperformed other strategies, and, in some scenes, it reduced the time to one fifth. However, in the Carnival scene, the improvement of the local optimization and ant colony optimization is not as significant. This is mainly because, in the Carnival scene, the number of mesh blocks is relatively small. The traversal of Naive Strategy 1 is almost equal to other optimized traversals.

Asynchronous data transfer. The I/O timings using different strategies without asynchronous data transfer are provided in Table 3. The hidden latency is shown in brackets. It can be observed that the asynchronous data transfer is beneficial to the performance for all scenes and all strategies. It can also be found that the hidden latency is decreased with better strategies. This is because with a better strategy, the in-core memory buffer is optimally used to load more blocks. With less free buffer space, the space to hide the I/O latency is less. Additionally, with a better strategy, when light and mesh blocks are loaded in memory, they tend to tend to take more

(a) Museum view #2 (b) Carnival view #1 (c) Airport view #2

Figure 7: *Views of different scenes.*

(a) View #1 (b) View #2

Figure 8: *Chinese town.*

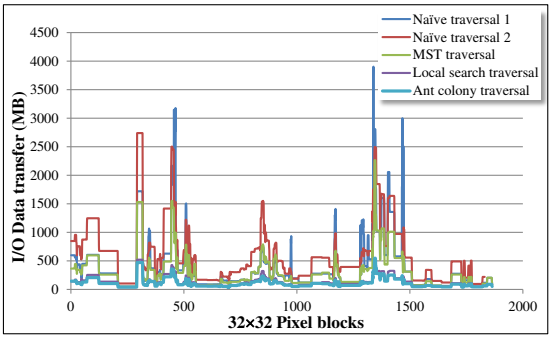


Figure 9: *The I/O data transfer using different strategies in the Museum scene. Data transferred for every* 32×32 *image block are counted and plotted.*

computation than that of a less optimal strategy. The less overlapping of computation and data transfer also results in less hidden latency. Although less latency is hidden, optimal strategies reduce the overall I/O cost more than less optimal strategies, especially in scenes where I/O costs dominate. With or without asynchronous data transfer, our graph-based data management can greatly reduce the overall I/O cost and achieve better performance.

Comparison with CPU-based approach. We also compared our GPU-based approach with a CPU-based one. Because all data are stored in CPU memory, in the CPU-based approach, there is no I/O cost. However, only the computational times for view #1 of Museum, Airport, Town and Carnival scenes are 3244, 3740, 2513 and 1958 seconds, respectively. These results demonstrate that our GPU-based implementation has one order of magnitude performance gain over the CPU-based approach.

7 Conclusion and Future work

We have presented a GPU-based rendering approach for the out-of-core many-lights problem. A graph-based data management algorithm is designed to handle the out-of-core data I/O of lights and geometry in the many-lights rendering framework. In order to achieve less I/O overhead, we formulate the data management as a graph traversal problem and employ different optimization schemes to guide the computation. Results demonstrate that our algorithm and the GPU-based implementation exhibit high coherence in accessing the data and provide better performance in rendering complex global illumination effects in out-of-core scenes with complex lighting.

There are several directions to be explored in the future. First, better graph traversal strategies for data management are worth exploring. Algorithms with better results usually require more computation time. The trade-off between the quality of TSP traversal and the time to optimize is an interesting problem to be explored. Second, it would be necessary to further extend the data storage into three levels – GPU memory, CPU memory, and hard disks – to handle larger scale scenes and more complex lighting. With more levels of data storage, the out-of-core data management becomes more challenging. Thirdly, it is interesting to incorporate glossy interreflections into the out-of-core many-lights rendering, where more lights are required and will aggravate the data management problem.

Compared with the well studied out-of-core path-tracing approaches, our out-of-core many-lights rendering method inherits some drawbacks of other many-lights rendering approaches, such as a general inefficiency at handling high-rank light transport effects (e.g., highly glossy materials and complex occlusions). Our main strength is the scalability and efficiency for rendering global illumination effects on massive models with a large number of point lights. It has been demonstrated that hundreds of million or billion VPLs are needed for movie-quality rendering [Kontkanen et al. 2011]. Moreover, with the rapid increase of captured geometry data size, the data size of lights need to be increased to capture detailed illumination for small-scale geometric features. The problem studied in this paper is thus important for high-quality rendering of large scale scenes.

Acknowledgements

We would like to thank the anonymous reviewers for their thoughtful comments. We also want to thank Kavita Bala, Sean Bell, Pramook Khungurn and Joe Kider for their help in improving this paper, and Dan Konieczka (The Carnival) and Alvaro Luna Bautista (The Museum) for permission to use their models. This work was partially supported by NSFC (No. 60903037, 61272301 and 61272305), the 973 program of China (No.2009CB320803) and the Fundamental Research Funds for the Central Universities (No. 2013FZA5015).

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 145–149.

CARSTEN DACHSBACHER, JAROSLAV KRIVANEK, M. H. A. K. A. A. B. W. 2013. Scalable realistic rendering with many-light methods. In *Eurographics*.

