박사 학위논문

Ph. D. Dissertation

정적 대용량 모델의 대화형 전역 조명

Interactive global illumination of non-deformable massive models

김 태 준 (金 兌 俊 Kim, Tae-Joon)

전산학과

Department of Computer Science

KAIST

2013

정적 대용량 모델의 대화형 전역 조명

Interactive global illumination of non-deformable massive models

Interactive global illumination of non-deformable massive models

Advisor : Professor Yoon, Sung-Eui

by

Kim, Tae-Joon

Department of Computer Science

KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science . The study was conducted in accordance with Code of Research Ethics[1].

2013. 5. 27.

Approved by

Professor Yoon, Sung-Eui

[Advisor]

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

정적 대용량 모델의 대화형 전역 조명

김 태 준

위 논문은 한국과학기술원 박사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2013년 5월 27일

심사위원장 윤 성 의 (인)

심사위원 박 진 아 (인)

심사위원 신 인 식 (인)

심사위원 김 민 혁 (인)

심사위원 Sun Xin (인)

DCS
20097107

김 태 준. Kim, Tae-Joon. Interactive global illumination of non-deformable massive models. 정적 대용량 모델의 대화형 전역 조명. Department of Computer Science . 2013. 82p. Advisor Prof. Yoon, Sung-Eui. Text in English.

ABSTRACT

Due to advances of modeling techniques, massive models are easily generated these days. Such massive models can consist of hundreds of millions of primitives and thus use more than tens of gigabytes. This high memory requirement is likely to cause serious performance issues on visualization and rendering because of the heavy loads of data accesses. Moreover, it is not trivial to make the data access pattern of global illumination be coherent which aggravates the performance degradation. In this thesis, three directions to address the issues are proposed. To reduce the expensive data transmission from external drives, we propose two kinds of compressed representations of massive models as in-core and out-of-core representations. The proposed representations show significant performance benefits of various application requiring random access. We further accelerate the performance by fully utilizing heterogeneous computing resources, CPU and GPU. We propose a novel framework which drastically reduces data transmission overhead between the heterogeneous resources. By using the framework, an interactive performance of rendering massive models with global illumination effects is achieved. Finally, we propose an optimization method which maximizes both rendering throughput and responsiveness. The rendering framework with the optimization robustly works with running machines of different performances. A user study is performed to show the benefit of the optimization.

Contents

List of Tables

List of Figures

Chapter 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. Such massive models can consist of hundreds of millions of triangles and thus use more than tens of gigabytes of memory.

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 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.

In order to reduce the data access time and memory requirements, out-of-core mesh compression techniques [1, 2, 3, 4, 5] have been introduced supporting random accesses which is required for various applications including ray tracing and collision detection. However, acceleration hierarchies such as bounding volume hierarchies (BVHs) used for the applications still remain as overhead.

The BVHs are widely used to accelerate the performance of various geometric and graphics applications. These applications include ray tracing, collision detection, visibility queries, dynamic simulation, and motion planning. These applications typically precompute BVHs of input models and traverse the BVHs at runtime in order to perform intersection or culling tests. Many different types of bounding volume (BV) representations exist such as spheres, axis-aligned bounding boxes (AABBs) and, oriented bounding boxes (OBBs).

A major problem with using BVHs is that BVHs require large amounts of the memory space. For example, each AABB and OBB node takes 32 and 64 bytes respectively. Therefore, BVHs of large models consisting of hundreds of millions of triangles can take tens of gigabytes of space. Moreover, the typical data access pattern on BVHs cannot be determined at the preprocessing time and is random at runtime. Therefore, accessing BVHs at runtime can have low I/O efficiency and cache utilization.

Although BVHs are intended to accelerate the performance of applications, the additional memory requirement of using BVHs can increase the working set size during the hierarchical traversal and can increase the data fetching time from the disk, which could negate the benefits of using BVHs. This high memory requirement of a BVH is likely to cause more serious performance issues in the coming years, given the well-known widening gap between the computational speed and the data access speed on current commodity hardware [6].

In this thesis, an out-of-core compact BVH representation, *random-accessible compressed BVHs (RACBVHs)* is proposed (Chapter 3), for various applications requiring random access on BVHs of massive models. A cluster-based layout-preserving BVH compression and decompression method is presented supporting transparent random access on the compressed BVHs. In advance, a compact in-core representation which has tightly combined with meshes and BVHs is proposed (Chaper 4).

Most prior methods for rendering massive models mainly have been focused on providing basic visual effects such as local illumination and hard shadows [7]. Supporting global illumination requires significantly more computation than local illumination. More importantly, unlike coherent rays such as

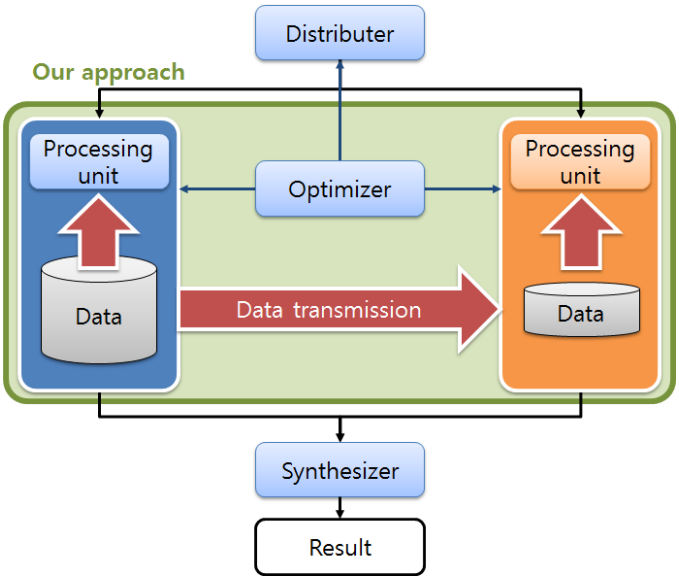


Figure 1.1: Proposed framework can be summarized as this figure. Components in the green box are contributions of our method. Blue and orange boxes indicate heterogeneous computing resources such as CPU and GPU. To maximize the performance, we significantly reduce data transmission cost (red arrows) and fully utilize the heterogeneous resources. In addition, we design a system optimization to make our framework works robustly regardless the target system and user's preferences.

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.

The graphics community has introduced the great computation power of modern graphics cards, which enables many real-time application with incore storage. And out-of-core scheme also gets benefits from it with amount works about highly parallel algorithms on hybrid cores of CPU and GPU. It opens the door to interactive global illumination rendering of massive models. But the video memory on-chip is more limited than the main memory, so the data transmission among three levels storages of disk, main memory and video memory becomes the key problem. We need a method to deeply exploit the tremendous computation power of GPU while alleviating the data transmission from GPUs as much as possible.

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 thesis, novel techniques are presented enabling interactive, high-quality global illumination 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.

1.1 List of related paper

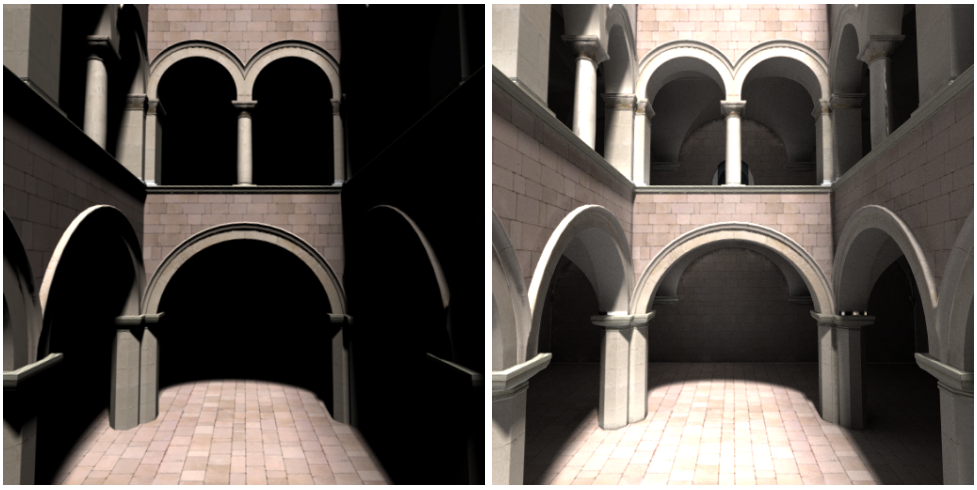
Some parts of this thesis have been published or in under revision. Following is a list of papers related to this thesis.

- **Tae-Joon Kim**, Bochang Moon, Duksu Kim, and Sung-Eui Yoon, *RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies* , IEEE Transactions on Visualization and Computer Graphics, vol. 16, no. 2, pp. 273-286, 2010.
- **Tae-Joon Kim**, Yongyoung Byun, Yongjin Kim, Bochang Moon, Seungyong Lee, and Sung-Eui Yoon, *HCCMeshes: Hierarchical-Culling oriented Compact Meshes* , Computer Graphics Forum, vol. 29, no. 2, pp. 299-308, 2010.
- **Tae-Joon Kim**, Xin Sun, and Sung-Eui Yoon, *T-ReX: Interactive Global Illumination of Massive Models on Heterogeneous Computing Resources* , IEEE Transactions on Visualization and Computer Graphics, under revision (minor).

Chapter 2. Background and related work

2.1 Global Illumination

Global illumination (GI) is a term for rendering algorithms which generate illumination effects from other objects that are not emitting lights themselves. Fig. 2.1 shows rendering results with and without GI effects. As seen in the right image, GI generates much realistic rendering image.



(a) Direct illumination (b) Global illumination

Figure 2.1: The left image shows rendering result of direct illumination and the right image shows rendering result including global illumination effects.

The GI effects can be computed by solving the rendering equation [8]:

$$L_o(x, w_o) = \int_{\Omega} f(x, w_o, w_i) L_i(x, w_i) cos\theta dw_i \tag{2.1}$$

where L_o is a radiance at x toward w_o, L_i is an irradiance from w_i to x, f is a reflectance at x with ingoing and outgoing direction, and θ is an incident angle (See Fig. 2.2). To compute pixel intensity at p in Fig. 2.2, we need to compute L_o at x toward w_o. However, there is not known any general way to compute the integral. Unbiased Monte Carlo ray tracing approach (e.g., bidirectional ray tracing [9]) based on the rendering equation is the standard reference solution of global illumination, but converges to the reference slowly. High-quality rendering techniques have been long studied, and good books are available [10, 11].

2.2 Interactive Global Illumination

One of common ways to reduce the computation cost of indirect illumination is that precompute each light transport between locations assuming the geometry is static. Precomputed radiance transfer(PRT) [12, 13, 14] proposed by Sloan et al. efficiently computes low frequency effects in static geometries and dynamic lights scenes by combination of precomputed basis illumination. Several researches

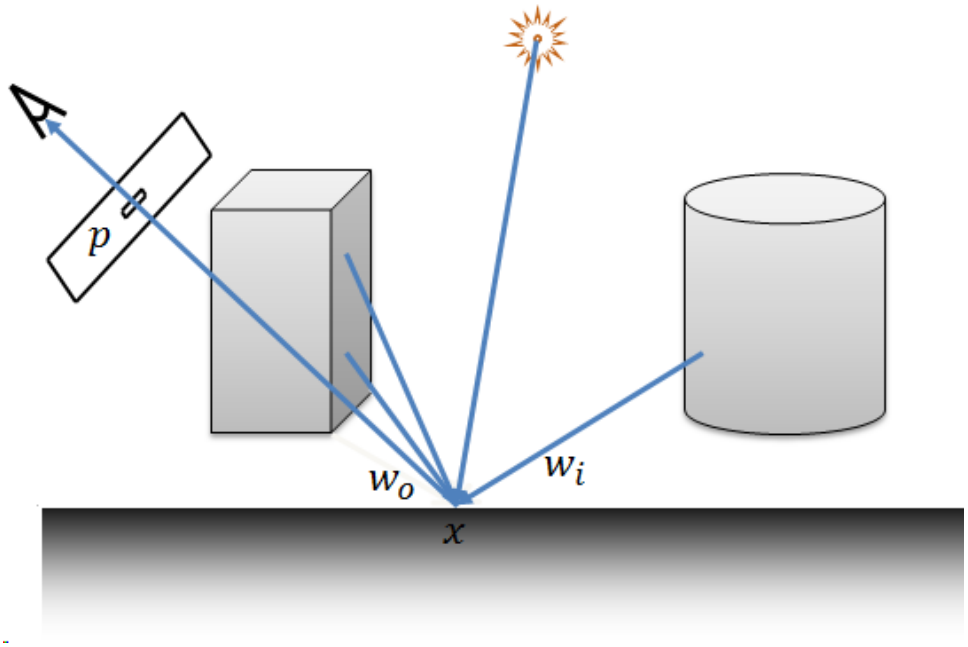


Figure 2.2: Computing Global Illumination

extend the PRT to handle high frequency effects by enhancing the basis [15, 16, 17, 18]. However, it is not easy to support dynamic lights and materials by using these kinds of PRT techniques.

Many extensions have been made to improve rendering performance while introducing bias. Two notable techniques among them are virtual point lights (VPLs) based radiosity [19] and photon mapping [20].

Keller [19] introduced a new kind of interactive approach which traces VPLs. Each VPLs has own shadow map which may have non negligible overhead when we use large number of VPLs for high quality illumination. Laine et al. propose an incremental method [21] that overcome the overhead by reusing the shadow maps. Also, Walter et al. solve the many light problem by using *light cut* [22].

Also, photon mapping has been extended to efficiently support an infinite number of photons given an available memory [23], stochastic rendering effects [24], and robust error estimation with a progressive rendering framework [25]. These techniques can be naturally combined with proposed method that focuses on handling massive models.

Another family of interactive GI, screen space approach, is well studied by limiting the interaction domain of indirect illumination to screen space. SSAO [26], SSDO [27], and ISPM [28] approximate GI using geometry information stored in depth buffer, however, they cannot compute interaction from objects behind occluder or out of viewing frustum. Reflective shadow map(RSM) [29] uses shadow maps for entry of indirect illumination point but the RSM ignores occlusion for the indirect illumination.

Recently, high performance GI approaches by taking advantage of massively parallel architectures such as GPU are actively studied. Wang et al. implement entire features including kd-tree building, photon tracing, and ray tracing on GPU [30] and show interactive performance with dynamic but relatively small scenes. Voxel cone tracing [31] gives high performance GI even in complex scene by tracing cones as approximations of bundles of indirect rays with sparse voxel octree. In this thesis, similar sparse voxel octree is used to compute indirect illumination as well. These GPU based techniques shows impressive performace for GI, however, they demonstrate rather low quality of rendering results because of a lot of

filtering and approximation.

2.3 Massive Model Rendering

There are orthogonal approaches for handling large-scale models: compact representation, cache-friendly, multi-resolution, etc.

2.3.1 Compact Representations

In the fields of computer graphics and visualization, mesh compression techniques have been well studied over the last decade and excellent surveys are available [32, 33]. Most previous mesh compression schemes were designed to achieve a maximum compression ratio as they were designed for archival use or for transmission of massive models [34, 35, 36]. They achieved this goal by encoding vertices [34, 37, 38], edges [39], and faces [40, 41, 42] in a particular order agreed upon by the encoder and decoder.

Most prior mesh compression techniques do not directly provide random access to the compressed meshes. Typically, in order to access a particular mesh element such as vertex, the whole compressed mesh must be sequentially decompressed to an uncompressed format that can support random access. In this section, we will focus on various compression techniques supporting random access on the compressed data.

Choe et al. [1, 2] proposed a single-resolution mesh compression method that supports selective rendering. Recently, Yoon and Lindstrom [3] proposed random-accessible compressed meshes for general applications requiring random mesh access. This method achieves up to a 20:1 compression ratio and improves the runtime performance of iso-contouring and layout re-computation. Several multi-resolution compression methods also support random access. Gobbetti et al. [4] proposed a compressed adaptive mesh representation of regular grids for terrain rendering. Also, Kim et al. [5] introduced a multi-resolution compression method for selective rendering.

Random access is one of the key components of the MPEG video compression format that allows users to browse video in a non-sequential fashion [43]. Particularly, the MPEG video codec method periodically inserts "intra pictures" as access points in the compressed scheme. Such intra pictures are compressed without using information from other frames. Then, subsequent frames are compressed by predicting the motion in between these intra pictures. For regular volumetric grids, wavelet-based compression methods [44, 45] that support random access have been proposed. Also, Lefebvre and Hoppe proposed a perfect spatial hashing method as an efficient random-accessible compressed image format [46].

Tree compression has been studied in many different fields [47]. These techniques include compressing the tree structure by linearizing the structure [48] and transforming the tree into a pre-defined tree [49]. However, these compressed trees do not support random access and do not preserve the layouts of the trees. There are relatively few research efforts on compressing BVHs. A BVH has two main components: BV information and indices to child nodes.

In order to compress the bounding volume information, fixed-rate quantization methods [50] are frequently used [51, 52, 53] , as applied to compress geometry of meshes [32]. Also, hierarchical encoding schemes were developed to further achieve a higher compression ratio and improve the compression quality [54, 55] by performing quantization of the bounding volume of a node within the region of the bounding volume of its parent node. These methods can support fast decoding and random access on

the quantized bounding volume information.

Many techniques assume a particular tree structure (e.g., complete tree) in order to completely remove most of cost related to encoding the tree structures [51, 56]. Recently, Lauterbach et al. [56, 57] introduced a Ray-Strip representation, which implicitly encodes a complete spatial kd-tree from a series of vertices. These techniques remove the cost of encoding tree structures. However, they may have low hierarchical culling efficiency, since they are incompatible with various optimized hierarchy construction methods [7, 58].

Compact sub-trees [59] reduce the space for tree structures given an assumption of the complete sub-trees. Lefebvre and Hoppe [60] employed local offsets to encode the location of child nodes given the pre-ordered layout of the tree. Jacobson [61] introduced a succinct tree, which supports arbitrary tree structures. The succinct tree supports random access with $O(1)$ time complexity and shows a compression ratio that is the asymptotic optimum. Proposed method also supports various kinds of tree structures and shows a compression ratio near the optimum in practice, while providing much faster runtime tree traversal than the succinct tree.

All of these techniques support random access on the compressed meshes, but assume a particular tree structure or a layout for the tree. Therefore, these methods may not be compatible with various hierarchy construction methods [62, 63, 58] optimized to achieve the high culling efficiency of BVHs.

A point-based approach [64, 65, 66] 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 [67, 68, 31]. It is well suited to GPU architectures thanks to its compact storage, and efficient traversal, and provides plausible rendering quality. Crassin [69] discussed some difficulties of sparse voxel octrees such as computing primary rays and detailed shadows that require a very high resolution of the voxels. Proposed approach addresses these issues by using a separated geometric representation and proposed augmented voxel representation for the shadow. VoxLOD [70] showed interactive color bleeding effects on massive models by using asynchronous voxel loading. Proposed rendering framework supports photon mapping and can generate more realistic outputs. A similar asynchronous loading for out-of-core voxels is used for the proposed method for providing better quality in a progressive manner, when the data bandwidth is available.

2.3.2 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 [71] assuming a particular cache size. Cache-oblivious techniques were shown to improve the cache coherence across different cache sizes [72]. In the field of ray tracing, there are a few techniques that maximize cache utilization by reordering rays [73, 74, 75]. 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 [76], but global illumination is not supported. Proposed method 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.

2.3.3 Multi-Resolution

Extensive research efforts have been put into designing various multi-resolution techniques for geometry [77], spatial hierarchy [78], and lighting [22]. Sparse voxel octrees [31] provide a multi-resolution scheme for all of them efficiently. In this thesis we extend this volumetric representation to provide interactive global illumination for massive models.

2.4 Utilizing CPU/GPU Hybrid Resource

Numerous algorithms are being implementing on GPU to take advantage of rapidly increasing GPU power. Meanwhile, there are some studies of maximizing overall performance by utilizing both CPU and GPU resources. HPCCD [79] well divides tasks based on characteristic of CPU/GPU architectures and shows high performance of continuous collision detection. Budge et al. propose generalized data management on CPU/GPU hybrid resources for path tracing [74].

2.5 Progressive Rendering and Adaptive Sampling

High quality rendering such as Monte Carlo ray tracing usually has high frame latency since it requires a lot of ray samples and computation cost. Therefore, progressive techniques for showing intermediate results with progressive refinement have been actively studied [80, 81, 82, 83, 23, 84, 85, 86]. In that sense, it is important feature that estimate error to assign more sample for reducing the error [87, 88, 89]. In addition, human perception is also considered for the adaptive sampling [90, 91]. Proposed method uses a kind of *saliency* metric [92] to determine sampling order because of its simplicity and efficiency.

Chapter 3. Random-Accessible Out-of-Core Compression

Ray-intersection test is a major computation of ray tracing based rendering algorithms. The intersection test requires geometric representation and its acceleration structure that mainly spend memory space. Out-of-core computation is often necessary when the size of the data exceeds the size of the main memory that causes significant degradation of runtime performance. Moreover, the typical data access pattern cannot be determined at the preprocessing time and is random at runtime that lowers I/O efficiency and cache utilization. Therefore, reducing the expensive data access time while supporting random access is essential for high performance rendering of massive models.

In this chapter, a random-accessible compression algorithm is proposed. As discussed in Chapter 2, random-accessible compression of geometric representation, especially for the meshes, is studied well. Therefore, we will focus on compression of acceleration structure, especially BVHs.

We present a novel compact BVH representation, *random-accessible compressed BVHs (RACBVHs)*, for various applications requiring random access on BVHs of massive models. We present a cluster-based layout-preserving BVH compression and decompression method supporting transparent random access on the compressed BVHs. We compress BVs of a BVH by sequentially accessing BVs in the BV layout of the BVH. During the compression, we decompose consecutive BVs into a set of clusters, each of which will serve as an access point for random access at runtime (Sec. 3.2). Our compression method preserves the original layout of a BVH and, thus, maintains high-cache utilization during the BVH traversal if the original layout maintains high cache-coherence. In order to allow various applications to transparently access the compressed BVHs, we provide a general BVH access API (Sec. 3.3). Given a BV node requested by the API, our runtime decompression framework efficiently identifies, fetches, and decompresses a cluster containing the data into an in-core representation that can efficiently support random access. Also, our method is easily extended to support parallel random access that can exploit the widely available multi-core CPU architecture. To demonstrate the benefits of our method, we implement two different applications, ray tracing and collision detection, by using our BVH access API (Sec. 3.4).

Overall, our approach has the following benefits:

1. **Wide applicability:** The provided BVH access API allows various applications to transparently access the compressed BVHs. Moreover, our BVH access API supports random access and does not restrict the access pattern of BVH-based applications. As a result, existing BVH-applications can be easily modified to take advantage of the benefits of our RACBVH representation.

2. **Low storage requirement:** Our RACBVH representation has up to a 12:1 compression ratio compared to an uncompressed BV representation. Also, we use random-accessible compressed meshes (RACMs) and achieve a similar compression ratio for the compressed meshes.

3. **Improved performance:** Our decompression method is fast and processes 4.3 M nodes per second by using a single CPU-core. Also, by selectively fetching and decompressing the small regions of the compressed BVHs and meshes requested by applications, we can reduce expensive data access time. As a result, we can achieve more than a 4:1 performance improvement on our tested applications

over using uncompressed BVHs and meshes.

Finally, we analyze our method and provide comparison over prior methods in Sec. 4.5.

3.1 Overview

In this section, we discuss issues that arise when using BVHs of massive models and give a brief overview of our approach to efficiently handle them.

3.1.1 BVHs of Massive Models

BVHs are widely used to accelerate the performance of intersection or culling tests in various applications. The leaf nodes of a BVH contain triangles of the original model. Each intermediate node of a BVH contains the BV information that encloses all the triangles located under the sub-tree rooted at the intermediate node. In this thesis, we use the AABB and a binary BVH due to its simplicity and the wide acceptance in various applications [58, 93].

High storage requirement: The storage requirement of BVHs can be very high for massive models consisting of hundreds of millions of triangles. For example, a simple AABB node representation has the following structure:

```
struct AABB {
  struct BV {
    Vector3f Min, Max;
  }; // bounding volume information.
  struct TreeStructure{
    Index Left, Right;
  }; // indices for child nodes.
};
```

Listing 3.1: **AABB Node Representation**

The **Min** and **Max** variables store the minimum and maximum extents of the AABB in the x, y, and z dimensions. Also, the **Left** and **Right** variables store the indices of child nodes in the case of intermediate nodes. Typically, a BVH is constructed until each leaf node has only one triangle. For the rest of the chapter, we assume that each leaf node of a BVH contains only one triangle and explain our method with this assumption. Later, we extend our method to support multiple triangles in leaf nodes in Sec. 4.5. In the case that leaf nodes contain only one triangle, the **Left** and **Right** variables of a leaf node store a triangle index of the triangle and a null index respectively.

This AABB structure requires 32 bytes per node. A model consisting of 100 million triangles requires about 6.4 GB of the main memory. Therefore, BVHs of massive models may not be loaded into the main memory and be accessed from the disk or through the network.

Random access pattern: Traversal on BVHs shows random access pattern for applications including ray tracing and collision detection. These applications typically take two inputs: two 3D objects for collision detection and one 3D object and a ray for ray tracing. The algorithm traverses BVHs of objects in the depth-first or breadth-first order as long as an intersection is detected between two inputs. In general, it is hard to predict the runtime access pattern on BVHs at the preprocessing time or to optimize

the access pattern at runtime. The data access time is often the main bottleneck of many applications that use BVHs of massive models.

Cache coherence: There have been several research efforts toward designing cache-coherent algorithms by reordering the runtime access patterns [94, 95] or by reordering the underlying data layout [96, 97]. These techniques reduce the number of expensive cache misses during random access on the data, since cache misses in various memory levels (e.g., L1/L2, main memory, and disk) are significantly expensive compared to the computation time [6].

Out-of-core techniques: Out-of-core techniques have been widely studied in order to handle massive models that cannot fit into the main memory [98]. These techniques aim at reducing expensive data access operations when dealing with massive models by only loading necessary data and performing local operations. However, due to the widening gap between data processing speeds and data access speeds [6], the time spent even on loading only the necessary data from the disk can be very expensive.

3.1.2 Our Approach

In order to efficiently access BVHs and improve the performance of various applications using BVHs, we propose a novel BVH compression and decompression method supporting random access. Our method has two main components: (1) a cluster-based layout preserving BVH compression and (2) a runtime decompression framework that transparently supports random access on the RACBVH representation without decompressing the whole BVH.

Cluster-based layout preserving BVH compression: We compress BVs of a BVH by sequentially reading BVs in the BV layout of the BVH. We choose our compression method to preserve the original layout of the BVH in order to achieve the high cache utilization which the original layouts may maintain. We decompose the original layout of the BVH into a set of clusters. We assign consecutive BVs in the BV layout to each cluster and set each cluster to have the same number (e.g., 4K) of BVs to quickly identify a cluster containing a BV node requested by an application at runtime. We compress each cluster separately from other clusters so that the clusters can be decompressed in any order.

Runtime BVH access framework: We define an atomic BVH access API supporting transparent and random access on the compressed BVHs. Our runtime BVH access framework first identifies a cluster containing a BV node requested by an application. Then, the runtime framework fetches and decompresses the cluster into an in-core representation. Based on our in-core representation, we can very efficiently support random access to applications. Our runtime BVH access framework is guaranteed to return the correct BV information of the requested data when applications access the compressed data via our BVH access API. We employ a simple memory management method based on a least recently used (LRU) replacement policy in order to handle massive models and their BVHs that cannot fit into the main memory.

3.2 Compression

In this section, we will explain our cluster-based layout preserving BVH compression method.

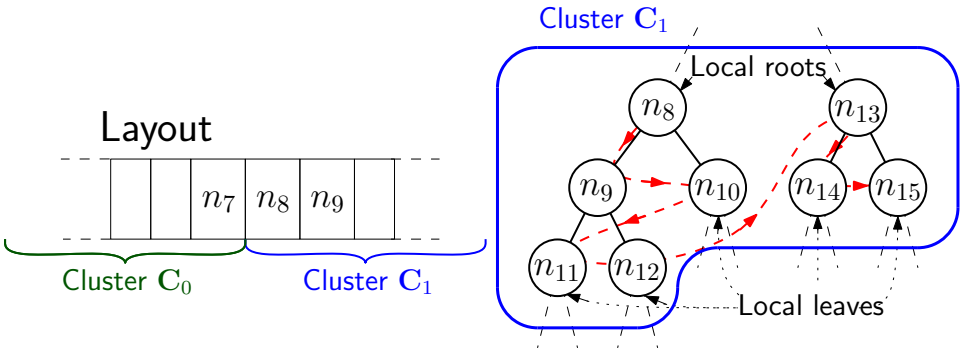


Figure 3.1: **Clusters from a BV layout:** This figure shows a BV layout on the left and a computed cluster, C_1, on the right. Red arrows indicate the BV layout of the BVH.

3.2.1 Layout Preserving BVH Compression

Our compression method sequentially reads and compresses BV nodes given in the format of Listing 3.1. As we read each BV node in the BV layout, we assign the BV node into a cluster. For clustering, we simply decompose consecutive BV nodes into a cluster, where each cluster has power-of-two nodes (e.g., 4K nodes). An index of a BV node increases sequentially as we compress each node. Then each BV node index can be encoded as a pair of indices, (C_i, l_i), where C_i is a cluster index to a cluster that the node is assigned to and l_i is a local index of the node within the cluster. Let us call the pair of indices a *pair index*. The pair index is also used in an earlier method of compressing massive meshes [99]. Note that each cluster may have one or multiple sub-trees (see Fig. 3.1). We call the root nodes of these sub-trees contained in a cluster *local roots* of the cluster. We also define *local leaf nodes* to denote leaf nodes of these sub-trees within each cluster. We define $Parent(n)$ to be a parent node of a node n. Also, *parent clusters* of a cluster c are defined to be clusters containing parent nodes of local roots of the cluster c. Similarly, we can define *child clusters* of a cluster.

For a new cluster, we initialize all the compression contexts. Therefore, each cluster can be decompressed in any order at runtime although we compress each cluster sequentially during the preprocessing time. We also define a *front* for a sub-tree located under each local root of a cluster during compression (see Fig. 3.2). As we compress each node, we add the node to the front and connect the node to its parent node, if the parent node is in the front. Once a node in the front is connected to its two child nodes, the node is deleted from the front. As a result, the front consists of BV nodes that are not yet connected to its two child BV nodes during compressing the cluster. We will use the front in order to compactly encode the tree structures of BVHs.

It is desirable that constructed clusters should contain BV nodes that are likely to be accessed together during the traversal of BVHs. Otherwise, we may have to load and decompress many clusters, which could lower the performance of applications. Since clusters are implicitly computed from original layouts of BVHs, BV nodes that are likely to be accessed together should be stored very closely in the layout of a BVH. Fortunately, there are a few layouts that satisfy this property. These layouts include the cache-efficient layouts of BVHs [97] and the van Emde Boas layout [96]. An example of clusters computed from different layouts is shown in Fig. 3.3. We will compare the performance of different layouts in Sec. 4.5.

We also propose using a dictionary-based compressor and decompressor [100] to achieve a high compression ratio and, more importantly, a fast decompression performance for efficient random access

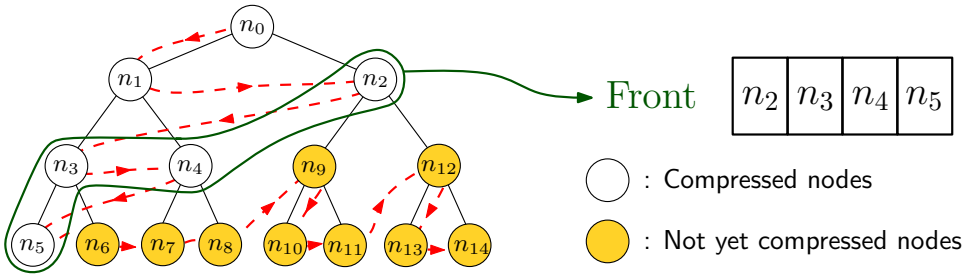


Figure 3.2: **A Front during Compression:** This figure shows a front of a sub-tree during compression. We use the front to compactly encode the tree structure of BVHs.

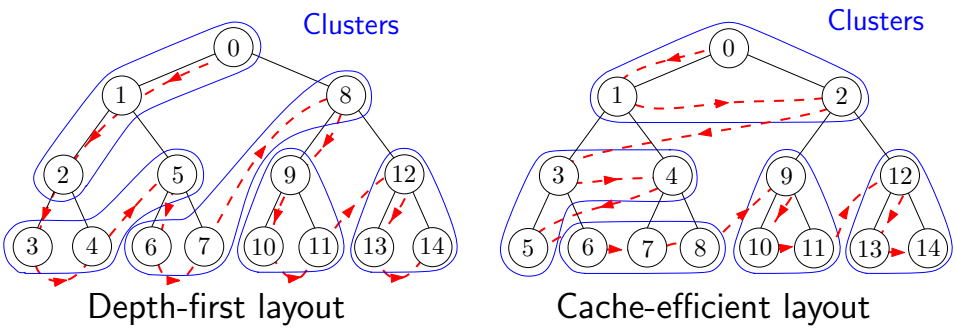


Figure 3.3: **Clustering with Different BV Layouts:** This figure illustrates different layouts of a BVH and constructed clusters by assigning the fixed number (e.g., 3) of consecutive BV nodes into clusters.

on our RACBVH representation. A pseudo-code of our compression method is given in Algorithm 1, for the clarity of explaining our method.

3.2.2 Encoding Bounding Volumes

The **Min** and **Max** variables of the BV node representation (Listing 3.1) store the minimum and maximum extents of an AABB node. We first quantize each component, i.e., x, y, and z, of the **Min** and **Max** of each BV node. We use a fixed-rate quantized method [50] for those components based on the root bounding volume of a BVH. We quantize the components conservatively to ensure that the quantized BV still encloses all the triangles of the original BV. Then we further compress the quantized **Min** and **Max** values based on the BV information of $Parent(n)$ of the node n currently being encoded. To do that, we predict two child BVs of $Parent(n)$ given the BV of $Parent(n)$. Then we only encode the difference between the predicted and actual BVs using our dictionary-based encoder. For the prediction, we partition the parent BV into two child BVs by dividing the longest axis of the BV into half. We find that this simple *median prediction method* works well in our tested benchmarks.

It is possible that a cluster containing a parent node $Parent(n)$ of a node n may be different from a cluster containing the node n. In this case, we cannot assume that the BV information of the $Parent(n)$ is available when decompressing the node n. Our runtime framework addresses this problem by guaranteeing the existence of the $Parent(n)$ by loading the cluster containing the $Parent(n)$. This will be discussed more in Sec. 3.3.

3.2.3 Encoding Tree Structures

The **Left** and **Right** of intermediate BV nodes (Listing 3.1) contain two child indices and represent the tree structure. However, the **Left** of a leaf node has a triangle index contained in the leaf node. We will describe our compression method for child indices of intermediate nodes and, next, for triangle indices of leaf nodes in the next two sections.

Encoding Child Indices

Initially, we attempted to directly encode child indices of the **Left** and **Right**. However, we found this approach shows poor compression results. This is mainly caused by the fact that there are many factors (e.g., layout types and tree structures) affecting child indices of a node and it is difficult to account for these factors when predicting child indices. Instead of encoding child indices given a node, we propose to encode a parent node index from a node n currently being encoded. The main rationale behind this design choice is that the tree structure above the node n is available at the time of encoding or decoding the node n. Therefore, encoding a parent node index is simply choosing a node from the already encoded tree structure instead of predicting a tree structure below the node n.

Note that either the **Left** or **Right** index of a parent node $Parent(n)$ is equal to the node index n_{idx} of the node n. Therefore, if we encode a parent node index of the node n, then, we can access its parent node $Parent(n)$ and fill either the **Left** or **Right** indices of $Parent(n)$ with the node index n_{idx} when we decompress the node n. To indicate whether the node n is the left or right child of its parent, we encode an additional bit.

Our BV node does not directly store any information about its parent node index. Fortunately, this information can easily be constructed during compression. To do this, we maintain a hash table. For a node n specified by a node index n_{idx}, our BV representation gives us indices of two child nodes, **Left** and **Right**, of the node n. We simply construct two hash map elements connecting a key of each child index to n_{idx} of the node n.

As we encode each node n, we attempt to find its parent node index by querying its node index n_{idx} to the hash table. If we can find the node index n_{idx} and, thus, its parent node index from the hash table, then we can find the parent node $Parent(n)$ specified by the parent node index among nodes stored in the front according to the definition of the front described in Sec. 3.2.1. For example, when we encode a node of n_6 in the example of Fig. 3.2, we first attempt to find its parent node n_3 using the hash table by querying the node n_6. Since the hash map element that connects n_6 to n_3, is already inserted when n_3 is encoded, we can find its parent node n_3. Also, the node n_3 is the second element in the front. Therefore, we encode 2. Then we insert n_6 to the front and remove n_3 from the front since the node n_3 is connected to its two child nodes. Finally, we add two hash map elements that connect two child nodes of n_6 to the node n_6.

Note that the number of nodes stored in the front is typically much smaller than the number of nodes in the BVH. Therefore, we can compactly encode the parent node index by encoding its position in the front. The average size of the front with the cache-efficient layouts is 13 when we assign 4 K nodes for each cluster in our tested models. We can compute the position of the parent node in the front and update the front in a constant time by using a dynamic vector for nodes in the front.

If the node n currently being encoded is the root node of the BVH or a local root of a cluster that the node n is assigned to, then we cannot find its parent node index by querying a node index n_{idx} to the hash table. The case of the root node can easily be identified and addressed. In the case of local roots

of a cluster, the parent nodes of those local roots are located in another cluster. Therefore, the front of the cluster that is currently being encoded does not have any information about the parent nodes of the local roots while the cluster is being compressed. One naive solution is to directly store the parent node index, which requires storing many bits. Instead, we encode the parent node index by decomposing it into a pair index of (C_i, l_i). Then we encode C_i among the parent clusters of the cluster that is currently being encoded and, then, encode l_i. We found that this method gives a high compression ratio since the number of parent clusters is typically very small (e.g., 2 on average for cache-efficient layouts).

Local leaf nodes: Given our compression scheme, we cannot compute the child indices of local leaf nodes of a cluster without decompressing its child clusters at runtime. This is because these child indices of local leaf nodes will be computed when processing child clusters containing the child nodes specified by the child indices. However, loading the child clusters of a cluster at runtime can significantly increase the working set size of applications and, thus, decrease the performance of applications. To avoid this problem, we directly encode left and right child indices stored in **Left** and **Right** only for local leaf nodes. To compactly encode left and right child indices of local leaf nodes, we employ a simple delta encoding. Given a child index that we are going to encode, we compute the difference between the index and a previously encoded index. Then we encode the difference using our dictionary-based encoder.

Encoding Triangle Indices

The **Left** index of a leaf node contains a triangle index. Since a BVH is constructed by spatially partitioning the triangles of a mesh, triangles stored in neighboring leaf nodes are highly likely to be spatially close and may even share edges between them. We use this observation to design two different compression methods that explicitly or implicitly exploit the connectivity of a mesh between leaf nodes.

Explicit utilization of the mesh connectivity: Our first method explicitly uses the underlying mesh connectivity. We encode triangle indices when we encounter leaf nodes while sequentially accessing BV nodes in the BV layout. For each encoded triangle index of a triangle, we compute three indices of its three neighboring triangles and store them in a cache. Then for a triangle index of a next leaf node, we attempt to encode the triangle index among three triangle indices stored in the cache. Otherwise, we encode the triangle index. We found that this method works well and about half of triangles indices of leaf nodes can be encoded by the small cache holding three previous neighboring triangle indices. However, one downside of this approach is that, to compute each triangle's neighbors, the mesh connectivity must be constructed during decompression, which is inefficient at runtime.

Implicit utilization of the mesh connectivity: In order to support efficient decompression and high compression ratio, we propose to use a simple delta coding method. We found that the difference between the previous and the current triangle indices is typically small when layouts of a BVH and a mesh have high coherence. Therefore, we encode the triangle index difference with our dictionary-based encoder if the difference is less than a small threshold (e.g., 10). We found that about 80% of the differences are within the threshold of 10 when using cache-efficient layouts. Otherwise, we encode the triangle index. This compression method implicitly utilizes the underlying layouts of both BVHs and meshes having the high spatial coherence between neighboring nodes and triangles. As a result, this implicit method requires fewer bits per node (bpn) and decompresses much more quickly.

Meta file: In order to support random access on the compressed BVH at runtime, we construct a meta file as we compress a BVH. The meta file has a starting address of each cluster in the compressed BVH and the number of BV nodes assigned to each cluster. Since the meta file takes only minor memory space (e.g., less than one MB), we store the meta file without any compression. Note that this meta file is constructed progressively as we compress a BVH in one pass.

3.2.4 Dictionary-based Compression

We employ one more layer of compressing the data by using a dictionary-based compressor to improve the compression ratio while achieving high decompression performance. Particularly, we use variations of the LZW method [100, pages 199–208] based on a simple dictionary. We choose the LZW method since it can quickly detect and compress repeating patterns in the sequences of symbols. For each different compression context, we initialize dictionary entries with all the symbols that the compression context can have. Then we allow adding a new entry consisting of a combination of symbols if the entry is not in the dictionary.

For the compression contexts of the delta encoding and the difference encoding for the quantized BVs, we further optimize our LZW method since these compression contexts show different characteristics compared to other contexts. In these compression contexts, we do not initialize the dictionary with symbols and start with an empty dictionary since there are too many possible symbols and we found that most of those symbols do not appear during compressing a cluster. We add a new symbol if we encounter the symbol during compression. However, we do not add new entries consisting of a combination of symbols since we found that there are not many repeating patterns in these compression contexts.

3.2.5 Random-Accessible Compressed Meshes (RACMs)

We employ the RACM representation [3] to further reduce the storage requirement of meshes, which are used together with BVHs for various applications. We use OpenRACM library [101] available online for computing RACMs and employ the provided mesh access API to access the compressed meshes. The original RACM method used the arithmetic encoder [102] to achieve a high compression ratio. We found that our dictionary-based encoder shows much higher runtime performance than the arithmetic encoder. Therefore, we modify the RACM method to use our dictionary-based method. We will provide more detail comparisons between our dictionary-based encoder and the arithmetic encoder in Sec. 3.5.2.

3.3 Runtime Decompression Framework

In this section, we present our runtime decompression framework that transparently supports random access on our RACBVH representation.

In-core BV representation: As we decompress BV nodes requested by applications, we store the decompressed BVs in the main memory in the format of the simple BV representation shown in Listing 3.1. The main benefit of using this in-core BV representation is that it can directly support random access without any computation overhead.

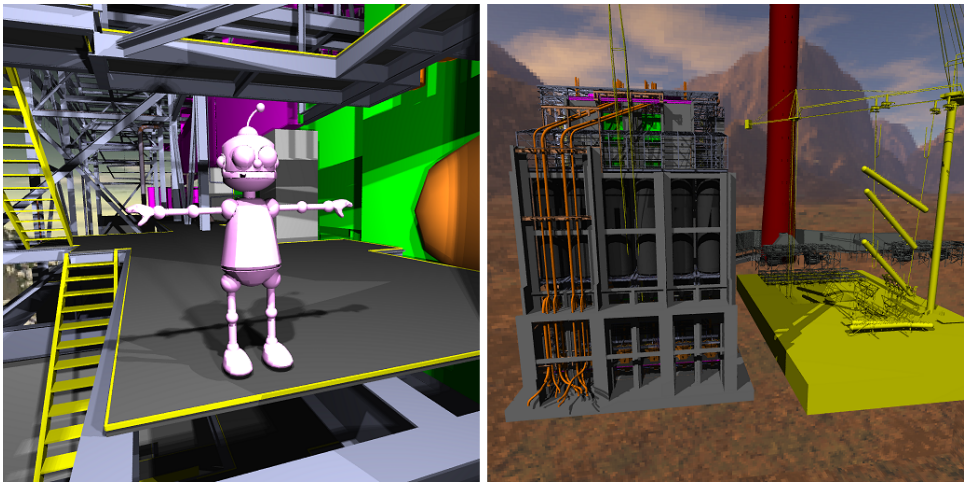


Figure 3.4: **Hubo and Power Plant Models:** The Hubo robot model (16 K triangles) is placed in the upper left corner of the power plant model (13 M triangles). The entire power plant model is shown on the right. We improve the performance by a factor of 2:1 for collision detection using our RACBVH representation.

3.3.1 BVH Access API

In order to allow various applications to transparently access our RACBVHs, we provide the following BVH access functions:

Index **GetRootIndex** (void): Return an index of the root node of a BVH.

BV & **GetBV** (Index n_{idx}): Return the BV information specified by the node index n_{idx}.

bool **IsLeaf** (Index n_{idx}): Return whether a BV node specified by the node index n_{idx} is a leaf.

Index **GetLeftChildIdx** (Index n_{idx}): Return an index of the left child node of a BV node specified by the node index n_{idx}. We also define a similar function for the right child.

Index **GetTriangleIdx** (Index n_{idx}): Return a triangle index stored at the leaf node specified by the node index n_{idx}.

Based on this BVH access API, we can traverse a BVH in a hierarchical manner or access any BV nodes in an arbitrary order. Also, we can support more advanced BVH access methods like front-based traversal [103] or hierarchy traversal from an arbitrary entry point [104], based on these atomic BVH access API.

3.3.2 Runtime BVH Access Framework

Our runtime data access framework first reads the meta file. Suppose that an application requests a BV node by calling the **GetBV** $(\cdot)$ function. Then, the runtime data access framework identifies a cluster containing the requested data, decompresses it into our in-core BV representation, and returns the data to the application. Since clusters have consecutive power-of-two BV nodes, we can compute a cluster index by performing a few bit operations to a given node index. Once a cluster index is computed, we refer to the meta file to acquire a starting address of the cluster on the compressed BVH and, then, we decompress the cluster.