CHRISTENSEN, P. H. 2008. Point-based approximate color bleeding. In *Pixar Technical Memo*, Pixar, 08–01.

DIETRICH, A., GOBBETTI, E., AND YOON, S.-E. 2007. Massive-model rendering techniques: A tutorial. *IEEE Comput. Graph. Appl. 27*, 6 (Nov.), 20–34.

DORIGO, M., AND STÜZLE, T. 2010. *Ant Colony Optimization: Overview and Recent Advances*, vol. 146 of *International Series in Operations Research and Management Science*. Springer US.

FRANK, S., AND KAUFMAN, A. 2009. Out-of-core and dynamic programming for data distribution on a volume visualization cluster. *Computer Graphics Forum 28*, 1, 141–153.

GHIANI, G., GUERRIERO, F., LAPORTE, G., AND MUSMANNO, R. 2003. Real-time vehicle routing: Solution concepts, algorithms and parallel computing strategies. *European Journal of Operational Research 151*, 1, 1 – 11.

GOBBETTI, E., KASIK, D., AND YOON, S.-E. 2008. Technical strategies for massive model visualization. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, ACM, New York, NY, USA, SPM '08, 405–415.

HACHISUKA, T., OGAKI, S., AND JENSEN, H. W. 2008. Progressive photon mapping. *ACM Trans. Graph. 27*, 5 (Dec.), 130:1–130:8.

HAŠAN, M., PELLACINI, F., AND BALA, K. 2007. Matrix row-column sampling for the many-light problem. In *ACM SIGGRAPH 2007 papers*, ACM, New York, NY, USA, SIGGRAPH '07.

HECKBERT, P. S., AND HANRAHAN, P. 1984. Beam tracing polygonal objects. *SIGGRAPH Comput. Graph. 18*, 3 (Jan.), 119–127.

JOHNSON, D., AND MCGEOCH, L. 1995. The traveling salesman problem: A case study in local optimization.

KAPLANYAN, A. S., AND DACHSBACHER, C. 2013. Adaptive progressive photon mapping. *ACM Trans. Graph. 32*, 2 (Apr.), 16:1–16:13.

KELLER, A. 1997. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 49–56.

KONTKANEN, J., TABELLION, E., AND OVERBECK, R. S. 2011. Coherent out-of-core point-based global illumination. *Comput. Graph. Forum*, 1353–1360.

MENEVEAUX, D., BOUATOUCH, K., AND MAISEL, E. 1998. Memory management schemes for radiosity computation in complex environments. In *Computer Graphics International*, 706–714.

NOVÁK, J., NOWROUZEZAHRAI, D., DACHSBACHER, C., AND JAROSZ, W. 2012. Virtual ray lights for rendering scenes with participating media. *ACM Trans. Graph. 31*, 4, 60:1–60:11.

OU, J., AND PELLACINI, F. 2011. Lightslice: matrix slice sampling for the many-lights problem. *ACM Trans. Graph. 30*, 6 (Dec.), 179:1–179:8.

PANTALEONI, J., AND LUEBKE, D. 2010. Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *High Performance Graphics*, 87–95.

PANTALEONI, J., FASCIONE, L., HILL, M., AND AILA, T. 2010. Pantaray: fast ray-traced occlusion caching of massive scenes. *ACM Trans. Graph. 29* (July), 37:1–37:10.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July), 66:1–66:13.

PRIM, R. 1957. Shortest connection networks and some generalizations. *BELL SYSTEM TECHNICAL JOURNAL*.

RITSCHEL, T., ENGELHARDT, T., GROSCH, T., SEIDEL, H.-P., KAUTZ, J., AND DACHSBACHER, C. 2009. Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph. 28*, 5 (Dec.), 132:1–132:8.

SCHNEIDER, P. J., AND EBERLY, D. 2002. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA.

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 7–13.

TELLER, S., FOWLER, C., FUNKHOUSER, T., AND HANRAHAN, P. 1994. Partitioning and ordering large radiosity computations. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '94, 443–450.

VITTER, J. S. 2001. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv. 33*, 2 (June), 209–271.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3, 153–165.

WALTER, B., FERNANDEZ, S., ARBREE, A., BALA, K., DONIKIAN, M., AND GREENBERG, D. P. 2005. Lightcuts: a scalable approach to illumination. *ACM Trans. Graph. 24*, 3, 1098–1107.

WALTER, B., ARBREE, A., BALA, K., AND GREENBERG, D. P. 2006. Multidimensional lightcuts. *ACM Trans. Graph. 25*, 3, 1081–1088.

WALTER, B., KHUNGURN, P., AND BALA, K. 2012. Bidirectional lightcuts. *ACM Trans. Graph. 31*, 4 (July), 59:1–59:11.

WANG, R., WANG, R., ZHOU, K., PAN, M., AND BAO, H. 2009. An efficient gpu-based approach for interactive global illumination. *ACM Trans. Graph. 28* (July), 91:1–91:8.

YOON, S.-E., AND LINDSTROM, P. 2007. Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (Nov.), 1536–1543.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5 (Dec.), 126:1–126:11.