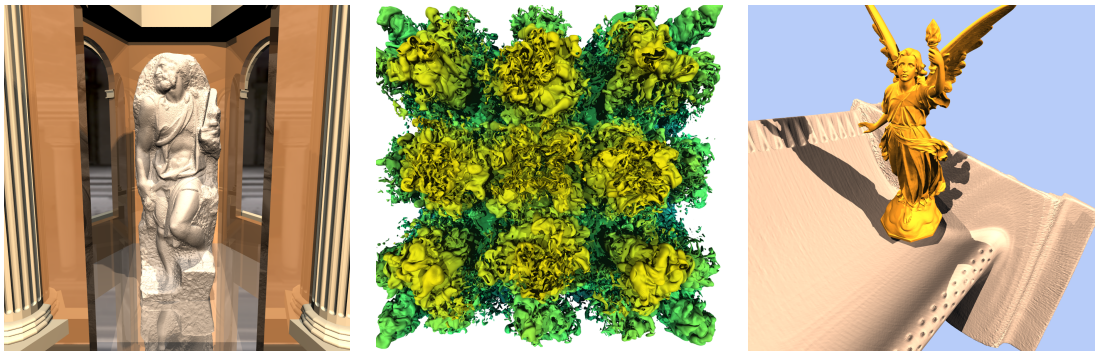
Decompressing a cluster: The process of decompressing a cluster is symmetric to the process of compressing it. As we read the compressed data of a cluster, we reconstruct the tree structures represented by the **Left** and **Right** indices and the BV information. The tree structure is reconstructed without extracting any information from other clusters. On the other hand, the BV information of a node is reconstructed by extracting the BV information from its parent node given our hierarchical BV compression method. For the local roots of a cluster, we cannot reconstruct the BV information of these local roots and their sub-trees if the BV information of parent nodes of these local roots is not available while decompressing the cluster. We call these nodes that we cannot reconstruct the BV information at the time of decompressing the cluster *incomplete nodes*. We also call all the rest of the nodes *complete nodes*. We use the most significant bit of **Right** to indicate whether a node is complete or not at runtime. For complete nodes, we can reconstruct their BV information as we decompress a cluster since their parent BV nodes are available. For each incomplete node, we decompress the differences between the predicted and actual BVs and, then, store them at the **Min** and **Max** variables. Also, we store the decompressed parent index at **Left** instead of computing the left and right indices of the node and storing them at **Left** and **Right**. We lazily construct the BV information and tree structures of incomplete nodes based on the data stored in the in-core BV representation.

BV completion: If the BV node requested by **GetBV**$(\cdot)$ is complete, we can simply return the stored in-core BV representation to the application. If the node is incomplete, then we search for its local root, n_l. This operation can easily be performed by using the parent node index stored in the **Left** of the incomplete node n. Then, we force to load a cluster that the parent node of the local root n_l is assigned to, if the cluster is not yet loaded. Note that, in order to load a cluster for the BV completion, we may need to recursively load another cluster. Fortunately, during the typical hierarchical traversal of a BVH, a cluster would have been already loaded if a node located in the cluster's subtree has already been accessed. Once we obtain the BV information of a parent node of the local root n_l, we can reconstruct the BV information based on the BV difference stored at the **Min** and **Max** of these incomplete nodes. We also compute the left and right child indices for the **Left** and **Right**. We complete the BV information of all the incomplete nodes of the sub-tree rooted at the local root node n_l within the cluster that n_l is assigned to. Then we set those incomplete nodes to be the complete nodes.

We explained how our runtime framework handles a call of **GetBV** $(\cdot)$. Our runtime framework also performs the similar procedure for all the other functions except for **GetRootIndex** (), which is processed very easily. Note that all the BVH access functions except for **GetRootIndex** () are called with a parameter of a node index, n_{idx}. These functions first find the requested node in the same manner of processing **GetBV** $(\cdot)$. Once the node is identified, we simply return the requested information (e.g., BV, child index, or triangle index) of the node to the application.

3.3.3 Memory Management

In order to handle massive models whose BV nodes and meshes cannot fit into the main memory, we employ a simple memory management. Given a pre-allocated memory pool of a size specified by the user, we perform the memory management at the granularity of clusters. We maintain a LRU-list of clusters that have been accessed by our BVH access API. Since updating the LRU-list of clusters is expensive, we update the list only when we encounter a new cluster that is different from the previously accessed cluster. Note that clusters located in lower BVHs are accessed less frequently than clusters located in upper portions of BVHs during the BVH traversal. Our simple LRU-based replacement method implicitly



(a) St. Matthew scene (b) Iso-surface model (c) Lucy and CAD turbine models

Figure 3.5: The left and middle images show results of ray tracing using our random-accessible compressed bounding volume hierarchies (RACBVHs) of St. Matthew model consisting of 128 M triangles and an iso-surface model consisting of 102 M triangles. The right image shows a frame during a rigid-body simulation using collision detection between two models including the Lucy model consisting of 28 M triangles. By using RACBVHs, we can reduce the storage requirement by a factor of 10:1 and, more importantly, improve the performance of ray tracing and collision detection by up to a factor of four over using uncompressed data.

considers this factor since clusters located at upper portions of BVHs are more likely to be re-visited during the BVH traversals and, thus, they are less likely to be unloaded.

Pre-loading the RACBVHs: Our RACBVH representations associated with the RACM representations require much less storage than uncompressed BVH and mesh representations. Therefore, it is possible to sequentially pre-load all the compressed data of massive models into 2–4 GB sized main memory of commodity hardware and access those data without the expensive disk I/O access at runtime. Since the sequential access to the disk during the pre-loading is much faster than random access [6], the pre-loading can be done quickly. We will show the performance of tested benchmark applications with and without pre-loading the compressed data in Sec. 3.4.2.

3.4 Results

We have implemented our compression method, out-of-core runtime decompression framework, and benchmark applications on a 3.0 GHz Intel Core2 Extreme-PC that has a quad-core CPU, with 32-bit WindowsXP, 4 GB of RAM, and a SATA disk drive having a sequential reading performance of 62 MB per second. We perform various tests by using a single thread, unless mentioned otherwise. We set our runtime decompression framework to use no more than 2.3 GB of the main memory to cache uncompressed data. Our compression method works with any layouts or BVH construction methods.

Benchmark models: We have tested our method with various benchmark models including the St. Matthew model (128 M triangles, Fig. 3.5(a)), the Lucy model (28 M triangles, Fig. 3.5(c)), a CAD turbine model (1.8 M triangles), the power plant model (13 M triangles, Fig. 3.4), a Hubo robot model (16 K triangles), and an iso-surface model (102 M triangles, Fig. 3.5(b)) extracted from a scientific simulation. More detail information about our benchmark models is shown in Table 4.1.

Table 3.1: Benchmark Models and Compression Results

Model	**Tri.**	**Vert.**	**Size (MB) of uncompressed**		**Size (MB) of compressed**		**Compressed bpn**		**Compressed ratio**	
	(M)	**(M)**	**BVH**	**Mesh**	**BVH**	**Mesh**	**Tree**	**BV**	**BVH**	**Mesh**
St.Matthew	128	64	7811	3933	814	320	8.66	18.0	9.6:1	15.4:1
Iso surface	102	51	6254	3912	709	288	8.75	20.3	8.8:1	17.0:1
Lucy	28	14	1684	1008	193	73	8.70	20.7	8.7:1	17.3:1
Power plant	13	11	778	389	66	55	9.22	12.3	12:1	8.8:1
CAD turbine	1.8	0.9	108	67	13	5.5	9.19	21.6	8.3:1	15.2:1

Model complexity, compression results, and compression ratios for our benchmark models are shown. ***Tree*** indicates tree structures in BVHs and ***BV*** indicates the BV information. Our method achieves up to a 12:1 compression ratio over the uncompressed BV representation shown in Listing 3.1. We use 16 bits for the quantization of the BV information, 4 K node cluster size, and cache-efficient layouts. ***bpn*** stands for bits per node.

3.4.1 Compression Results

We construct BVHs of benchmark models and store them in a cache-coherent manner, particularly, cache-efficient layouts [97]. We choose this layout since it shows a high compression ratio and, more importantly, the best runtime performance among the tested layouts, as we will see later. We quantize the BV information using 16 bits. In this configuration, we are able to achieve 27.5 bits per node (bpn) on average for our benchmarks. For the St. Matthew model, our compression method spends 18.0 bpn to encode the BV information and 8.66 bpn to encode tree structures. Compared to the uncompressed AABB representation (Listing 3.1), we achieve up to a 12:1 compression ratio in our benchmark models. In addition, we achieve about a 13:1 compression ratio for the tested meshes by using RACMs. Overall, we achieve a 10:1 compression ratio on average by using RACM and RACBVH representations compared to using uncompressed BVHs and meshes. Please refer to Table 4.1 for more detail compression results.

Decompression performance: Our compression method can compress 0.4 M nodes per second. On the other hand, our decompression method can process 4.2 M nodes (= 135 MB) per second when we decompress clusters sequentially. The one order of magnitude faster decompression performance is mainly because we do not need to construct various data structures like the hash table that are only needed during the compression process. Because of the fast decompression performance, our method can improve the performance of many applications that use BVHs of massive models.

3.4.2 Benchmark Applications

We implement two different applications, ray tracing and collision detection, to verify the benefits of our proposed method. We choose these two applications because they have different access patterns. Ray tracing typically traverses larger portions of BVHs while collision detection accesses smaller and more localized portions of BVHs. We implement these two applications with the proposed BVH access API. For comparison, we can also set our BVH access API to access uncompressed BVHs stored in the format of our in-core BV representation on the disk without changing any application code. Note that applications get the original un-quantized BV information from the stored uncompressed BV nodes.

Therefore, applications get tighter BV information, which can achieve higher culling efficiency during the BVH traversal, compared with our quantized BVs of the RACBVHs.

Ray Tracing

We implement a BVH-based ray tracer [93, 105] for the distributed ray tracing [106]. We construct BVHs optimized with the surface-area heuristic (SAH) [62], which is a well known method for constructing acceleration hierarchies that maximize the performance of ray tracing. We also use the projection method [62] for fast triangle-ray intersection tests. To do this, we compute various quantities (e.g., best projection planes, triangle normals, etc.) on the fly as we read and decompress the RACM representation. We use 512 by 512 image resolution for the image generation tests.

We first test our BVH-based ray tracer only using primary rays with small models (e.g., the Stanford bunny model). Our single-thread BVH-based ray tracer can process 1 million rays per second. Note that since our ray tracer handles massive models that cannot fit into the main memory, it runs at an out-of-core mode, which requires another data management layer for accessing massive data. We expect that the ray-packet methods [62] can further improve the performance of our ray tracer.

We test our method with one light source and no reflection when we look at St. Matthew model as shown in Fig. 3.5(a). In this configuration, many of the rays are coherent. By only using RACMs with the uncompressed BVHs, we achieve a 1.5:1 performance improvement over using the original uncompressed data. However, by using both RACBVHs and RACMs, we are able to achieve a 2.6:1 performance improvement on average over using the uncompressed data. We also test the pre-loading of the compressed data. In this case, we spend 18 seconds on sequentially reading and pre-loading 1.1 GB of the RACM and RACBVH. We observe a 4.4:1 performance improvement over the original uncompressed data at runtime. Note that we cannot pre-load the uncompressed data since it takes about 11.7 GB.

We also measure the rendering time during ray tracing the St. Matthew model in the scene setting shown in Fig. 3.5(a). In this scene, we use 7 point light sources, reflections, and 3 by 3 stratified sampling; this configuration results in many incoherent rays. Initially, the camera is located far from the St. Matthew model and, then, it zooms to the face of the St. Matthew. We observe a 1.5:1 and a 2.2:1 performance improvement on average by using RACMs with uncompressed BVHs and by using both RACMs and RACBVHs respectively over by using the original uncompressed data. Fig. 3.6 shows the ray tracing time for each of these three methods. The performance improves by using RACBVHs and RACMs because the I/O time is reduced due to the fast decompression performance and selective decompression during the BVH traversal.

We also measure the performance of ray tracing the iso-surface model shown in Fig. 3.5(b). We use 5 point light sources and 3 by 3 stratified sampling. In this configuration, we use both RACM and RACBVH representations. We improve the performances by a factor of 2.6:1 and 1.9:1 on average over using uncompressed data with and without the pre-loading the compressed data respectively.

Collision Detection

We implement collision detection and integrate it into a rigid-body simulation [107]. Collision detection identifies colliding regions between two models by traversing BVHs. We create a benchmark where we drop the Lucy model on top of the CAD turbine model (see Fig. 3.5(c)).

We measure the collision detection time during a simulation that runs for 215 simulation steps. In the tested benchmark models, the sizes of compressed RACBVHs and RACMs are 198 MB and 78 MB

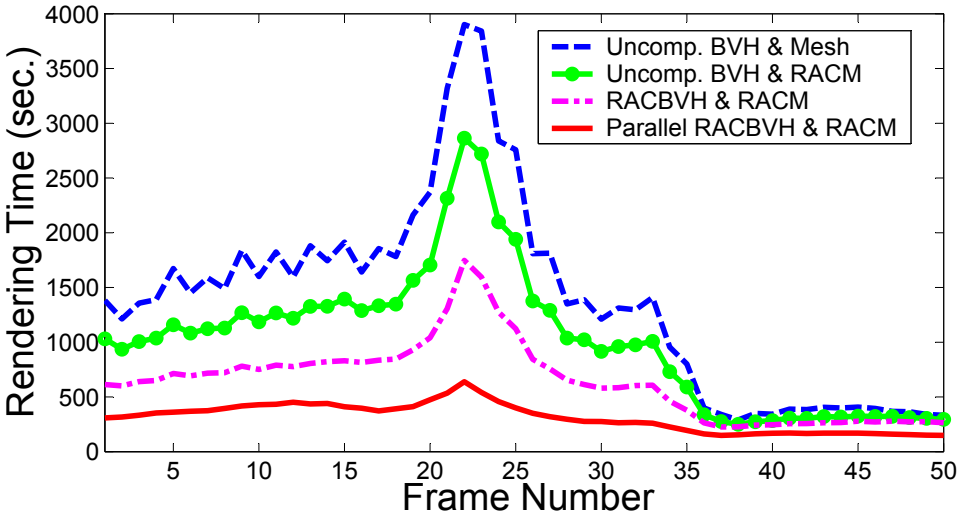


Figure 3.6: **Performance of Ray Tracing St. Matthew Scene:** This graph shows the rendering time during ray tracing the St. Matthew scene shown in Fig. 3.5(a). By using random-accessible compressed meshes (RACMs) without the random-accessible compressed BVHs (RACBVHs), we achieve a 1.5:1 performance improvement over using the original uncompressed data. By using both RACBVHs and RACMs, we achieve a 2.2:1 runtime performance improvement on average. Also, by using four threads with RACBVHs and RACMs (labeled as Parallel RACBVH and RACM), we achieve an additional 2 times improvement and thus improve the performance by a factor of 4.4:1 over using the original uncompressed data. Accessing uncompressed data with multiple threads does not improve the performance because the disk I/O access is the main bottleneck.

compared to 2.8 GB of the original meshes and BVHs. Since RACMs and RACBVHs can be stored in main memory, we pre-load RACMs and RACBVHs, if they are used. Pre-loading RACMs and RACBVHs take about 4 seconds. We improve the performance of the collision detection on average by a factor of 1.4:1, 2.5:1, and 2.9:1 using only RACMs, only RACBVHs, and both RACMs and RACBVHs respectively over using uncompressed data. On average, collision detection accessing RACMs and RACBVHs takes 5 milliseconds (ms) per step. We also test and measure performance of another collision detection benchmark consisting of the power plant and Hubo models (see Fig. 3.4). In this case, we achieve a 2.1:1 performance improvement over using uncompressed data.

3.4.3 Parallel Random Access

Our compression and decompression methods can be extended to support parallel random access and exploit the widely available multi-core CPU architecture. To enable the parallel random access in our method, we use a lock to avoid completing the same node from multiple threads. Also, we use a pseudo LRU method, the second-chance algorithm [108] that reduces locking the same node in the LRU list during the update of the list. We test the performance of our method with the ray tracing benchmark. For our parallel ray tracer, we simply divide the image plane into multiple units and assign each unit to a thread. Then, each thread generates and processes primary and secondary rays while accessing compressed data. The memory pool that caches decompressed data is shared among threads. We use four threads and pre-load all the compressed data. We measure the rendering time during the ray tracing of the St. Matthew model with incoherent rays in the scene setting described in Sec. 3.4.2.

Table 3.2: Cluster Size vs. Performance

Cluster size	256	512	1K	2K	4K	8K	16K
Compression ratio	9.3	9.7:1	9.8:1	9.7:1	9.6:1	9.4:1	9.2:1
RT w/ coherent rays (s)	352	339	351	329	319	318	316
RT w/ incoherent rays (s)	2,220	2,242	2,278	2,307	2,339	2,422	2,470
Collision detection (ms)	4.69	4.74	4.93	5.02	5.34	5.76	6.13

This table shows the compression ratio and ray tracing time (RT) of the St. Matthew model and collision detection time, as a function of the cluster size.

When we use the original data, the performance decreases by a factor of 1:1.8 by using four threads over using a single thread. Since four different threads request more disk I/O accesses, these more I/O accesses worsen the disk seek performance and lower the performance of ray tracing. On the other hand, our method achieves a 2.1:1 performance improvement over using a single thread. Therefore, our method achieves a 4.4:1 performance improvement for ray tracing the St. Matthew model over using the original data. This higher scalability of our method is achieved by removing the expensive and low-performing disk I/O access. The performance of ray tracing the St. Matthew model when using four threads is shown in Fig. 3.6.

3.5 Analysis and Comparison

In this section, we analyze the performance of our method and compare its performance with other related techniques.

3.5.1 Analysis

The performance of our method is affected by several factors. We discuss them in terms of their impact on the runtime performance of applications. We report various results by using a single CPU core.

Cluster size: In order to see how the compression ratio varies as a function of cluster sizes, we compute different versions of the RACBVHs of the St. Mathew model with different cluster sizes ranging from small to large sizes: 0.5K, 1K, 2K, 4K, 8K and 16K nodes (see Table 3.2). Note that encoding BVs and parent nodes in the front requires more bits as the cluster size becomes larger, since BV prediction errors vary more and the front's size is larger. On the other hand, encoding parent nodes of local root nodes and child indices of local leaf nodes require fewer bits as the cluster size becomes larger, since there are fewer local roots and leaf nodes. Given these two factors, we achieve the highest compression ratios when the cluster size is 1 K nodes. However, the compression ratios are rather stable in the tested cluster sizes.

We also test the performance of ray tracing and collision detection with the different cluster sizes. We first measure ray tracing time of St. Matthew model with coherent rays. The performance improves even when we use very large clusters, since the disk I/O performance improves by reading larger clusters and loaded clusters may be reused during the processing of many coherent rays. However, when we perform ray tracing with incoherent rays, the performance worsens as the cluster sizes become larger, since we have to frequently load and unload clusters, caused by incoherent data access pattern. Also,

the performance of collision detection goes down as the cluster size is larger, since we have to load larger clusters, where most of nodes may not be accessed given the very localized data access pattern of collision detection. Although it is very hard to conclude which cluster size is the best across different applications, we found that clusters with 1 K to 4 K nodes show high performances among the tested applications.

Layouts: We also compare the performance of applications with different layouts of BVHs. We compute depth-first, van Emde Boas [96], and cache-efficient layouts [97] of our tested benchmark models. Surprisingly, the best compression results are achieved with the depth-first layouts. However, we found that cache-efficient layouts show the best runtime performance, followed by van Emde Boas, and depth-first layouts. This is mainly because of the high spatial and cache coherences that the cache-efficient layouts maintain. Performance results with different layouts of ray tracing the St. Matthew model are shown in Table. 3.3.

Number of triangles per leaf node: Our simple BV representation (Listing 3.1) can be extended to store multiple triangles by using a global triangle index list. If a leaf node contains multiple triangles, we store indices of these triangles consecutively in the list. Then the **Left** and **Right** simply contain the starting and ending positions of these triangle indices in the list. For efficient intersection tests, we compute a sub-BVH for triangles contained in a leaf node on the fly. Also, we cache the computed sub-BVHs of leaf nodes and apply our memory management method to these cached data. We compare the performance of ray tracing the St. Matthew model with different number of triangles per leaf node: 1, 4, 16, and 128. As the number of triangles per leaf node increases, the size of BVHs decreases. The performance peaks when we assign 16 triangles to each leaf node and, in this case, our method shows a 1.8:1 performance improvement over using the uncompressed data.

Extensions to other types of BVs and k-ary BVHs: At a high level, our compression method compresses a BV by first quantizing the BV and predicting its two child BVs by assuming a most likely partitioning plane during the BVH construction. This idea can be applied easily to other types of BVs such as spheres and OBBs, since these BVs and AABBs are constructed using a similar hierarchical partitioning scheme. Also, our method could be extended to support k-ary tree structures. During encoding child indices, we explicitly encode extra information to indicate whether a node is the left or right node of its parent node. In the case of k-ary tree structures, we can simply encode a position of a node among k different child nodes.

Limitations: Accessing the RACBVH and RACM representations has some overhead. For example, we have to perform a few bit operations computing cluster indices for each **GetBV** $(\cdot)$ call. Also, the compressed BVs may give fewer tight extents and cause more intersection tests for applications because they are quantized more conservatively than the original uncompressed BVs. Therefore, when all the data needed to perform an application are already in main memory, our method might lower the performance. To verify this, we perform the ray tracing of a simplified St. Matthew model consisting of 8 M triangles, whose BVH and mesh can be stored in main memory. We load all the data into main memory. In this case, we observe 1% more intersection tests and, thus, about a 1% lower performance using our method than using the uncompressed data at runtime. Also, we perform the collision detection of the rigid-body simulation by using simplified models that can be stored in main memory. We observe 3% more intersection tests and 3% lower performance using our method than using the uncompressed data.

Table 3.3: Layouts vs. Performance

Layouts	Size of RACBVHs(MB)	Compression ratio	Rendering time(sec.)
Depth-first	776	10.1:1	334
van Emde Boas	777	10.0:1	332
Cache-efficient	814	9.6:1	319

This table shows the size of our RACBVHs with different layouts. The compression ratio is computed over the uncompressed in-core BV representation. Cache-efficient layout shows the best runtime performance during ray tracing the St. Matthew model.

3.5.2 Comparison

We compare our method with prior work on reducing sizes of BVHs.

Hierarchical quantization methods: QSplat [54] and Quantized kd-tree [55] employed fixed-rate quantization methods. We also use a fixed-rate quantization method and further compress them by using a simple prediction method and encoding prediction errors with a dictionary-based encoder. We choose to use this more aggressive compression method to further reduce the storage requirement and lower the expensive data access time, which is getting more expensive given the current computation trend [6]. We compare our compressed representation over the quantized BVH representation, whose bounding volume information is quantized to a fixed 16 bits. Our method achieves a 6:1 compression ratio over the quantized BVH representation. Moreover, our method still improves the runtime performance 2.3 times over using the quantized BVH for ray tracing the St. Matthew model.

Gzip compression: One can use the gzip compression method to compute compact BVHs. We compare our method with the gzip compression method. Since BVHs that are compressed by gzip do not provide random access directly, we perform gzip compression for each cluster. We use the zlib library [109]. Gzip achieves a 3.2:1 compression ratio over the original uncompressed BVHs and its decompression throughput is 17 MB per second. However, it performs 6.7% slower than using uncompressed data for the ray tracing, mainly because of its low decompression performance. Our method achieves a compression ratio about 3 times higher and a decompression performance six times faster. As a result, our method improves the runtime performance more than 3 times over using gzip for the ray tracing. This result is not surprising since gzip uses a combination of LZ77 and Huffman encoding [100], which are expensive compression methods which do not work well for the non-repeating floating values found in the original uncompressed BVHs. We measure the data access time including the I/O time and decompression time during ray tracing the St. Matthew model with different compression methods and original uncompressed data (see Fig. 3.7). As can be seen in the figure, our RACBVH representation shows much higher compression ratio and lower data access time.

Ray-Strip methods: Recently, Lauterbach et al. [56] proposed a *RayStrip*, an in-core compact hierarchy representation for ray tracing and further improved it in a following work [57]. Its main idea is to compute a triangle strip and build a balanced implicit spatial kd-tree on the triangle strip. This method is optimized for an in-core ray tracer and is not tested with other applications. Since this method always uses a complete spatial kd-tree, its runtime performance may be lower than ray tracers

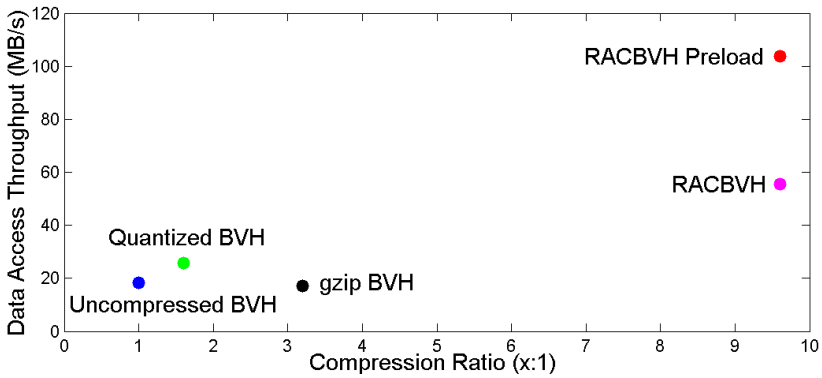


Figure 3.7: **Data Access Throughput vs. Compression Ratio:** This graph shows the data access throughput during ray tracing and the compression ratio of each compression method. The data access throughput is computed by summing the data accessing time and decompression time. Uncompressed BVHs do not require any decompression time. We set the data fetching time to be zero in the case of pre-loading the RACBVHs. Note that the higher compression ratio does not always cause the higher data access throughput.

that use optimized hierarchies (e.g., hierarchies optimized with the SAH). Also, our method achieves a 50% higher compression ratio and may perform better because it can use optimized BVHs. However, the Ray-Strip representation can be used as a compact in-core representation in our runtime framework for the improved runtime performance. The Ray-Strip representations could be used for lower regions of BVHs whose hierarchy quality has less impact on the overall ray tracing performance.

Statistical compression methods: Most prior compression methods that target higher compression ratio use statistical methods such as an arithmetic encoder [102]. We also initially tried an arithmetic encoder to achieve higher compression ratio. We found that, by using the arithmetic encoder, we can improve a compression ratio by a factor of two, but the decompression performs about three times slower than using our dictionary-based method. Therefore, the proposed dictionary-based method improves the runtime performance by a factor of two for the tested benchmark applications over the arithmetic encoder.

3.6 Conclusion and Future Work

We have presented a novel compression and runtime BVH decompression framework that transparently supports random access on the compressed BVHs. Our compression method preserves the original layout of a BVH and sequentially compresses BVs of a BVH. In order to support random access on the compressed BVHs, we decompose an input BVH into a set of clusters. Each cluster contains consecutive BV nodes and serves as an access point at runtime. We propose a general BVH access API to transparently support random access on our RACBVH representation. Our decompression framework selectively fetches, decompresses, and stores data in our in-core BVH representation. We have demonstrated the benefits of our methods on two applications having different characteristics. We achieved up to a 12:1 compression ratio and up to a 4:1 runtime performance improvement in the tested benchmarks.

There are many interesting avenues for future work. Our current method achieved two times performance improvement by supporting parallel random access on our compressed representations with four

CPU-cores for ray tracing. Although this 50% parallel efficiency is higher than that achieved by accessing the original data, we would like to achieve a higher parallel efficiency by designing more compact in-core representations and aggressive compression methods. Also, we would like to apply our method to highly parallel architectures such as GPUs and Larrabee architecture [110]. Also, some of interactive ray tracers employ levels-of-detail (LOD) hierarchies. These LOD-based ray tracers can have very high space requirements due to the LOD hierarchy. We would like to apply our method to reduce the memory requirements of LOD hierarchies and design an interactive LOD-based ray tracer. Finally, our current method preserves the mesh layouts, causing our method to store meshes separately from BVHs. It may be possible to achieve a higher compression ratio and runtime performance by coupling meshes and BVHs, without preserving the original triangle layouts. It may be interesting to design such a method, while maintaining coherent mesh layouts within BVHs.

Algorithm 1 Encode(N)

Require: A node index N

```
Index P    // Parent node index of N
if N ∉ Hash, the hash table then
    // N is a local root
    Encode(0)
    Assign and encode N's parent node to P
else
    // N is not a local root
    P ← Hash.Find(N)
    int k ← Front.GetOffset(P)
    Encode(k + 1)
end if

Update Front and Hash.

// Quantize and encode the BV of the N
QBV pQBV ← Quantize(GetBV(P))
QBV nQBV ← Quantize(GetBV(N))
QBV predQBV ← Predict(pQBV)
Encode( diff. between predQBV and nQBV)

if N is a leaf node then
    // Encode triangle indices
    Index triIdx ← GetTriIdx(N)
    Encode(triIdx − lastTriIdx)
    lastTriIdx ← triIdx
else if N is a local leaf then
    // Encode child indices
    Index leftIdx ← GetLeftChildIdx(N)
    Index rightIdx ← GetRightChildIdx(N)
    Encode(leftIdx − lastChildIdx)
    Encode(rightIdx − leftIdx)
    lastChildIdx ← rightIdx
end if
```

Chapter 4. Random-Accessible In-Core Compression

Although the data transmission time is greatly reduced by using compression techniques (Chapter 3), the number of disk I/O remains same because the cached in-core representation is identical, thus performance improvement is limited. In this chapter, an in-core compression technique is proposed to cache more information. Unlike out-of-core compression techniques, the decompression performance should be much faster since the decompression is performed for every single data query.

In order to achieve high compression ratio, the proposed method encodes mesh and BVH information together. Note that each representation is encoded separately in Chapter 3. Also, because of the coupled encoding, the out-of-core compression part is redesigned.

In this chapter, we propose a novel hierarchical-culling oriented compact mesh representation, *HCCMesh*, for massive models. The HCCMesh supports various tree structures of BVHs and efficiently provides random hierarchical traversal and culling. To compute the HCCMesh representation, we first construct a BVH from a mesh and then decompose the BVH into a single high-level BVH and multiple low-level BVHs. Then we compress each low-level BVH into our in-core HCCMesh representation, *i-HCCMesh* (Sec. 4.2), and then further compress it into our out-of-core representation, *o-HCCMesh* (Sec. 4.3). Given a general out-of-core data access framework (Sec. 4.1), we selectively fetch the o-HCCMesh of a low-level BVH requested during the hierarchical traversal and decompress it into the i-HCCMesh. Our HCCMesh representations offer the following benefits:

- **Low memory requirement:** Our i-HCCMesh and o-HCCMesh has 3.6:1 and 10.4:1 compression ratios on average over a naively quantized representation. This low memory requirement reduces the data access time and the size of the working set during the hierarchical traversal.
- **High performance improvement:** We test our method on ray tracing, photon mapping, non-photorealistic rendering, and collision detection (Fig. 4.1 and Sec. 4.4.1) and compare our method over the naively quantized representation. We can handle models ten times larger in these applications without the expensive disk I/O thrashing by using our representation (Fig. 4.2). In the case when we can avoid the disk I/O thrashing, we can improve the performance by up to two orders of magnitude.

4.1 Overview

In this section we give an overview of our approach to efficiently access meshes and BVHs of massive models.

Meshes and BVHs: Our method takes a triangular mesh and a BVH constructed from the mesh as two inputs. The mesh can have multiple attributes (e.g., color and normal) for each vertex and triangle. Our technique does not require a particular hierarchy construction method for BVHs. We, however, assume a full binary BVH and the AABB as a BV because of its simplicity and wide use in numerous applications [58, 7]. We further assume that each leaf node of a BVH contains a single triangle of a

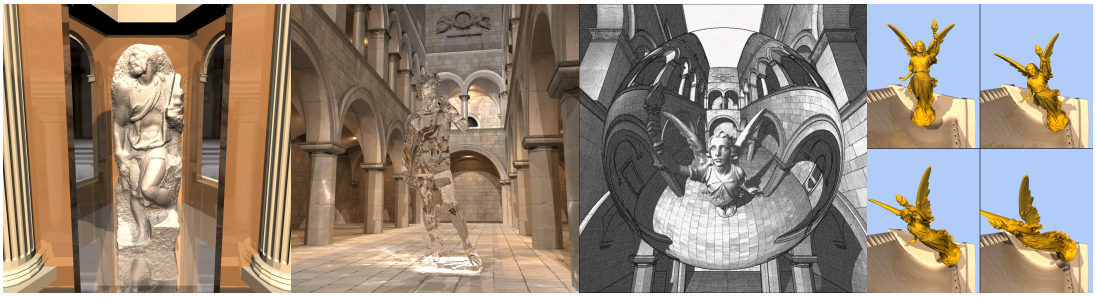


Figure 4.1: **Applications:** These figures show images of applications using our HCCMesh representations. From left, we show a Whitted-style ray tracing of the St. Matthew, photon mapping on a transparent David model in the Sponza scene, a line-art style rendering of the Lucy model reflected on a sphere, and collision detection between the Lucy and a CAD turbine model

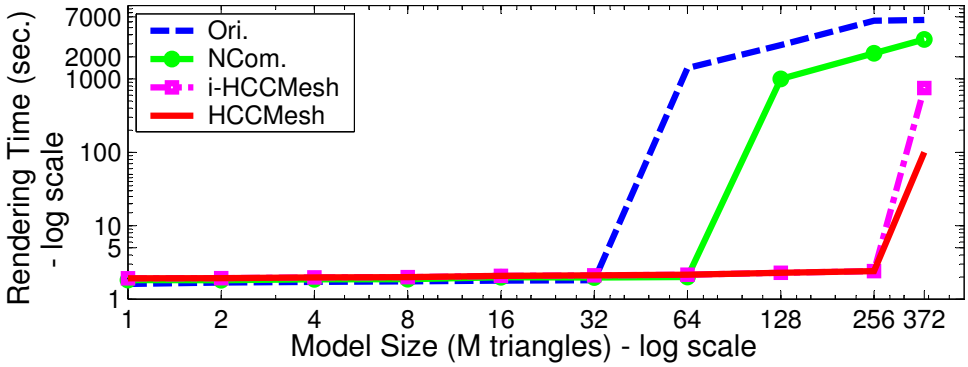


Figure 4.2: **Ray Tracing Time vs. Model Complexity:** This graph shows the rendering time with various model complexities of the St. Matthew model shown in Fig. 4.1. We measure the performance of ray tracing with our HCCMesh, the original (**Ori.**), and naively compressed (**NCom.**) representations.

mesh. It is straightforward to extend our method to rooted binary trees and k-ary trees. In Sec. 4.5 we extend our method to support other types of BVs and leaf nodes that contain multiple triangles of the mesh. Unless mentioned otherwise, the term of a BV refers to an AABB.

Random hierarchical traversal: To perform ray tracing, collision detection, etc., BVHs of meshes are traversed hierarchically. If a node is accessed during the hierarchical traversal, its two child nodes are stored in a queue or a stack and then are used for a breadth-first or depth-first traversal. Also, a node and its sub-tree can be culled during the traversal. Therefore, it is hard to predict the runtime access pattern on the hierarchies at the preprocessing time or to optimize the access pattern at runtime. We define such an access pattern as a *random hierarchical traversal* and optimize our representation to efficiently support this type of the access pattern.

4.1.1 Out-of-Core Runtime Access Framework

To handle meshes and BVHs of massive models that cannot fit into main memory, we employ an out-of-core runtime access framework [98]. The framework maintains a memory pool, whose size is determined by the available main memory.

To use this out-of-core access framework, we first decompose an input BVH into a single high-level sub-BVH and multiple low-level sub-BVHs (Fig. 4.3), using a simple clustering method. For simplicity,

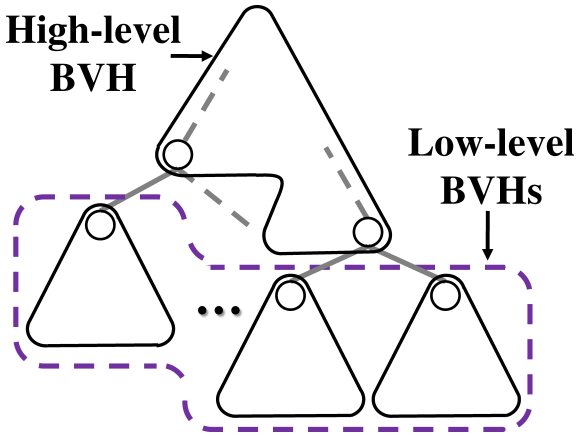


Figure 4.3: Decomposition of a BVH to high-level and low-level BVHs

we call these sub-BVHs high-level and low-level BVHs respectively. We construct a low-level BVH such that it has less than 512 vertices in the BVH, in order to design a compact in-core BV representation (Sec. 4.2.2). To compute such low-level BVHs, we traverse the original BVH in a bottom-up manner and count the number of vertices associated with each intermediate node. If an intermediate node has less than 512 vertices and its parent node has more, then we determine the node to be a root node of a low-level BVH. All the ancestor nodes above root nodes of these low-level BVHs are assigned to the high-level BVH.

As a BVH is traversed, an application may request a BV node or a mesh element (e.g., a vertex or a triangle). Our runtime access framework identifies the high-level or a low-level BVH containing the requested data. If the BVH has not been loaded yet, we load it, mark its availability in a page table, and return the data to the application. We also employ a simple memory management method based on the least-recently used (LRU) replacement policy. To implement the LRU replacement policy, we maintain a LRU list containing BVHs that have been accessed. This framework is easily extended to a parallel access mode and is a well-known concept used in many different out-of-core methods [98].

Even with this general out-of-core framework, we found that it still takes a huge amount of time to load and access data for massive models. This is mainly because external drives (e.g., disks) have low reading performance and because BVHs and meshes have high memory requirements. Also, once the size of working set is greater than the available main memory, expensive I/O thrashing occurs and drastically degrades the performance.

4.1.2 Our Approach

In order to reduce the data fetching time from external drives and to lower the memory requirement of meshes and BVHs, we compute our HCCMesh representation for each low-level BVH. Our HCCMesh representation has in-core and out-of-core parts. The in-core HCCMesh representation, i-HCCMesh, tightly integrates the mesh and BVH representations. We compress i-HCCMeshes further to reduce the expensive data access time from external drives for applications that run in an out-of-core mode. We do not compress the high-level BVH since it is frequently used and is relatively small (e.g., 4 MB for a model consisting of 100 M triangles). Fig. 4.4 shows the overall structure of the out-of-core access framework with our main contributions.

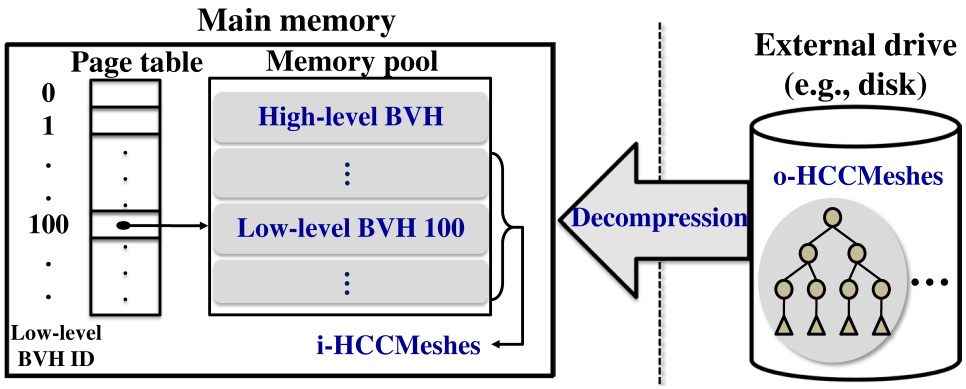


Figure 4.4: **Out-of-core Runtime Access Framework:** Given the framework, our main contributions (shown with blue colors) are compact in-core and out-of-core HCCMesh representations and compression methods that allow fast runtime decompression performance.

When a low-level BVH is requested at runtime, we fetch its corresponding o-HCCMesh from an external drive, decompress it, and store it in main memory as our i-HCCMesh representation which efficiently supports the random hierarchical traversal. In order to enable a high overall performance improvement, our methods support high compression ratios and fast decompression performance. If all the i-HCCMeshes fit into main memory, the o-HCCMeshes are sequentially fetched and decompressed into the i-HCCMeshes as the application begins. When all the o-HCCMeshes, but not all the i-HCCMeshes, fit into main memory, we load the o-HCCMeshes to main memory without any decompression and then decompress them into the i-HCCMeshes when necessary, in order to remove the expensive disk I/O access at runtime. Otherwise, the o-HCCMeshes are fetched on demand from the disk and decompressed into i-HCCMeshes while using the LRU-based memory management.

4.2 i-HCCMesh Representation

In this section we describe our in-core HCCMesh representation, i-HCCMesh, and discuss its results.

4.2.1 Overall Representation

A common AABB node records two types of information: 1) the minimum and maximum bounds of an AABB and 2) information about its tree structure. Each intermediate node has two left and right indices to encode the left and right child nodes. Each leaf node stores a triangle index in the same position used for recording the left or right indices in the intermediate nodes. This simple AABB representation uses 32 bytes and efficiently supports the random hierarchical traversal. However, using this simple AABB representation on massive models may require huge amounts of memory space (e.g., more than 6 GB for a mesh consisting of 100 M triangles).

An i-HCCMesh representation of a low-level BVH (Fig. 4.5) consists of a header, a BV node array, an inter-connection array, and a vertex array. The header contains the numbers of BV nodes and vertices associated with the BVH. The BV node array contains the BV information and the inter-connection array represents the tree structures of the BVH. Also, the vertex array contains the mesh information. Each BV node in the i-HCCMesh is compressed to take only 4 bytes.

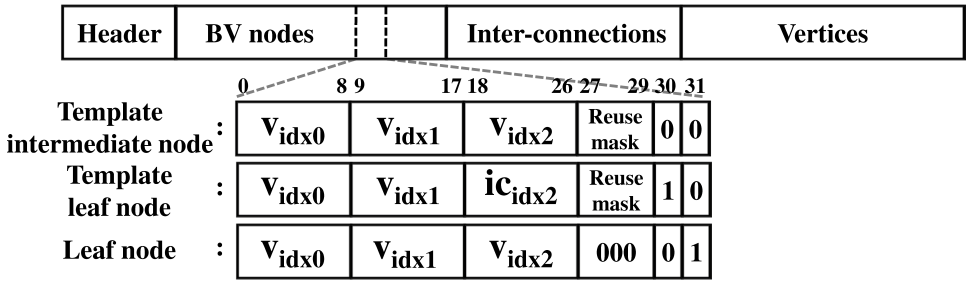


Figure 4.5: **i-HCCMesh of a Low-Level BVH:** The i-HCCMesh consists of a header, a BV array, an inter-connection array, and a vertex array. There are three different BV node types in the i-HCCMesh and two bits are used to encode the type of a node.

4.2.2 Encoding Bounding Volumes

The BV of a node is constructed to tightly enclose all the triangles contained in the sub-tree of the node [7, 58]. Therefore, at least one vertex of a mesh is on a boundary of a BV and can define the boundary (see the parent BV in Fig. 4.6-(a)). This observation [57] allows us to efficiently encode the BV information of AABBs.

Since an AABB has 6 extents (minimum x, y, z and maximum x, y, z), we identify 6 vertices that define the 6 extents. Since these vertices are often shared among multiple BVs, we store their coordinates in the vertex array (Fig. 4.5) and use indices to the array to define extents of a BV. This representation requires $6 \times C$ bits to define an AABB, where C is the number of bits required to encode an index to the vertex array. If a low-level BVH contains many vertices, its working set size will increase and the cache coherence will lower during the hierarchical traversal. Therefore, in our current implementation, we set C to 9, which limits the size of the vertex array to 512 vertices. Later we will explain another reason why we choose C to be 9 particularly.

We further reduce the memory requirement of the BV by observing that some of 6 vertex indices defining an AABB are not required during the hierarchical traversal of a BVH. As we traverse the hierarchy, the BV information of a parent node of the currently accessed node can be easily available by caching and fetching the BV of the parent node in a stack or queue used for the hierarchical traversal. Since BVs are constructed to enclose triangles tightly, some of extents of a BV and its parent BV may be identical. In the case of the AABB, the following property is satisfied.

Inheritance property: For each coordinate axis (e.g., x, y, and z), one of the minimum extents of two child AABB nodes is inherited from (and thus same to) the minimum extent of their parent AABB node. The same property holds for maximum extents. Its proof is trivial.

Because of the inheritance property, at least 6 of the 12 extents of two child nodes are same as the extents of their parent BV. Therefore, instead of encoding all the 12 extents using vertices, we can reuse 6 of the 12 extents from a parent AABB.

In order to compactly encode the inherited extents, we use a 6-bit *reuse mask*. Each bit of the reuse mask corresponds to either minimum or maximum extent in x, y, and z coordinate dimensions. If a left AABB node inherits an extent from its parent AABB, we set the extent's corresponding bit to 1. Otherwise, we compute a vertex index defining the extent. We perform this process for each of 6 different extents. Instead of maintaining another reuse mask for the right node, we use the same reuse mask for the right AABB node. For the right node, we inherit an extent from the parent node, when the bit is set to be 0. It is possible that both child nodes may inherit the same extent from their parent

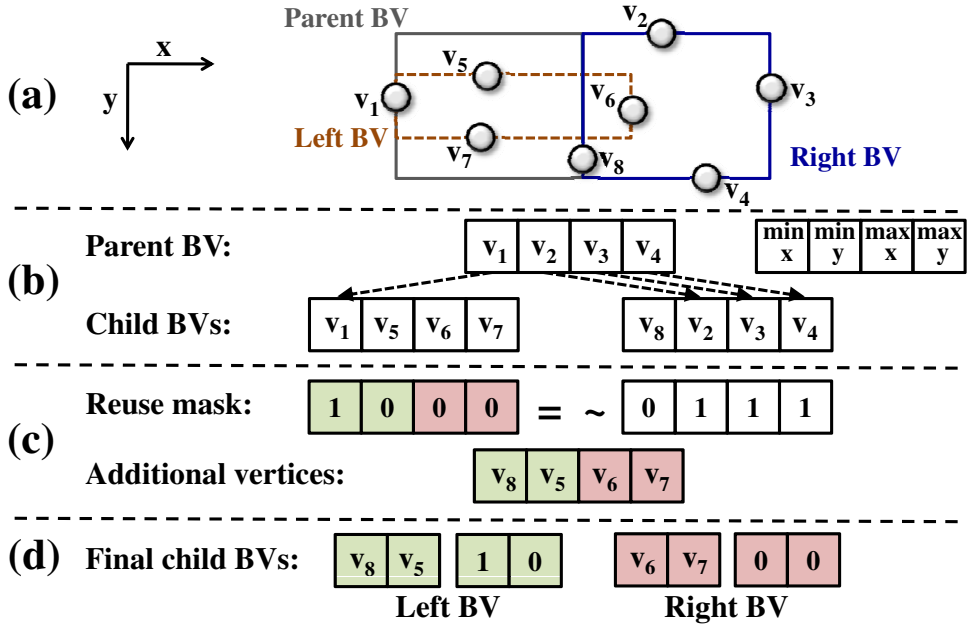


Figure 4.6: **2D Example of In-core BV Encoding:** (a) The figure shows AABBs of a node, its two child AABBs, and vertices defining extents of the BVs. (b) Vertices defining BVs are shown. (c) Reuse mask and vertices that are additionally specified are shown. (d) The final BV encoding for two child nodes.

node, but only one of the two child nodes inherits the extent in our scheme. A 2D example of our in-core BV encoding is shown in Fig. 4.6.

In our scheme, two child AABB nodes are represented by a 6-bit reuse mask and by 6 vertex indices. We divide them in half and distribute three vertex indices and three bits of the reuse mask to each child node. We store all the BVs of a low-level BVH in the BV array associated with the BVH (see Fig. 4.5). This representation requires $3 \times C + 3$ bits per BV. Since we use 9 bits for C in our current implementation, the in-core BV information uses 30 bits, which are stored in 4 bytes; the unused two bits are used to encode three different types of a node, which will be explained later in Sec. 4.2.3.

When a BV of a node is requested at runtime, we identify extents inherited from its parent BV, using 6-bit wide reuse mask and six vertex indices stored in the node and its sibling node.

4.2.3 Encoding Tree Structures

In order to compactly represent tree structures, we propose a novel structure, *tree templates*. k-height tree templates are defined as all the possible tree structures of trees having a k tree height; the height of a tree containing only one node is 1. For example, suppose that we have a tree with a height of 3. There are only 3 different tree structures and thus there are three 3-height tree templates (see Fig. 4.7-(a)).

To represent the structure of a tree with any height using tree templates, we horizontally partition the tree into sub-trees consisting of a height of k in a top-down manner. This simple partitioning method computes sub-trees consisting of a height of 1 to k. Then we encode the tree structure of each sub-tree by encoding an index to all the possible 1- to k-height tree templates (see Fig. 4.7-(b)). The number of k-height tree templates, $T(k)$, is computed as the following (see the supplementary report for the proof

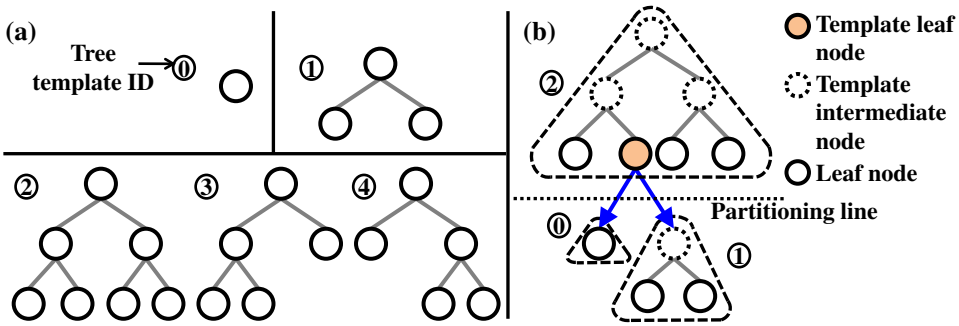


Figure 4.7: **Tree Templates:** The left figure shows possible tree templates of a height of 1, 2, and 3, with its ID. The right figure shows a tree partitioning example with sub-trees that have up to a height of 3 and the computed tree template IDs.

that is available at our project URL):

$$T(k) = \begin{cases} 2 * T(k-1) * \left(\sum_{i=1}^{k-2} T(i)\right) + T(k-1)^2 & \text{for } k \geq 3 \\ 1 & \text{for } 1 \leq k \leq 2 \end{cases}$$

This function grows extremely fast, $O(2^{2^k})$, but it is reasonably small when k is small (e.g. $T(3) = 3$ and $T(4) = 21$). In our current implementation, we use 1- to 4-height tree templates since there are only 26 different types.

Let us call an intermediate node of the original BVH, but a leaf node within a sub-tree after the partitioning to be a *template leaf node*. We also call all the rest of intermediate nodes *template intermediate nodes* (see Fig. 4.7-(b)). We still call leaf nodes of the original BVH leaf nodes. These are three types of a BV node in our representation.

Although we represent each partitioned sub-tree with a tree template, the links between tree templates (e.g, blue arrows in Fig. 4.7-(b)) should be encoded additionally. We call such links *inter-connections*. However, there is not enough space to encode this information in the 4 byte BV structure. To encode these inter-connections, i.e., child nodes of template leaf nodes, within the 4 byte BV structure, we use the *inter-connection array* for each low-level BVH (Fig. 4.5). Then we design each template leaf node to record an index, ic_{idx2}, pointing to the inter-connection array. We store the index, ic_{idx2}, in the position of a vertex index in the template leaf node (see the structure of template leaf nodes in Fig. 4.5) and then store the vertex index at an entry referred by the index ic_{idx2} in the inter-connection array. In addition to recording a vertex index, each entry of the inter-connection array records 1) two IDs of tree templates of the left and right sub-trees and 2) two indices that point to the root BVs of these two sub-trees in the BV array. Once we have accessed an entry in the inter-connection array from a template leaf node, we can traverse to the left or the right child node of the node.

4.2.4 Encoding Meshes

We quantize vertex geometry and normals of the mesh using simple quantization methods [54] and colors are also encoded by using a color palette. We store three vertex indices of the triangle stored in each leaf node in the same positions where three vertex indices are recorded for an intermediate node. Therefore, we can use a 4 byte BV structure for leaf nodes too.

Note that each low-level BVH has its vertex array containing vertices used by the BVH. This representation can improve the cache coherence during the BVH traversal, since every required vertex

is located in the array. However, it can lower the compression ratio, since vertices shared by multiple triangles can be stored multiple times in different low-level BVHs. We found that 14% of vertices of the original mesh are duplicated in our tested models and these duplicated vertices take 5% of the size of i-HCCMesh.

4.3 o-HCCMesh Representation

We further compress i-HCCMesh for the out-of-core case, which requires the expensive data access time to read data from an external drive. Since the i-HCCMesh is already compressed, it is hard to compress the i-HCCMesh further using general compression methods (e.g., gzip). Therefore, we propose a dictionary-based compression method that allows a fast decompression performance while reducing the storage requirement further.

4.3.1 Compression Method

We compress an i-HCCMesh of a BVH by traversing each node of the BVH in a depth-first order. When we encounter a leaf node during the traversal, we encode its contained mesh information: 3 vertex indices, vertex coordinates, and their attributes. To compress the mesh information, we use the streaming mesh compression method [111], since it runs quite fast while achieving a high compression ratio. We modify the streaming mesh compression method that employed a statistical compression method [100] to use our simple dictionary-based compression method, which will be explained later in Sec. 4.3.2. We make this modification since the statistical compression lowers the runtime performance of applications compared to using our method; the statistical compression achieves only 18% more compression, but has 2 times slower decompression performance. We also use our dictionary-based compressor to encode tree template IDs and inter-connections.

We quantize each vertex coordinate to 16 bits and store it in the o-HCCMesh. We further compress it using a parallelogram prediction method [32] and then encode the prediction error using our dictionary-based compressor. To apply the prediction method, we construct the mesh information during compression and decompression as used in the streaming mesh compression method [111]. We also treat other attributes in a similar manner.

We do not encode any of the BV information in the o-HCCMesh, because they can be reconstructed from the encoded tree structures and the mesh information contained in leaf nodes. Once we decoded the tree structure and the mesh information of the BVH symmetrically to the compression method, then we traverse the decompressed tree in a bottom-up manner and re-construct the AABB information of a node based on the vertex information contained in the sub-tree rooted at the node. This BV reconstruction process is done efficiently in linear time, proportional to the number of triangles contained in the BVH.

4.3.2 Dictionary-based Compression

We employ one more layer of compression using a simple dictionary-based compressor. This compressor maintains a dictionary consisting of s different entries. We create a separate dictionary for each compression context (e.g., tree templates, vertex indices, reuse masks, etc.). In order to choose symbols to be included in the dictionary, we compute a probability table for symbols and then choose the s symbols that appear most frequently. If a symbol currently being encoded is in the table, then the index of the table entry is encoded. Otherwise, we simply encode the symbol itself. We use a fixed number

of bits to encode an index of a dictionary entry, instead of a Huffman encoding [100], since we prefer to have faster decompression performance.

We propose a simple, but automatic method for determining the size, s, of a dictionary, in order to achieve an optimal compression ratio for each compression context. Let $f(i)$ be the probability function of the ith symbol appearing in the input symbols, sorted in the order of a decreasing probability. Also, let p be the probability that an input symbol being encoded is in the dictionary. Then, $p = \sum_{i=1}^{s} f(i)$ and the number of bits, $B(s)$, required to represent the compressed data is $p \log_2(s) + (1-p)C_o$, where C_o is the number of bits required to encode the original symbols.

The size, s, of the dictionary table achieving an optimal compression ratio can be found by minimizing the required number of bits, $B(s)$. Note that as we increase s, the term of $p \log_2(s)$ monotonically increases from zero, while the term of $(1-p)C_o$ monotonically decreases from C_o. Therefore, as we increase s, $B(s)$ will decrease initially and then increase at one point, s_m. The optimal s is simply computed by choosing either $s_m - 1$ or s_m that gives us a lower $B(\cdot)$.

4.4 Results and Applications

We have implemented and tested our method with a variety of applications that require random hierarchical traversal and culling. We use an 2.83 GHz quad-core machine with 4 GB main memory and 1 K by 1 K image resolution for all the rendering results, unless mentioned otherwise. Refer to the accompanying video for visual results of our applications.

We have computed our i-HCCMeshes and o-HCCMeshes with various benchmark models (Table 4.1). We compare HCCMeshes with the original uncompressed mesh and BVH, which uses the simple AABB node (32 bytes), vertex coordinates (12 bytes), vertex normals (12 bytes), vertex colors (12 bytes), and triangles (32 bytes) [62]. We also compare HCCMeshes with a naively compressed BVH and mesh, whose vertex, normal, BVs, and colors are quantized in the same manner applied to the HCCMesh representation. We call the original and naively compressed representations **Ori.** and **NCom.** respectively. We use cache-oblivious layouts [7] for meshes and BVHs of **Ori.** and **NCom.** since they are known to reduce the number of cache misses.

Compression ratio and decompression performance: On the tested benchmark models, the i-HCCMeshes achieve 7.2:1 and 3.6:1 compression ratios on average over **Ori.** and **NCom.** respectively. Also, the o-HCCMeshes achieve 20.9:1 and 10.4:1 compression ratios on average over **Ori.** and **NCom.** respectively (Table 4.1). Our compression method can process 102 K triangles per second to compute the i-HCCMeshes and o-HCCMeshes together. Our decompression method decoding from the o-HCCMeshes to the i-HCCMeshes can process 2.3 M triangles per second, when we use a single core and exclude the time spent on reading data from an external drive.

4.4.1 Applications

To demonstrate the benefits of using our HCCMesh representation, we test our method on a variety of applications that require random hierarchical traversal and culling on the meshes. At a high level, our tested applications can be classified as ray tracing applications (including Whitted-style ray tracing, non-photorealistic rendering, multi-resolution ray tracing, and photon mapping) and collision detection application. Also, we discuss how we can apply our methods to compress other types of hierarchies such as kd-trees and multi-resolution hierarchies. We found that the performance of our method follows a

Model	T (M)	Type	OS	NS	i-HCCMesh						o-HCCMesh				
					BV	Tr	M	CS	CR	CR'	Tr	M	CS	CR	CR'
St. Matthew	372	S	40459	20226	2836	600	1997	5474	7.4	3.7	478	1212	1738	23.3	11.6
Sim. St. Ma.	128	S	13925	6961	976	212	700	1902	7.3	3.7	167	496	678	20.5	10.3
Lucy	28	S	3008	1502	210	46	154	413	7.3	3.6	36	113	153	19.7	9.8
David	8	S	897	449	63	13	46	122	7.4	3.7	11	33	45	19.9	10.0
Power plant	13	C	1547	738	97	24	125	248	6.2	3.0	18	71	92	16.8	8.0
DE tanker	82	C	10142	4794	623	162	880	1677	6.0	2.9	116	477	614	16.5	7.8
Iso-surface	102	I	11148	5572	781	171	594	1554	7.2	3.6	135	510	657	17.0	8.5
Turbine	2	S	192	96	13	2.9	9.9	26.4	7.3	3.6	2.3	8.6	11	17.4	8.7
Sponza	.06	A	12.9	5.2	0.5	0.1	1.9	2.6	5.0	2.0	0.1	0.4	0.9	14.3	5.8

Table 4.1: **Benchmark Models: T(M)** is the number of million triangles. **S, C, I**, and **A** in the **Type** column represent scanned, CAD, iso-surface, and architecture types of models. **OS**, **NS**, and **CS** are the sizes of original, naively compressed, and HCCMesh representations. **BV** and **Tr** are the BV information and tree structures in BVHs respectively. **M** is the mesh information stored in each HCCMesh representation. These are shown in a megabyte unit. **CR** and **CR'** are compression ratios over the **OS** and **NS** respectively. Sim. St. Ma. represents a simplified St. Matthew model consisting of 128 M triangles.

similar pattern in many of these applications. Therefore, we report results of extensive tests only with the Whitted-style ray tracing to explain main runtime characteristics of our method.

Whitted-Style Ray Tracing

We implement a Whitted-style BVH-based ray tracer [105] that can use HCCMeshes, **Ori.**, and **NCom**. Moreover, in order to compare the performance of the ray tracer on various model complexities (Fig. 4.2), we compute several simplified versions (e.g., from 1 M to 256 M versions) of the St. Matthew model consisting of 372 M triangles. We use 7 point light sources with shadow and reflections to generate the rendering result shown in the leftmost image of Fig. 4.1.

When **Ori.** and **NCom.** fit into the available main memory, the ray tracer using only the i-HCCMeshes shows 33% and 6% lower performance than **Ori.** and **NCom.** respectively, mainly because of the overhead of decompressing the i-HCCMeshes (Fig. 4.2). However, from the 64 M version of the St. Matthew model for **Ori.** and from the 128 M version of the same model for **NCom.**, **Ori.** and **NCom.** do not fit into main memory. On the other hand, the i-HCCMeshes of the 64 M and 128 M versions fit into main memory and we improve performances by more than two orders of magnitude over **Ori.** and **NCom.** by using the i-HCCMeshes. Such a huge performance improvement is achieved by reducing the memory requirement and by avoiding the expensive disk I/O thrashing.

Note that the performance using the i-HCCMeshes goes down when we test the original 372 M St. Matthew model, since the i-HCCMeshes of the model do not fit into the 4 GB main memory and require the disk I/O access at runtime. However, by using the o-HCCMeshes and i-HCCMeshes together, we can reduce the disk I/O access time and achieve 63 times and 34 times improvements over using the **Ori.** and **NCom.** respectively. The HCCMeshes of our biggest tested model can fit into the 4 GB main memory. We expect that models consisting of more than 800 M triangles would not fit in the 4 GB main memory. However, even in this case, we expect that our method would improve performances close to

the o-HCCMesh's storage reduction (e.g., up to 20 times and 10 times) for the tested applications over **Ori.** and **NCom.**.

Non-Photorealistic Rendering (NPR)

NPR is attracting more attention, since it can effectively convey salient features of models to viewers. Recently, a GPU-based real-time technique has been developed to render reflections and refractions in line-art styles [112]. However, this technique has not been tested with massive models, which cannot be efficiently handled by GPU ray tracing. We implement this technique with our CPU-based Whitted-style ray tracer (Sec. 4.4.1) using our HCCMesh representations. We use 25 point lights in the Sponza scene with the Lucy model (Fig. 4.1). For all the tests in the rest of the paper, we use a single-core CPU with 2 GB memory memory. Using our representation takes 15 seconds for the NPR of the scene and improves the performance by a factor of about two orders of magnitude over using **Ori.** because of removing the disk I/O thrashing.

Multi-Resolution Ray Tracing

Multi-resolution techniques are widely used to improve the performance of many rendering algorithms [113]. One downside of most multi-resolution representations is that they usually require more storage space than single-resolution representations. Our HCCMesh representations can be applied to reduce the storage and memory requirements of multi-resolution representations.

We apply our method to a BVH augmented with LOD representations for multi-resolution ray tracing [78]. A LOD representation of this method consists of a normal of a LOD plane and an associated LOD error. We quantize floating point data. Then as we traverse the tree in a depth-first order, we compress the quantized data further by using a simple prediction method and by encoding the prediction error using our dictionary-based compressor. For example, when we have to compress a normal of a node, we predict its normal based on the normal associated with a node we encountered earlier during the hierarchy traversal. Note that we cannot store this LOD information into our 4 byte in-core BV structure of the i-HCCMesh. Instead, we compute LOD representations only for template leaf nodes. The original multi-resolution ray tracing method [78] also computes their LODs for nodes, whose depths are multiples of three or four in the hierarchy. For these template leaf nodes we encode their additional LOD representations in the inter-connection array of each low-level BVH.

The original multi-resolution representation takes 8.7 GB [78] for the 128 M version of the St. Matthew model. The i-HCCMesh and o-HCCMesh representations reduce its storage requirement to 859 MB and 393 MB respectively. Note that typical working set sizes of the multi-resolution rendering methods are chosen to be smaller than the available main memory. Therefore, we do not expect our HCCMeshes to improve the runtime performance of multi-resolution rendering methods. However, our method shows a comparable performance (e.g., 32% lower performance) to the multi-resolution ray tracer using the original multi-resolution representation [78].

Photon Mapping

Photon mapping is a widely used for generating photorealistic visualizations. The rendering quality of photon mapping depends on the number of photons generated. For complex illuminations and scenes, we may have to generate a huge number of photons. Also, photon mapping uses a kd-tree or BVH of triangles of the model. Therefore, the memory requirement of photon mapping can be very high.

Although our HCCMesh representations are designed mainly for meshes and their associated BVHs, they can also be applied to photons, i.e. point clouds and photon kd-trees. In an in-core representation, we compress the tree structures of kd-trees using our tree templates. Each kd-node of the photon kd-tree contains information about a photon's incoming direction, intensity, etc. However, we do not compress these data further in the in-core representation, since they are already compactly represented in the photon kd-trees [114]. In an out-of-core representation we compress data using a prediction and error encoding technique similar to the one used to encode the LOD representation.

To render the David model in the Sponza scene with photon mapping (see Fig. 4.1), we generate 66 million photons from 34 light sources. The original photon kd-tree requires 2 GB of memory. Our in-core and out-of-core representations take 1.6 GB and 0.4 GB respectively; the compression ratio for the in-core representation is small, since we only compress the tree structure. We achieve a comparable rendering performance by using the HCCMeshes to that achieved by using the original representation.

Collision Detection

Collision detection is an essential technique for enabling user interaction. In practice, BVHs are widely used in practice [115]. We implement a rigid body simulation and drop the Lucy model on the top of a CAD turbine model (Fig.4.1). The uncompressed original, **Ori.**, and the naively compressed, **NCom.**, representations use 3.2 GB and 1.6 GB respectively, while the HCCMeshes reduce the memory requirement to 164 MB. Both our HCCMeshes and **NCom.** fit into the 2 GB main memory. By using the HCCMeshes, collision detection takes 184 ms for each simulation time step and we improve the performance by 2.1 times and 25 times over **NCom.** and **Ori.** respectively. Since the data access pattern of collision detection is more localized than ray tracing [97], we achieve lower performance improvements (e.g., 25 times) than those (e.g., about three orders of magnitude) achieved with ray tracing, even when we remove the disk I/O thrashing.

4.5 Discussions and Comparisons

In this section we discuss extensions to our method and compare our method to other prior methods.

Other types of BVs: Our method can be applied to oriented bounding boxes [116] and spherical BVs. Typically, the spherical BVs are widely used and a sphere is specified by its center and radius. We can represent spheres tightly enclosing triangles of a mesh with vertices of the mesh, since a sphere can be uniquely represented by four vertices. However, it is not easy to take advantage of inheritance property in the case of spherical BVs since vertices may not be shared by parent and child BVs. Our method can be applied to encoding kd-trees, as demonstrated with photon kd-trees in Sec. 4.4.1.

Leaf nodes containing multiple triangles: We can easily extend our current HCCMesh representation to encode multiple triangles in each leaf node. We currently specify three vertex indices of a triangle in each leaf node. Instead, we can use a triangle array containing vertex indices of all the triangles contained in a low-level BVH. Then we can simply encode the starting and end positions of vertex indices of triangles contained in a leaf node. We can construct a BVH for multiple triangles stored in each leaf node on the fly at runtime. We found that our representations with a single triangle in each leaf node perform better than those with multiple triangles (e.g., 1, 4, 16, and 128) to each leaf node for ray tracing the St. Matthew model. This is mainly because computing a BVH requires $O(nlogn)$ time

complexity, compared with the $O(n)$ time complexity of reading BVs from our representation, where n is the number of triangles contained in the BVH. Another alternative to building BVHs on the fly is to perform intersection tests without building BVHs. However, we found that this alternative method shows worse results than building the BVHs on the fly in the ray tracing application. Moreover, this alternative method can be very problematic for collision detection, since it causes quadratic time complexity.

We also compute a BVH of the 372 M St.Matthew model by assigning 16 triangles to each leaf node and then naively compress the BVH by quantizing BVs, vertices, etc. This naively compressed BVH takes 7.1 GB and it takes 21 minutes for ray tracing the model in the scene setting as shown in the leftmost image of Fig. 4.1. We also test a BVH constructed by assigning 128 triangles to each leaf nodes, but found that this performs worse than assigning 16 triangles to each leaf node. The HCCMesh that contains a single triangle in each leaf node takes 1.7 GB and takes 100 seconds for ray tracing the model. Our method requires much less memory requirement and performs better than the naively compressed BVH that contains multiple triangles in each leaf node.

Comparisons: Our method shows even higher compression ratios (e.g., about 3 times and 8 times higher than the ReduceM [57] and the LBVH [51] respectively) and, more importantly, supports various tree structures that are constructed from different optimized hierarchy construction methods [7, 58]. Since an optimized hierarchy can show 2 or more performance improvement than a naively constructed hierarchy [62], our method can perform better and handle bigger data sets. The RACBVHs (Chapter 3) support various tree structures. However, it does not use any compact in-core representation nor tightly integrate meshes and BVHs; it simply uses a separate compact mesh representation, RACMs [3]. Therefore, our i-HCCMesh and o-HCCMesh representation achieve 7.2:1 and 1.6:1 higher compression ratios over the in-core and out-of-core representation of the RACBVH/RACM respectively. We compare the performance of ray tracing the original St. Matthew model with our representation and the RACBVH/RACM. Our method improves the performance by 20 times over the RACBVH/RACM. Furthermore, our method has been tested with a much broader set of applications which have different characteristics, compared to all the work mentioned above.

We also compare our method using tree templates with succinct trees [61]. Encoding tree structures using tree templates shows a 30% lower compression ratio, but improves the performance of the tree traversal by 4.4 times over using succinct trees for ray tracing the St. Matthew model. This performance improvement is due to the more efficient random access performance of our tree template representation. We also compare the performance of ray tracing using the o-HCCMeshes of the 128 M version of the St. Matthew model to that using gzipped i-HCCMeshes compressed by running the gzip to i-HCCMeshes. We compress each low level BVH and corresponding mesh independently to support random access. The o-HCCMeshes are compressed three times more than the gzipped i-HCCMeshes. Moreover, ray tracing using the o-HCCMeshes runs 17 times faster than ray tracing using the gzipped i-HCCMeshes.

Compared with prior mesh compression methods mentioned in Chapter 2, the storage overhead of our representations may be high. This is mainly because our HCCMesh representations are designed to support efficient random hierarchical traversal and culling on the encoded mesh rather than achieving the highest compression ratio.

Limitations: Our method can be easily applied to rooted binary trees and k-ary trees. However, their compression ratios may be lower than those of computed with full binary trees, since there are many more tree templates with rooted binary trees and k-ary trees. Also, our i-HCCMesh and o-HCCMesh

representations have runtime decompression overheads. For small models that can fit into main memory, the overhead of our method may lower the runtime performance (e.g., by 33% for the Whitted-style ray tracing) compared to using the uncompressed data, as discussed in Sec. 4.4.

4.6 Conclusion and Future Work

We have presented a HCCMesh representation, which tightly integrates a mesh and a BVH. We believe that our HCCMesh representation is the first method that has been tested on various applications including rendering and collision detection that require the random hierarchical traversal. The i-HCCMesh and o-HCCMesh achieved 3.6:1 and 10.4:1 compression ratios on average over a naively compressed representation respectively. We can reduce the memory requirement of handling massive models and thus can handle models ten times larger without the expensive disk I/O thrashing. Moreover, by avoiding the disk I/O thrashing, we observed performance improvements by up to two orders of magnitude, compared to the original and naively compressed representations. Also, even if our HCCMeshes cannot fit into main memory, we expect that our method would improve performances by a factor close to its compression ratios to the original and other compressed representations.

In addition to addressing the current limitations of our method, we would like to extend our current method to highly parallel architectures such as GPUs and Larrabee [110]. Also, we would like to further improve the decompression performance of our method by exploiting data-level parallelism of GPUs and Larrabee architecture.

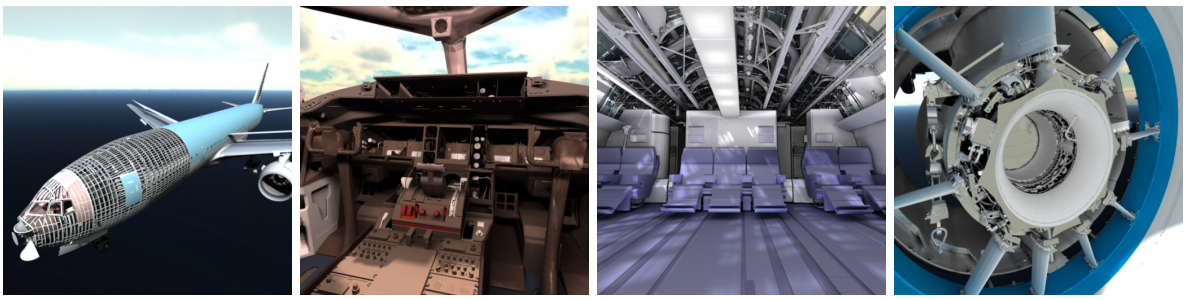
Chapter 5. Rendering Framework on Heterogeneous Computing Resources

In this chapter 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 consists of separate, three different levels-of-details (LoD) for the input model: the original polygonal representation, and coarse and fine volumetric representations (Sec. 5.2). 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 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



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

Figure 5.1: 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.

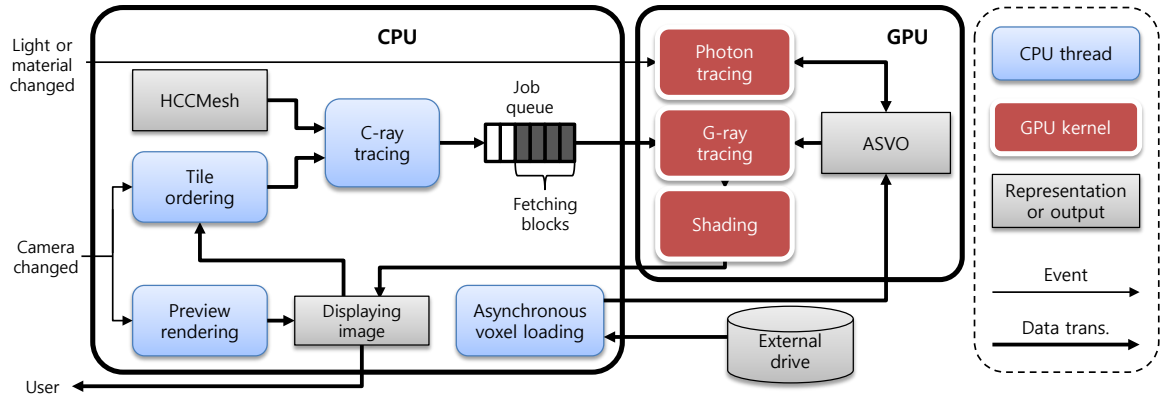


Figure 5.2: This figure shows our rendering framework and data transitions between different modules.

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 chapter 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. 5.1) 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.

5.1 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. 5.2).

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. 5.2 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.

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

5.2 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. 5.3). We pre-load all the data

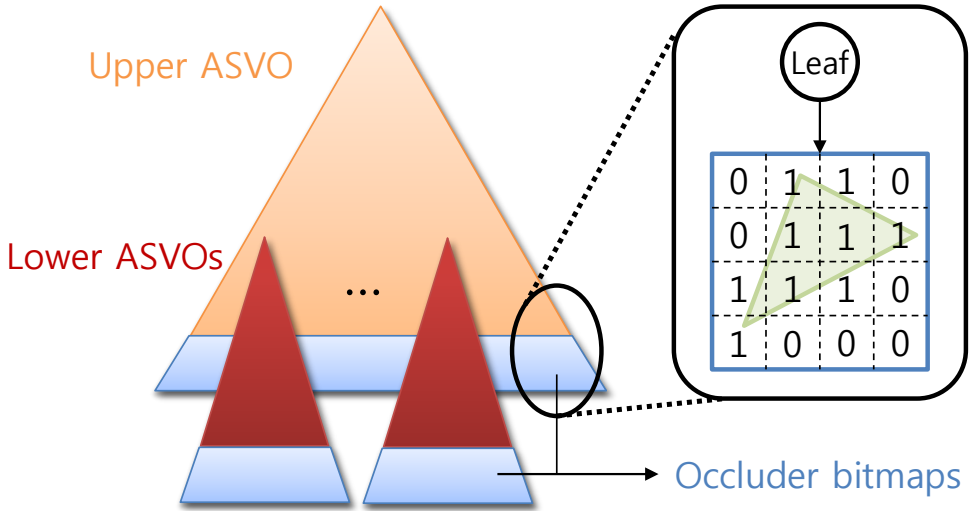


Figure 5.3: **Augmented Sparse Voxel Octree (ASVO)**: Our ASVO consists of upper and lower ASVOs, each of which is combined with occluder bitmaps for every leaf node. The occluder bitmap has bit values, each of which indicates whether its corresponding sub-voxel overlaps with the original geometry.

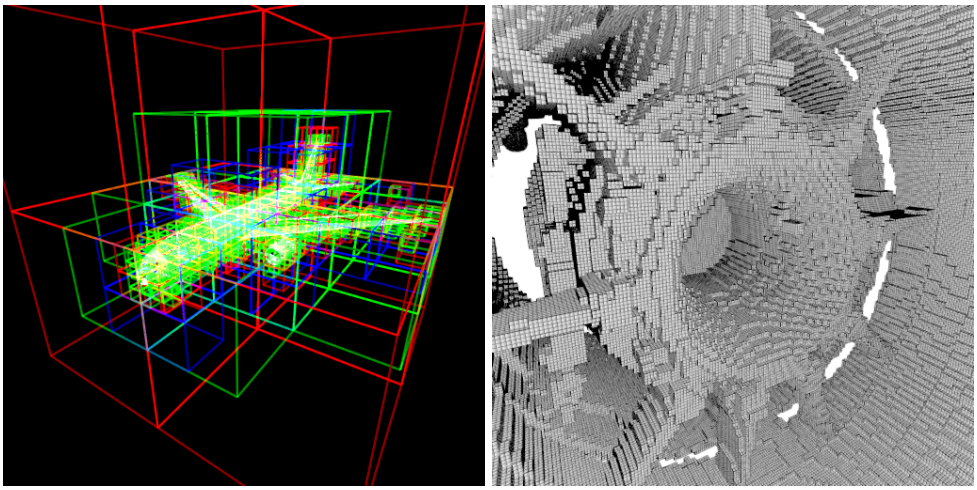
of the upper sparse voxel octree and its corresponding occluder bitmaps to the video memory of GPU and thus avoid their data transmission overhead between CPU and GPU at runtime. On the other hand, necessary portions of lower ASVOs (and their occluer bitmaps) are identified at runtime and asynchronously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

5.2.1 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. 5.4(a). The constraint on three dimensional sizes of the ASVO would generate many empty octree nodes. We store octrees as sparse octrees [31] 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



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

Figure 5.4: The left figure visualizes bounding cubes of voxels that have a depth of six, while the right figure visualizes leaf nodes of the ASVO.

runtime in a similar manner to [31].

5.2.2 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. 5.3. 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 its corresponding lower ASVO asynchronously [117] 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

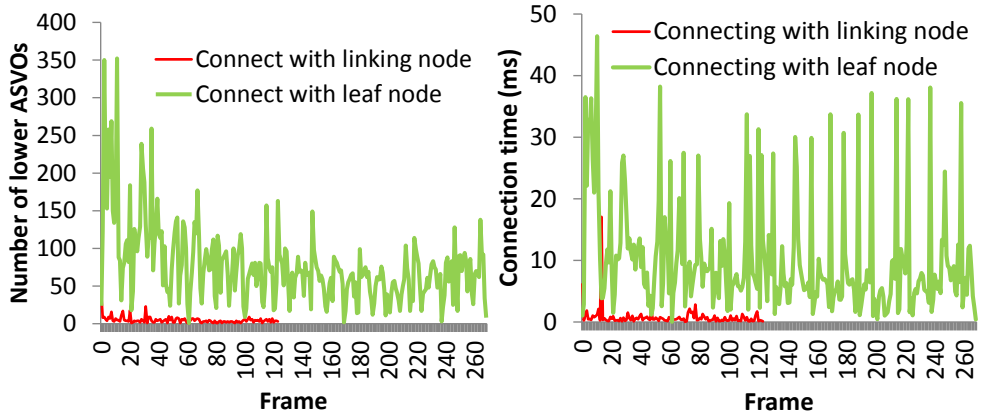


Figure 5.5: The left figure shows a number of loaded lower ASVOs per each frame, and the right figure shows synchronization time used for connecting lower ASVOs to the upper ASVO.

of loaded lower ASVOs and synchronization time in each frame between two methods of connecting lower ASVOs to linking and leaf nodes (Fig. 5.5). 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. 5.6). 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.

5.3 Rendering Algorithm

When we receive events of light or material changes we trigger the photon tracing module (Fig. 5.2). For photon tracing, we use the common photon tracing method of the standard photon mapping approach [20]; 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 per-

Figure 5.6: 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.

form photon tracing in the GPU side and accumulate photons on leaf voxels [31] 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. 5.8).

Finally, shading is done with photon mapping, and we perform the well-known G-Buffer based filtering [118] to reduce a variance level of indirect illuminations.

5.3.1 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. 5.6). 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 [119]. 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. 5.1(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.

5.3.2 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 [120]. 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. [121] 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. 5.7).

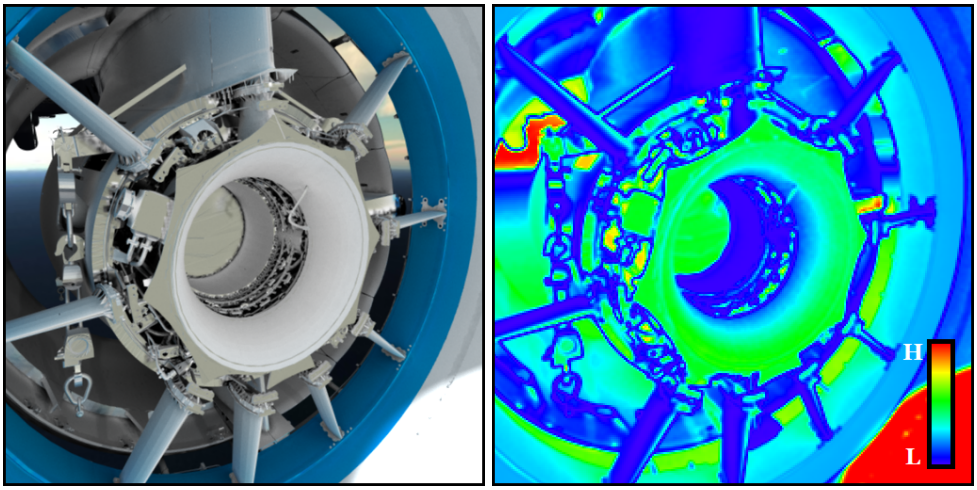


Figure 5.7: The right image visualizes a saliency map of the left image. Red colors indicate high saliency, while blue colors low saliency.

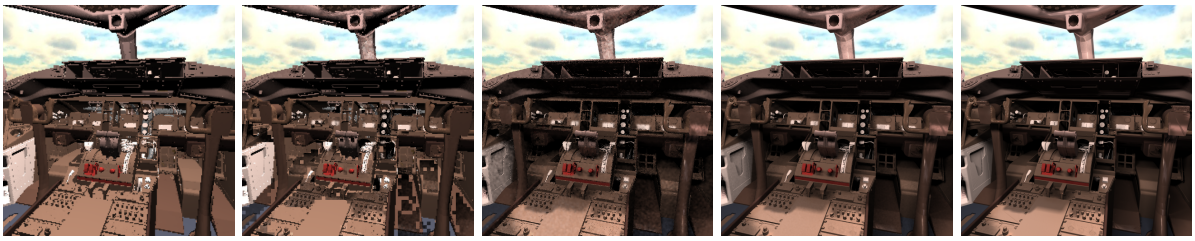
We have tested with different tile ordering including our saliency-based, z-curve based, random, and row-by-row ordering. We observed that z-curve based ordering shows the best performance followed by row-by-row, ours, and random ordering. Nonetheless, the performance differences between our method and 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. 5.2), 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. 5.4.2. 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.

5.3.3 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 [122] 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



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

Figure 5.8: 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.

utilization of GPU, we process a bundle of rays simultaneously by considering the SIMT architecture of the modern GPUs [123]; 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 [75]. In particular, we sort the rays based on their ray directions and then assign rays with similar directions to a warp.

5.3.4 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. 5.2.

5.4 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 5.1) that have different characteristics. Our main benchmark model is Boeing 777 model (Fig. 5.1) 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

Model	Tri (M)	Size (MB) of			r_u	$r_u * r_l$
		HCCM.	u-ASVO	l-ASVO		
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 5.1: 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.

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 5.3) 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 5.3) with 1 and 2 area lights, respectively. Triangles in these models are distributed relatively regularly, but are highly tessellated.

5.4.1 Implementation Details

We elaborate implementation details that are important to achieve high performance of our method reported in this chapter.

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 5.1.

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 [105]. 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. [124].

5.4.2 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. 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. 5.1, following reference views listed by Wald et al. [76]. 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 [20].

As shown in Table 5.2, our approach shows the response time of 25.9 ms∼36.9 ms across different views when we use $n_s = n_g = 2$. These results directly indicate that users can get a feedback within this response time, even when they 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. Progressive results for the cockpit viewpoint are shown in Fig. 5.8.

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. 5.9). 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 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. 5.3.2.

			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 5.2: 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. 5.1. We also report complete image time, **CIT**, for comparison with other work. Time is reported in ms unit.

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 5.3 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 [125]. Even though it is not investigated further, it is straightforward

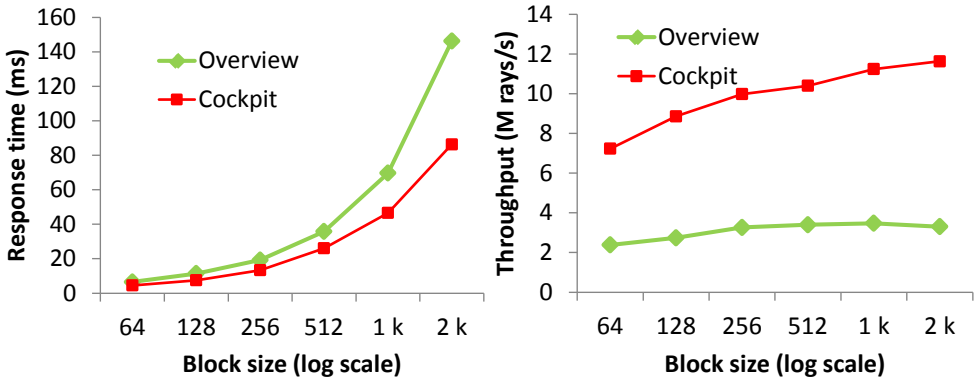


Figure 5.9: These graphs show response time and ray processing throughputs as a function of the fetching block sizes.

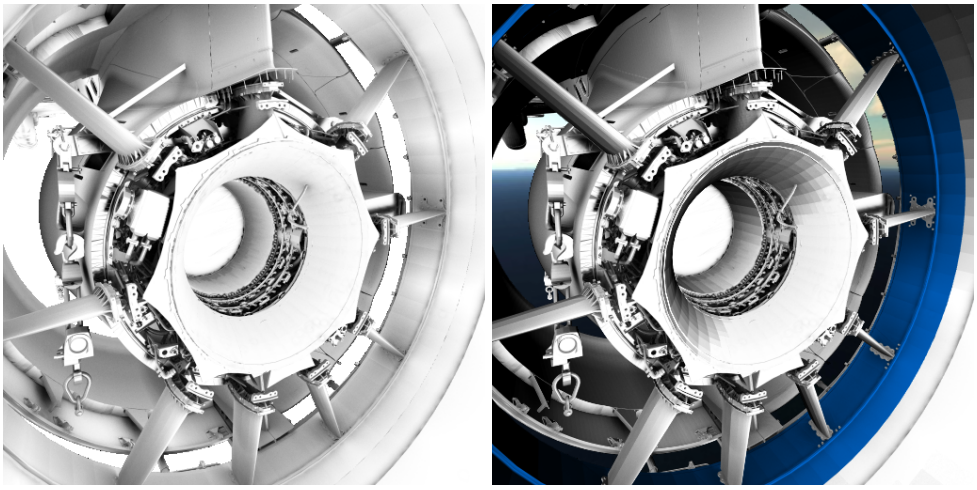


Figure 5.10: Ambient occlusion (AO) result on the left and the final image (on the right) combining AO and direct illumination.

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. 5.10).

5.4.3 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

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

Figure 5.11: 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.

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. 5.11); 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 [70, 78] 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 [74]. 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.

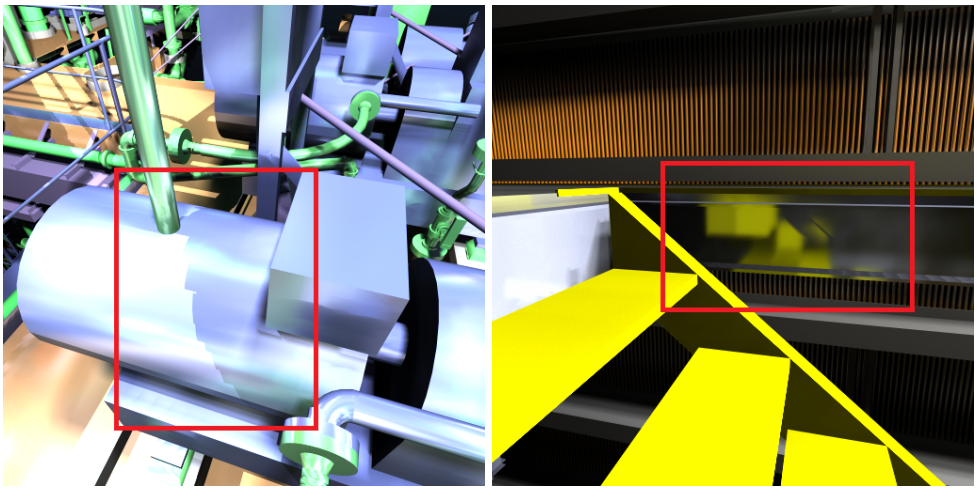
Comparisons with prior voxel octrees. Crassin et al. [68] proposed efficient voxel octrees as an volumetric LOD representation, and used the same representation for filtered (i.e. low-frequency effects) global illumination with small models that can fit into main memory [31]. At a high level there are 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 [30]. This is mainly because our voxel representation drastically reduces the computation of global illumination. The technique proposed by Wang et al. [30] 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.

5.5 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. 5.12). 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



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

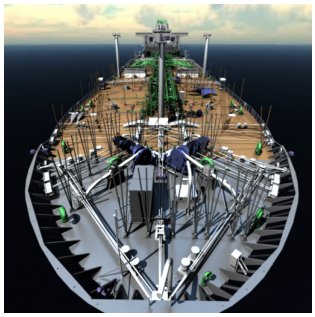
Figure 5.12: These images show artifacts caused by a limited resolution of our volumetric representation.

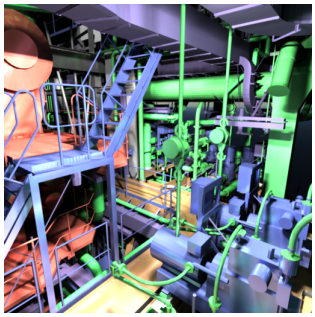
may have storage overheads for small models such as Sponza model because the ASVOs do not depend on number of primitives. Note that these are common drawbacks of voxel based ray tracing.

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 [77] 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.

There are many other interesting avenues for future work. Our method can be extended to adopt progressive photon mapping [126] 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.

View1	View2	

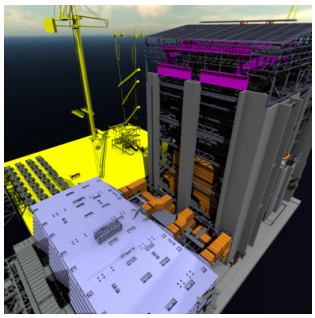


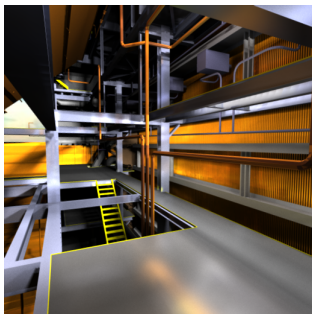


Double Eagle Tanker (82 M triangles)

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

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

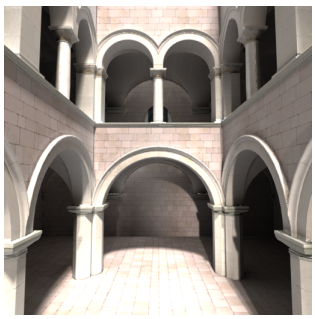




Power plant (13 M triangles)

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

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



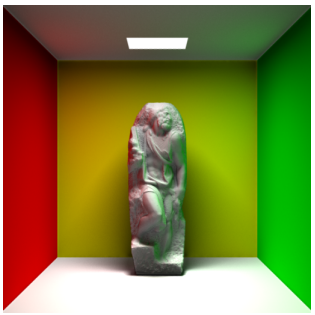
Sponza (66 k triangles)

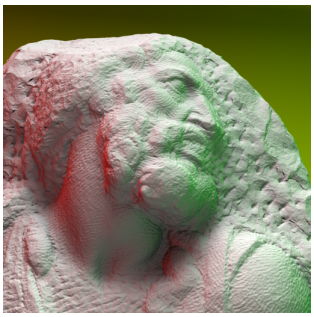
Size (MB) of HCCMesh	2.6
Size (MB) of u-ASVO	191

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

Table 5.3: 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.

View1	View2	

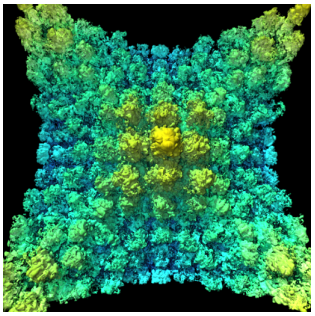


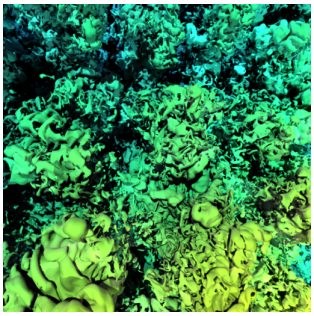


St. Matthew (372 M triangles)

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 5.4: (Continue) 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.

Chapter 6. Interactive System Optimization

As seen in Sec. 5.4, responsiveness and ray processing throughput depend on some parameters including size of fetching block, number of secondary rays per primary ray. Also, the responsiveness and throughput vary depending on test machines and view points. In the Sec. 5.4, the parameters are manually defined in a given test machine and a view point to maximize both responsiveness and throughput. Obviously, better parameters can be adopted for different test environments, or even for the tested environment in the previous chapter. In this chapter, we design an automatic and systematic algorithm to determine such parameters to maximize both responsiveness and rendering throughput. Since high rendering throughput gives high rendering quality in general and users actually care about the quality more than the throughput, we set our optimization goal to maximize rendering quality while minimizing response time.

In the following sections, we define some metrics for the rendering quality and the responsiveness than fomulate optimization function.

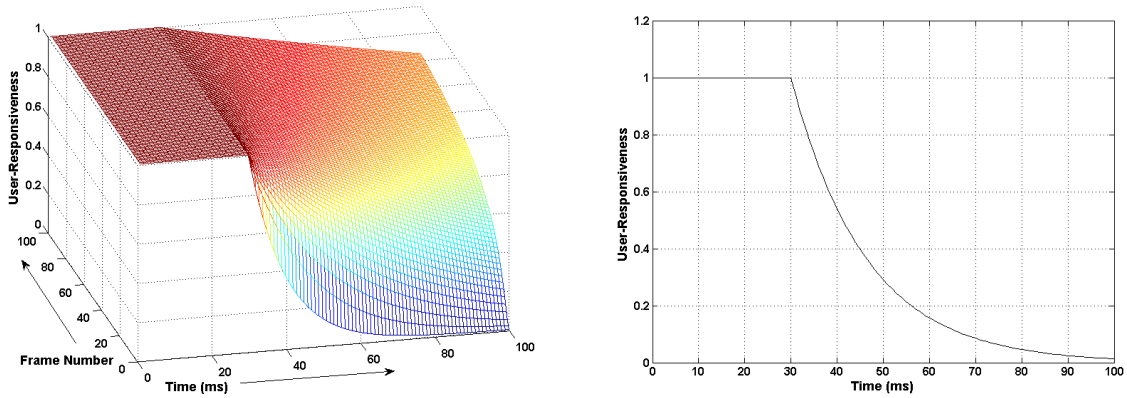
6.1 Overview

Proposed system is targeted to users who design large-scale scenes interactively. When users change scene attributes such as lights or material information, rendering results with global illumination are interactively shown. The results are progressively refined for the high responsiveness. Then users easily can estimate final look of their design in advance. We assume that computation time of each atomic operation is predictable. Most of rendering algorithm meet the assumption because the time costs of the atomic operations have well-known relationship with parameters; for ray tracing based rendering framework, the processing time is linearly increased to number of ray samples.

At every frame, the optimization system find parameters which maximize quality improvement and minimize response time. This can be formulated as following:

$$\max_{X_f} \left(\Delta Q(f, X_f) \times R(f, T(X_f)) \right) \tag{6.1}$$

where X_f is a vector of parameters at frame f, Q is quality of an image, $\Delta Q(f, X_f)$ is quality improvement at frame f using X_f, $T(X_f)$ is processing time using X_f, and $R(f, t)$ is an user-responsiveness metric at frame f and time t. The user-responsiveness metric reflects users' satisfaction depending on response time. For example, the value R is drastically decreased when the response time is longer than the time threshold of user's perception. Note that any quality metric or user-responsiveness metric can be adopted. However, we need to estimate the quality with X_f before using the X_f. In this thesis, we use the $PSNR$ (Peak Signal-to-Noise Ratio) which is the most commonly used in image processing field. Also, we propose a frame-varying user-responsiveness metric (Sec. 6.2).



(a) User-Responsiveness Metric

(b) User-Responsiveness Metric at Frame 2

Figure 6.1: The left figure visualizes a user-responsiveness metric (6.4) with $\alpha = 0.1$, $\beta = 0.7$, and $t_{thr} = 30$. The right figure shows the metric at frame 2.

6.2 Metrics

The $PSNR$ is defined as:

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX^2}{MSE} \right) \tag{6.2}$$

where MSE is defined as:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (P_i - \hat{P}_i)^2 \tag{6.3}$$

where P_i is intensity of i-th pixel of current image, $\hat{P}_i$ is intensity of i-th pixel of reference image, and n is number of pixels in an image. Note that a reference image is necessary to compute the $PSNR$. Since it is not able to make the reference image in advance, we estimate the MSE by using pixel variance then compute the $PSNR$. See Sec. 6.3 for the details of the estimation.

We propose a user-responsiveness metric which reflects human's perception and target user's preference. Similar to [127], we assume that "events that complete in a threshold or less are believed to have imperceptible latency and do not contribute to user dissatisfaction". In addition, we assume that a user prefers different quality-responsiveness trade offs depending on two situations; when the user dynamically modify a scene, the user prefer immediate response to the modification. On the other hand, the user prefer fast convergence, or much quality improvement, when the user stay a particular view to carefully see details of the rendering results. In order to reflect the preference, we apply a frame number to our user-responsiveness metric as follows:

$$R(f,t) = \begin{cases} 1 & t < t_{thr} \\ e^{-\alpha(t-t_{thr})f^{\beta}} & t \geq t_{thr} \end{cases} \tag{6.4}$$

where t_{thr}, α, and β are user specific values and f is a frame number. R reflects the fact that human does not distinguish the time differences which below a threshold t_{thr} (e.g., 30 ms). Fig. 6.1(a) visualizes the user-responsiveness metric as a function of frame and time, and Fig. 6.1(b) shows the metric at a specific frame.

As a result, the user has maximum satisfaction when the result is shown within t_{thr}. If the result is shown after the threshold t_{thr} and the user percepts the delay, the satisfaction is exponentially decreased

as time goes on. The slope of the decrement is determined by α and f^{β}. When the frame number f is high which means the user is staying a particular view, the slope will be become low, in other words, the rendering system could use more time to produce more quality improvement within a frame. For example, say the estimated quality improvement within 10 ms is 10 dB and the improvement using 50 ms is 30 dB, and $R(10) = 1, R(50) = 0.5$. In this case, the system chooses to use 50 ms since the value of optimization function 6.1 is bigger than with 10 ms.

Note that we can easily apply the hard-constraint with benefit and cost heuristics presented by Funkhouser and Carlo [128] by using a step function:

$$R'(f,t) = \begin{cases} 1 & t < t_{thr} \\ 0 & t \geq t_{thr} \end{cases} \tag{6.5}$$

The benefit and cost heuristics based optimization is formulated as follow:

$$\begin{aligned} &\text{Maximize :} \\ &\qquad \textstyle\sum_{job} Benefit(...) \\ &\text{Subject to :} \\ &\qquad \textstyle\sum_{job} Cost(...) \leq \text{TargetFrameTime} \end{aligned}$$

When we define the $Benefit$ as estimated quality improvement, ΔQ, and $Cost$ as estimated processing time, T, and use R' for the constraint, the optimization problem becomes equivalent to our optimization problem.

6.3 Estimation of Rendering Quality

There are two difficulties to compute ΔQ in the optimization function 6.1 using $PSNR$ as the quality metric; we cannot compute neither $PSNR$ nor MSE of rendered image because we do not have the reference image, and even though we can, we cannot compute the rendered image as well before the rendering process. Note that our goal is to find a vector of optimized parameters for the rendering process. Therefore, in this thesis, we estimate the $PSNR$ and then use the estimated $PSNR$ as our quality metric. Note that if a quality metric for the parameters is computable for a rendering system, the metric can be directly applied to our optimization function.

In order to estimate the increment of $PSNR$ by using a vector of parameters, we estimate a MSE of each pixel then compute MSE of the frame. Note that MSE of an image or a frame is different from MSE of a pixel; The MSE of an image is computed by using two images and the pixel MSE is computed by using statistics of samples which contributes the pixel. However, the MSE of an image can be computed by averaging the MSE of all pixels. A pixel MSE of current image can be decomposed to variance and bias of samples; $MSE(P_i) = var(P_i) + bias(P_i, \hat{P}_i)^2$ where P_i is intensity of i-th pixel of current image and P_i is intensity of i-th pixel of reference image. If the target rendering system is unbiased, the pixel MSE is equal to the variance of the pixel. For the proposed rendering system, *T-ReX* in chapter 5, we ignore the bias term of the pixel MSE because the rendering system is already biased. Therefore, we only consider pixel variance to estimate the pixel MSE.

A pixel intensity is an average of intensities of samples related to the pixel. Since the sample intensity is i.i.d. in T-ReX,

$$var(P_i) \propto \frac{1}{n} \quad \text{or} \quad var(P_i) = \frac{k}{n} \tag{6.6}$$

where n is a number of samples which contribute the P_i and k is a proportional constant. Therefore, we can easily estimate $var(P_i)$, hence $MSE(P_i)$ with $n+1$ samples by simple regression. Similarly, if all pixel has same number of samples, n, the $PSNR$ of an image can be estimated:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX^2}{MSE}\right) = 10 \cdot \log_{10}\left(\frac{MAX^2 \cdot n}{k}\right) = c_1 \log_{10} n + c_2 \tag{6.7}$$

where c_1 and c_2 are constants. Fig. 6.2(a) shows measured $PSNR$ as a function of number of samples per pixel. We use a converged rendering image (more than 100 k frames) for the reference. We have tested with cockpit viewpoint (Fig. 5.1(b)). Note that x-axis has a log scale. In conclusion, the measured $PSNR$ is well fit the derived function 6.7 as seen the Fig. 6.2(a).

We compute variance of pixel intensity, $var(P)$, at every frame for all pixels. This process is memory-efficiently done by maintaining summation of intensities and squared intensities per pixel , rather than storing all previous samples. Since we cannot compute the variance at initial frames, we use *preview* image (See Sec. 5.1) for the initial frames for the T-ReX system. Fig. 6.2(b) shows estimated $PSNR$ as a function of number of samples per pixel, based on the variance. The function of the estimated $PSNR$ does not have similar shape of measured $PSNR$ and even the estimated $PSNR$ is not expected to asymptotically converged to the measured $PSNR$. We have realized that this is because the computed pixel variance includes not only variance from sampling but also variance from features themselves (e.g., edges). In order to remove the variance from features, we use dual buffers for the variance and evenly distribute the samples to the buffers, inspired by [129]. Then compute each variance of buffers and then use subtract values as the sample variance. Fig. 6.2(c) shows estimated $PSNR$ by using the dual buffering. As seen the figure, the estimated $PSNR$ is asymptotically converged to the measured $PSNR$. One drawback of the dual buffering is that it needs double samples to get reasonably accurate variances (e.g., more than 60 samples).

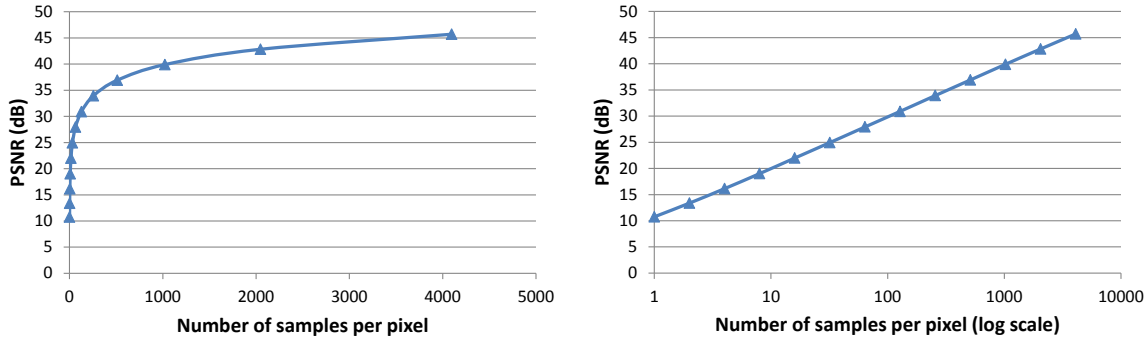
Now we can estimate $PSNR$ for a rendered image if all pixel has same number of samples. Also, we can estimate future $PSNR$ by using regression of previous samples with equation 6.7. However, pixels of a frame are not always have same samples for the progressive rendering system. Especially for the adaptive rendering system, each pixel can have various number of samples. In the case of our target rendering system, T-ReX, difference of number of samples between two pixels is at most one. In other words, samples are evenly distributed to all pixels through several frames; we call this number of frames *complete image frame, CIF*. Therefore, we can estimate the $PSNR$ for each CIF.

6.3.1 Estimation of Quality Improvement

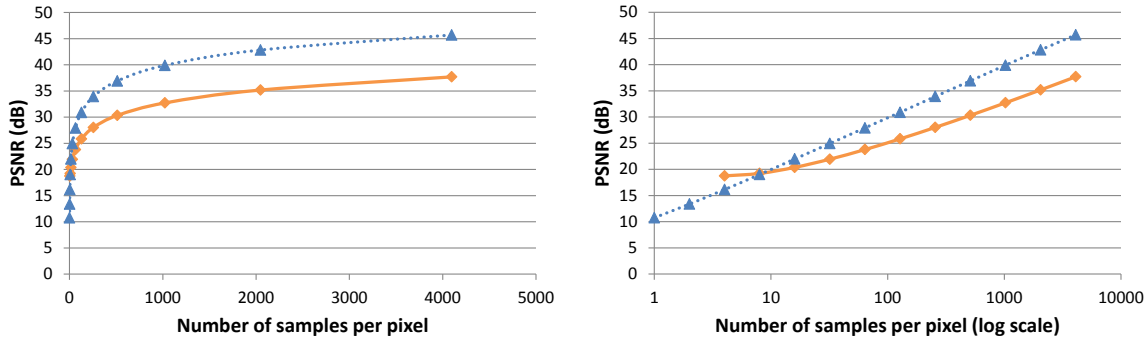
ΔQ can be approximated as a total derivative form:

$$\Delta Q \approx \sum_{x \in X} \left(\frac{\partial Q}{\partial x} \Delta x\right) \text{ , where } \Delta x \text{ is small} \tag{6.8}$$

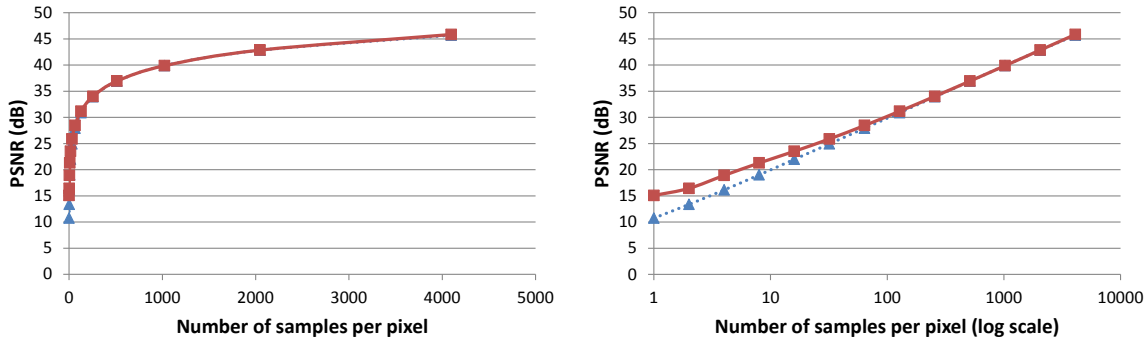
We approximate $\partial Q / \partial x$ by using difference of estimated quality of two consecutive CIFs, $Q(F)$ and $Q(F+1)$, where F is a CIF. To verify the validity of the linear approximation, we have measured $PSNR$ as a function of number of samples (not per pixel) starting from a CIF. Fig. 6.3 shows the measured $PSNR$ starting from CIF 1, 2, 10, and 100, respectively. We are able to observe that the $PSNR$ is almost linear starting from CIF 2.



(a) Measured PSNR

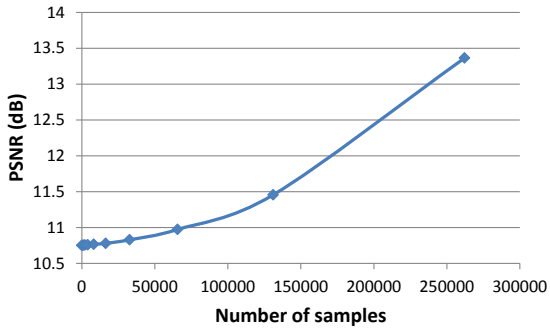


(b) Estimated PSNR using single buffer

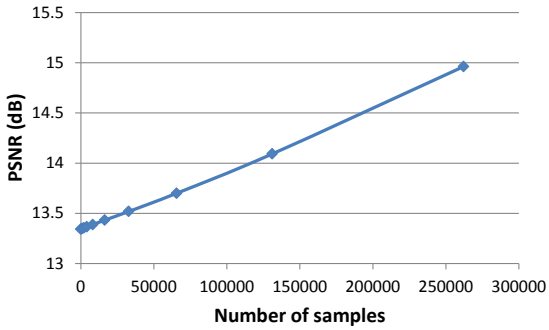


(c) Estimated PSNR using double buffers

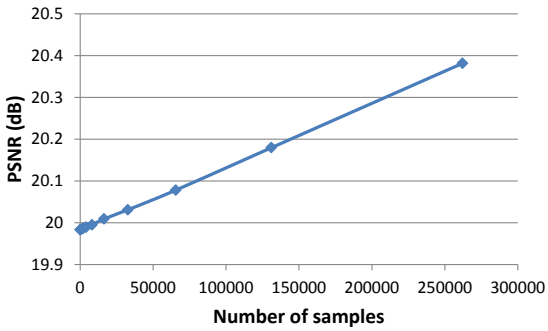
Figure 6.2: The top figures show measured $PSNR$ as a function of number of samples per pixel. Converged image (more than 100 k frames) of cockpit view point (Fig. 5.1(b)) is used as the reference image. The middle and bottom figures show estimated $PSNR$ using single and double buffers, respectively. Dotted lines are same as Fig. 6.2(a).



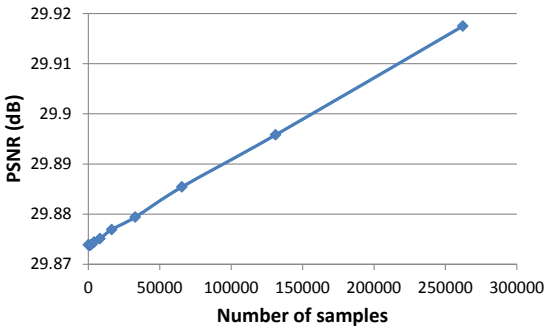
(a) Starting from CIF 1



(b) Starting from CIF 2



(c) Starting from CIF 10



(d) Starting from CIF 100

Figure 6.3: These figures show measured PSNRs as a function of number of samples starting from CIF 1, 2, 10, and 100. After CIF 2, the measured PSNR is linearly increased within a CIF.

6.4 Runtime Optimization

Solving Eq. 6.1 is a nonlinear programming. If the parameters to be found are integers, then the problem becomes nonlinear integer programming which is NP-hard. Since the optimization is integrated into interactive system, the optimization process is expected to be very efficient. The problem can be approximated by using piecewise linear functions to make the nonlinear programming (NLP) to linear programming (LP). Then we can solve the problem by using well-known, efficient LP solver. Finally, we further approximate the solution by simply rounding the parameters to make them integers.

In our implementation, we only use the parameters of power of two, then solve the optimization problem by exhaustive search since the numbers of the candidates are limited. Because of logarithm nature of the estimated quality as shown in Sec. 6.3, using the limited set of the candidates of the parameters gives reasonable results (See Sec. 6.5) with negligible computational cost (less than 1 ms).

As shown in Fig. 6.2(c), the quality optimization has errors in initial frames because of lack of samples. To reduce the errors of the initial frames, we cache well-optimized parameters for each view points. When the number of samples becomes enough (e.g., more than 100 samples per pixel) for a view point, we store the fitting parameters (e.g., c_1 and c_2 in Eq. 6.7). The fitting parameters are used for the initial, inaccurate frames when a user set same view point in the cache. In order to avoid divergence of the cache and increase reuseability of the cached data, we quantize position and direction space of the view points. We store the cached fitting parameters into external drives for next use.

6.5 Results

We have applied the proposed optimization method to T-ReX rendering system of Ch. 5. We have tested two different machines. One is a PC which has 3.3 GHz Intel Core i7 CPU (hexa-core), 24 GB RAM, NVIDIA GTX 680 graphics card with 2 GB DRAM, and HDD. Another one is a laptop computer which has 2.4 GHz Intel Core i7 CPU (quad-core), 8 GB RAM, NVIDIA GTX 670MX graphics card with 3 GM DRAM, and HDD.

As the parameters to be optimized, we use the size of fetching block (or number of primary rays per frame), the number of shadow rays per primary ray, and the number of gathering rays per primary ray. We have tested three rendering frameworks which of each has fixed but different parameters; parameters for maximizing responsiveness, used in previous T-ReX system, and for maximizing rendering throughput, respectively. Also, we have tested two rendering frameworks, Funkhouser and Carlo and proposed method in this chapter, both have dynamic parameter optimizations.

6.5.1 Performance

We have tested five rendering frameworks including fixed and dynamic parameter optimizations. For the fixed parameter optimizations, we have tested three configurations including parameters which maximize the responsiveness, which are used in the previous chapter, and which maximize the rendering throughput. For the dynamic parameter optimizations, we have tested Funkhouser and Carlo, and our method.

Table. 6.1 shows performance of the tested rendering frameworks in terms of response time and rendering throughput. We have tested on a PC (**M1**) and a laptop (**M2**) described at the beginning of this section and used overview (Fig. 5.1(a), **V1**) and cockpit (Fig. 5.1(b), **V2**) view points. We set $t_{thr} = 10ms$, $\alpha = 0.1$, and $\beta = 1.3$ as the user parameters of Eq. 6.4. Note that our optimization method

			Fixed parameters			Dynamic parameters	
			Max. Resp.	T-ReX	Max. Thr.	Funkhouser	OPT-TReX
M1	V1	Resp. T (ms)	**3**	23	76	18	**21∼91**
		M rays/s	1.8	31.1	**65.5**	7.3	**7.0∼65.3**
	V2	Resp. T (ms)	**5**	47	302	21	**22∼299**
		M rays/s	0.9	12.9	**14.9**	8.9	**8.9∼14.6**
M2	V1	Resp. T (ms)	**5**	47	188	10	**11∼186**
		M rays/s	1.0	15.8	**25.2**	6.7	**6.7∼25.1**
	V2	Resp. T (ms)	**9**	106	870	12	**10∼850**
		M rays/s	0.5	4.5	**5.1**	2.0	**2.0∼5.1**

Table 6.1: This table shows rendering performance including response time, **Resp. T**, and ray processing throughput measured in **M rays/s** with different parameters and optimization methods. **Max. Resp** indicates parameters for maximizing responsiveness are used and **Max. Thr** indicates parameters for maximizing throughput are used. **T-ReX** indicates parameters used in Ch. 5 are used. **Funkhouser** indicates optimization method with hard-constraint [128] and **OPT-TReX** indicates proposed optimization method in this chapter. **M1** and **M2** indicate test machine of PC and laptop, respectively. **V1** and **V2** indicate test view points of overview (Fig. 5.1(a)) and cockpit (Fig. 5.1(b)), respectively.

has a range of response time and rendering throughput between Funkhouser and Carlo and framework for maximizing throughput. Following our design choice, our method optimize the parameters for high responsiveness at the initial frames, then change the parameters for higher rendering throughput. As an example, Fig. 6.4 shows the changes of number of shadow rays and gathering rays per a primary ray.

6.5.2 User Study

To evaluate benefits of our method for a user who performs a design task interactively, we perform a user study. We have tested four rendering frameworks of Sec. 6.5.1. We exclude the frameworks with parameters which are used in the previous chapter, because the parameters are manually optimized for the test machine; it is expected to have similar performance to the best case. Following experiments of Ou et al. [130], subjects were asked to perform three kinds of *matching trials* for each configurations above. In the matching trials, subjects adjust only one parameter which change a light or material information to match a given, target image.

The matching trials include changing the light position, size of the light area, and glossiness of the material. We use the cockpit scenes (Fig. 6.5(a) and 6.5(e)) for the changing the light information (the position and the size of the area) and a window scene (Fig. 6.5(i)) for the changing material information. Eleven expert subjects participated in the experiments. All subjects were researching on graphics and had a good understanding of lighting, show effects from area light, and shading glossy materials. To avoid learning effects, we first trained the subjects with our main comparison target, Funkhouser and Carlo, with all tasks for more than ten minutes. Also, we randomized the starting and target parameter configurations when the subjects move their tasks. In addition, we asked the subjects to perform all the task using a rendering system, then move to another system.

As the test machine, the PC is used. The parameters the subjects adjusting consist of a hundred levels (0.00 to 0.99) and we set the thresholds of 5%. When the subjects have decision, the subjects were asked to submit their decision by clicking a button. If the adjusted parameter of the decision is within

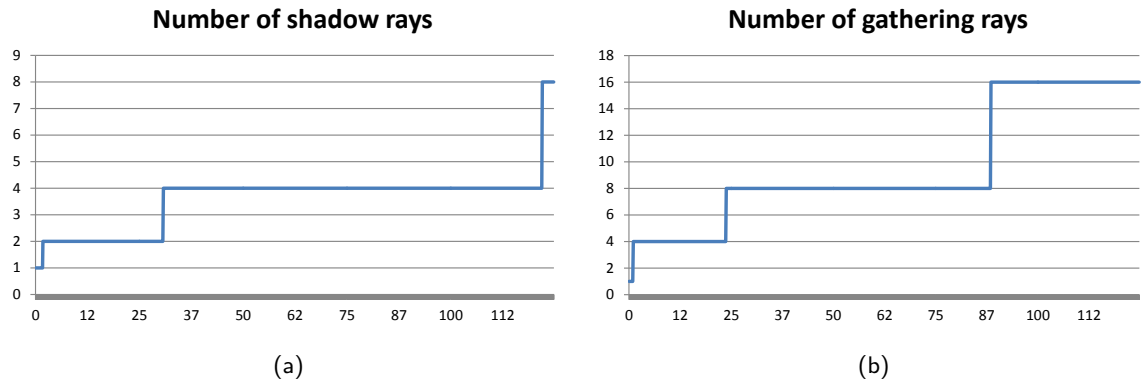


Figure 6.4: Optimization results of the number of shadow rays and gathering rays per a primary ray at each frame are shown.

the threshold, we record the task time. If not, we let the subjects to try again. We have counted the number of trials.

Fig 6.5.2 shows relative time to finish the matching trials. The right most results of the figure shows average time. We have observed that our method shows best performance on average followed by Funkhouser and Carlo, maximum throughput, and maximum responsiveness system for the three tasks.

6.6 Conclusion

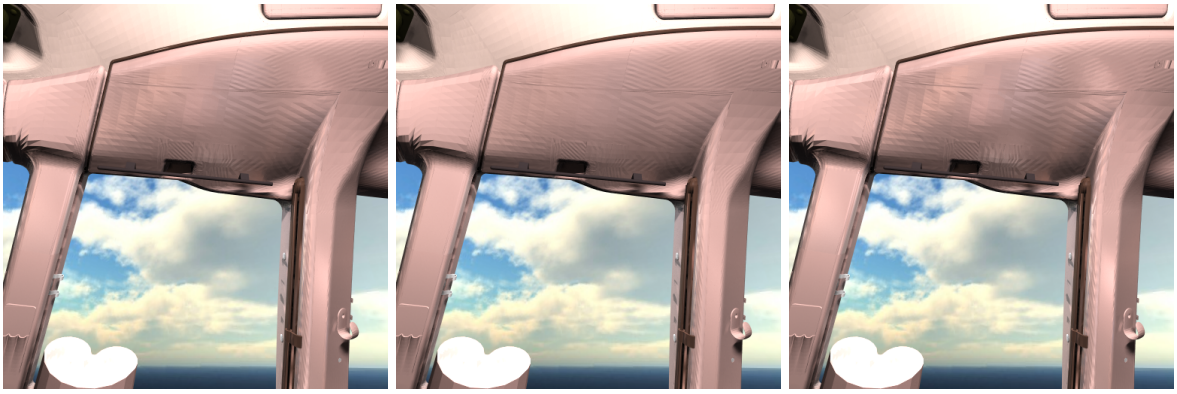
We propose an analytic approach to optimize the parameters given a rendering system. We propose a novel user-responsiveness metric which is dynamically adopted by a number of frames. By using the metric and quality estimation, the system automatically find the optimized parameters which maximize both responsiveness and rendering throughput for best user experience. Our method shows the robust performance, not depending on test machine. We also shows the benefits by evaluating with user study.

(b) Reference (c) Parameter = 0.00 (d) Parameter = 0.99

(d) Changing light position

(f) Reference (g) Parameter = 0.00 (h) Parameter = 0.99

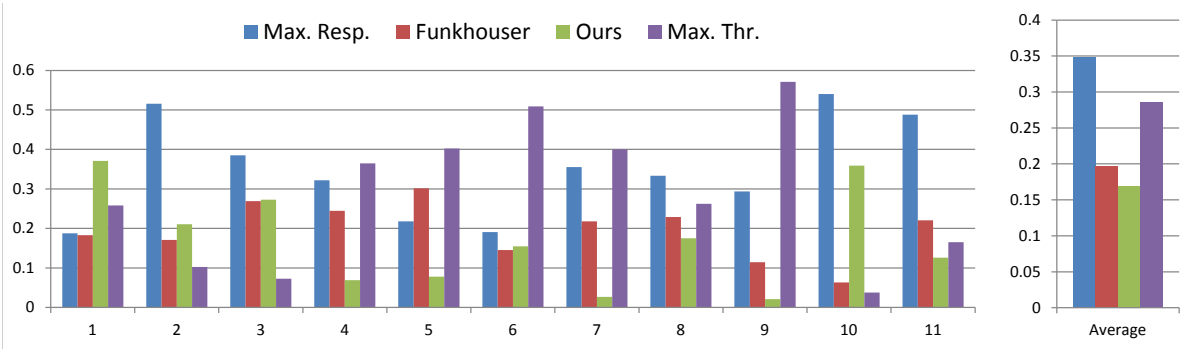
(h) Changing size of light area



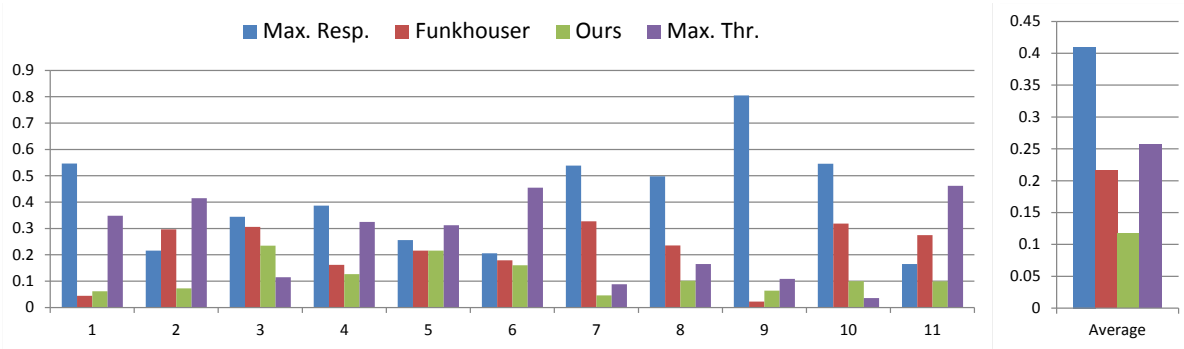
(j) Reference (k) Parameter = 0.00 (l) Parameter = 0.99

(l) Changing glossiness of material

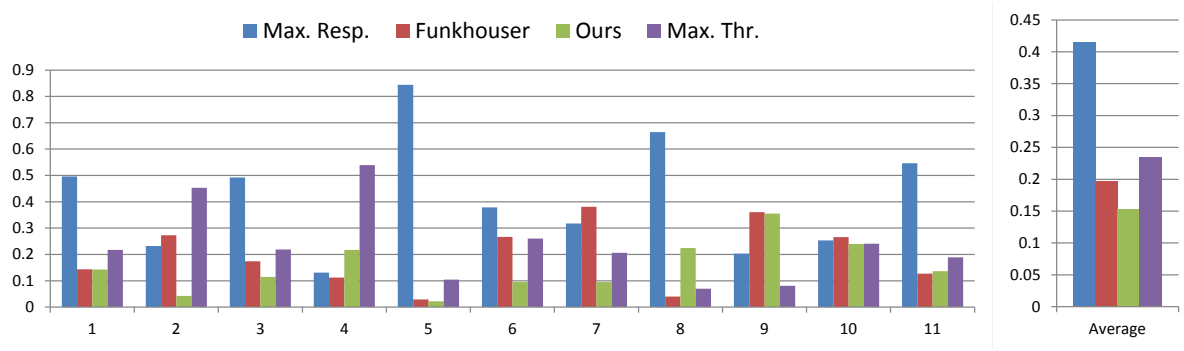
Figure 6.5: Subjects were asked to adjust a parameter between 0.00 to 0.99 to match the rendering results to the reference (the left most image). Adjusting the parameter changes the light or the material information.



(a) Changing light position



(b) Changing size of light area



(c) Changing glossiness of material

Figure 6.6: These figures show relative time to finish the matching trials of changing the light position, the size of the light area, and glossiness of the material.

Chapter 7. Conclusion

In this thesis, we have presented the problem of rendering large-scale models supporting dynamic changing of information of lights and materials. To achieve high rendering performance, we have proposed 1) in-core and out-of-core compression techniques to reduce data transmission costs, 2) utilization of heterogeneous computing resources, and 3) system optimization to robustly support both high rendering throughput and responsiveness.

We have presented a novel compression and runtime BVH decompression framework that transparently supports random access on the compressed BVHs. Our compression method preserves the original layout of a BVH and sequentially compresses BVs of a BVH. In order to support random access on the compressed BVHs, we decompose an input BVH into a set of clusters. Each cluster contains consecutive BV nodes and serves as an access point at runtime. We propose a general BVH access API to transparently support random access on our RACBVH representation. Our decompression framework selectively fetches, decompresses, and stores data in our in-core BVH representation. We have demonstrated the benefits of our methods on two applications having different characteristics. We achieved up to a 12:1 compression ratio and up to a 4:1 runtime performance improvement in the tested benchmarks.

In addition, we have presented a HCCMesh representation, which tightly integrates a mesh and a BVH. We believe that our HCCMesh representation is the first method that has been tested on various applications including rendering and collision detection that require the random hierarchical traversal. The i-HCCMesh and o-HCCMesh achieved 3.6:1 and 10.4:1 compression ratios on average over a naively compressed representation respectively. We can reduce the memory requirement of handling massive models and thus can handle models ten times larger without the expensive disk I/O thrashing. Moreover, by avoiding the disk I/O thrashing, we observed performance improvements by up to two orders of magnitude, compared to the original and naively compressed representations. Also, even if our HCCMeshes cannot fit into main memory, we expect that our method would improve performances by a factor close to its compression ratios to the original and other compressed representations.

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.

7.1 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. 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 differ-

entials [77] 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.

There are many other interesting avenues for future work. Our method can be extended to adopt progressive photon mapping [126] for better quality and better handling of dynamic changes. Also, we would like to apply our approaches to distributed rendering system such as render farm. We believe that our optimization process can be adopted to the clustering system with extension of the process to consider the network configurations. Similarly, we also would like to extend the techniques to mobile system whose running environments are much limited compared to PC.

References

[1] S. Choe, J. Kim, H. Lee, S. Lee, and H.-P. Seidel, "Mesh compression with random accessibility," in *Israel-Korea Bi-National conf.*, 2004, pp. 81–86.

[2] S. Choe, J. Kim, H. Lee, and S. Lee, "Random accessible mesh compression using mesh chartification," *IEEE Trans. on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 160–173, 2009.

[3] S.-E. Yoon and P. Lindstrom, "Random-accessible compressed triangle meshes," *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization)*, vol. 13, no. 6, pp. 1536–1543, 2007.

[4] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli, "C-bdam - compressed batched dynamic adaptive meshes for terrain rendering," *Computer Graphics Forum*, vol. 25, no. 3, pp. 333–342, 2006.

[5] J. Kim, S. Choe, and S. Lee, "Multiresolution random accessible mesh compression," *Eurographics*, vol. 25, no. 3, pp. 323–332, 2006.

[6] J. L. Hennessy, D. A. Patterson, and D. Goldberg, *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, 2007.

[7] S.-E. Yoon, E. Gobbetti, D. Kasik, and D. Manocha, *Real-Time Massive Model Rendering.* Morgan & Claypool Publisher, 2008.

[8] J. T. Kajiya, "The rendering equation," in *Computer Graphics (SIGGRAPH '86 Proceedings)*, D. C. Evans and R. J. Athay, Eds., vol. 20, Aug. 1986, pp. 143–150.

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

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

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

[12] P.-P. Sloan, J. Kautz, and J. Snyder, "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments," in *SIGGRAPH.* New York, NY, USA: ACM, 2002, pp. 527–536.

[13] J. Lehtinen, "A framework for precomputed and captured light transport," *ACM Trans. Graph.*, vol. 26, no. 4, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1289603.1289604

[14] R. Ramamoorthi, "Precomputation-based rendering," *Found. Trends. Comput. Graph. Vis.*, vol. 3, no. 4, pp. 281–369, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1561/0600000021

[15] R. Ng, R. Ramamoorthi, and P. Hanrahan, "All-frequency shadows using non-linear wavelet lighting approximation," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers.* New York, NY, USA: ACM, 2003, pp. 376–381.

[16] X. Liu, P.-P. J. Sloan, H.-Y. Shum, and J. Snyder, "All-frequency precomputed radiance transfer for glossy objects," in *Rendering Techniques*, 2004, pp. 337–344.

[17] R. Wang, J. Tran, and D. Luebke, "All-frequency relighting of glossy objects," *ACM Trans. Graph.*, vol. 25, no. 2, pp. 293–318, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1138450.1138456

[18] Y.-T. Tsai and Z.-C. Shih, "All-frequency precomputed radiance transfer using spherical radial basis functions and clustered tensor approximation," in *ACM SIGGRAPH 2006 Papers*, ser. SIGGRAPH '06. New York, NY, USA: ACM, 2006, pp. 967–976. [Online]. Available: http://doi.acm.org/10.1145/1179352.1141981

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

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

[21] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila, "Incremental instant radiosity for real-time indirect illumination," in *Proceedings of Eurographics Symposium on Rendering*, 2007, pp. 277–286.

[22] 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.

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

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

[25] 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.

[26] M. Mittring, "Finding next gen: Cryengine 2," in *ACM SIGGRAPH 2007 courses*, ser. SIGGRAPH '07. New York, NY, USA: ACM, 2007, pp. 97–121. [Online]. Available: http://doi.acm.org/10.1145/1281500.1281671

[27] T. Ritschel, T. Grosch, and H.-P. Seidel, "Approximating dynamic global illumination in image space," in *Symp. on I3D Graphics and Games*, 2009, pp. 75–82.

[28] M. McGuire and D. Luebke, "Hardware-accelerated global illumination by image space photon mapping," in *Proceedings of the Conference on High Performance Graphics 2009.* ACM, 2009, pp. 77–89.

[29] C. Dachsbacher and M. Stamminger, "Reflective shadow maps," in *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ser. I3D '05. New York, NY, USA: ACM, 2005, pp. 203–231. [Online]. Available: http://doi.acm.org/10.1145/1053427.1053460

[30] 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.

[31] 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.

[32] P. Alliez and C. Gotsman, "Recent advances in compression of 3d meshes," *Advances in Multiresolution for Geometric Modelling*, pp. 3–26, 2004.

[33] C. Gotsman, S. Gumhold, and L. Kobbelt, "Simplification and compression of 3d meshes," in *Tutorials on Multiresolution in Geometric Modelling.* Springer, 2002, pp. 319–361.

[34] C. Touma and C. Gotsman, "Triangle mesh compression," in *Graphics Interface*, 1998, pp. 26–34.

[35] O. Devillers and P.-M. Gandoin, "Geometric compression for interactive transmission," in *IEEE Visualization*, 2000, pp. 319–326.

[36] J. Peng and C.-C. J. Kuo, "Geometry-guided progressive lossless 3d mesh coding with octree (ot) decomposition," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 609–616, 2005.

[37] P. Alliez and M. Desbrun, "Valence-driven connectivity encoding for 3D meshes," in *Eurographics*, 2001, pp. 480–489.

[38] F. Kälberer, K. Polthier, U. Reitebuch, and M. Wardetzky, "Freelence - coding with free valences," *Comput. Graph. Forum*, vol. 24, no. 3, pp. 469–478, 2005.

[39] M. Isenburg and J. Snoeyink, "Face Fixer: Compressing polygon meshes with properties," in *SIGGRAPH*, 2000, pp. 263–270.

[40] S. Gumhold and W. Strasser, "Real time compression of triangle mesh connectivity," in *SIGGRAPH*, 1998, pp. 133–140.

[41] J. Rossignac, "Edgebreaker: Connectivity compression for triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 1, pp. 47–61, 1999.

[42] H. Lee, P. Alliez, and M. Desbrun, "Angle-analyzer: A triangle-quad mesh codec," *Computer Graphics Forum*, vol. 21, pp. 383–392, 2002.

[43] D. L. Gall, "Mpeg: a video compression standard for multimedia applications," *Communications of the ACM*, vol. 34, pp. 46–58, 1991.

[44] I. Ihm and S. Park, "Wavelet-based 3d compression scheme for interactive visualization of very large volume data," *Computer Graphics Forum*, vol. 18, no. 1, pp. 3–15, 1999.

[45] F. Rodler, "Wavelet based 3D compression with fast random access for very large volume data," in *Pacific Graphics*, 1999, pp. 108–117.

[46] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," in *SIGGRAPH*, 2006, pp. 579–588.

[47] J. Katajainen and E. Makinen, "Tree compression and optimization with applications," *International Journal of Foundations of Computer Science*, vol. 1, no. 4, pp. 425–447, 1990.

[48] S. Zaks, "Lexicographic generation of ordered trees," *Theoretical computer science*, pp. 63–82, 1980.

[49] D. Zerling, "Generating binary trees using rotations," *Journal of ACM*, vol. 32, no. 3, pp. 694–701, 1985.

[50] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, 1991.

[51] D. Cline, K. Steele, and P. K. Egbert, "Lightweight bounding volumes for ray tracing," *Journal of Graphics Tools*, vol. 11, no. 4, pp. 61–71, 2006.

[52] J. Mahovsky, "Ray tracing with reduced-precision bounding volume hierarchies," PhD Thesis, University of Calgary, 2005.

[53] P. Terdiman, "Opcode: Optimized collision detection," 2003. [Online]. Available: http://www.codercorner.com/Opcode.htm

[54] S. Rusinkiewicz and M. Levoy, "Qsplat: A multiresolution point rendering system for large meshes," *SIGGRAPH*, pp. 343–352, 2000.

[55] E. Hubo, T. Mertens, T. Haber, and P. Bekaert, "The quantized kd-tree: Efficient ray tracing of compressed point clouds," in *IEEE Symp. on Interactive Ray Tracing*, 2006, pp. 105–113.

[56] C. Lauterbach, S.-E. Yoon, and D. Manocha, "Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing," in *IEEE/EG Symposium on Interactive Ray Tracing*, 2007, pp. 19–26.

[57] 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.

[58] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," *Computer Graphics Forum*, vol. 19, no. 1, pp. 61–81, 2005.

[59] V. Havran, "Cache sensitive representation for the bsp tree," in *Proc. of Compugraphics*, 1997.

[60] S. Lefebvre and H. Hoppe, "Compressed random-access trees for spatially coherent data," in *Eurographics Symposium on Rendering*, 2007, pp. 339–349.

[61] G. Jacobson, "Space-efficient static trees and graphs," in *Symp. on Foundations of Computer Science*, 1989, pp. 549–554.

[62] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," Ph.D. dissertation, Computer Graphics Group, Saarland University, 2004.

[63] ——, "On fast Construction of SAH based Bounding Volume Hierarchies," in *EG/IEEE Symposium on Interactive Ray Tracing*, 2007, pp. 33–40.

[64] P. H. Christensen, "Point-based approximate color bleeding," Pixar Animation Studios, Tech. Rep., 2008.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] C. Crassin, "Beyond programmable shading: Dynamic sparse voxel octrees for next-gen real-time rendering," in *ACM SIGGRAPH 2012 courses*, 2012.

[70] A. T. Afra, "Interactive ray tracing of large models using voxel hierarchies," *Comput. Graph. Forum*, vol. 31, no. 1, pp. 75–88, 2012.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

[76] 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.

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

[78] 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.

[79] 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.

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

[81] B. L. Saec and C. Schlick, "A progressive ray-tracing-based radiosity with general reflectance functions," in *Photorealism in Computer Graphics (Proceedings Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics, 1990)*, K. Bouatouch and C. Bouville, Eds., 1992, pp. 103–116.

[82] M. Feda and W. Purgathofer, "Progressive ray refinement for monte carlo radiosity," in *Fourth Eurographics Workshop on Rendering*, M. F. Cohen, C. Puech, and F. Sillion, Eds. Eurographics, Jun. 1993, pp. 15–26, held in Paris, France, 14–16 June 1993.

[83] F. Suykens and Y. Willems, "Adaptive filtering for progressive Monte Carlo image rendering," in *WSCG 2000 Conference Proceedings*. Citeseer, 2000.

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

[85] J. Chen, B. Wang, and J.-H. Yong, "Improved Stochastic Progressive Photon Mapping with Metropolis Sampling," R. Ramamoorthi and E. Reinhard, Eds. Prague, Czech Republic: Eurographics Association, 2011, pp. 1205–1213. [Online]. Available: http://diglib.eg.org/EG/CGF/volume30/issue4/v30i4pp1205-1213.pdf

[86] C. Knaus and M. Zwicker, "Progressive photon mapping: A probabilistic approach," *ACM Trans. Graph.*, vol. 30, pp. 25:1–25:13, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1966394.1966404

[87] T. Hachisuka, W. Jarosz, R. P. Weistroffer, K. Dale, G. Humphreys, M. Zwicker, and H. W. Jensen, "Multidimensional adaptive sampling and reconstruction for ray tracing," in *ACM SIGGRAPH*, 2008, pp. 33:1–33:10.

[88] R. S. Overbeck, C. Donner, and R. Ramamoorthi, "Adaptive wavelet rendering," in *ACM SIGGRAPH Asia 2009 papers*, ser. SIGGRAPH Asia '09. New York, NY, USA: ACM, 2009, pp. 140:1–140:12. [Online]. Available: http://doi.acm.org/10.1145/1661412.1618486

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

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

[91] J.-P. Farrugia and B. Peroche, "A progressive rendering algorithm using an adaptive perceptually based image metric," in *EUROGRAPHICS CONFERENCE PROCEEDINGS*, 2004.

[92] M.-M. Cheng, G.-X. Zhang, N. J. Mitra, X. Huang, and S.-M. Hu, "Global contrast based salient region detection," in *IEEE CVPR*, 2011, pp. 409–416.

[93] I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies," *ACM Transactions on Graphics*, vol. 26, no. 1, p. 6, 2007.

[94] L. Arge, G. S. Brodal, and R. Fagerberg, *Cache-Oblivious Data Structures in Handbook of Data Structures*. CRC Press, 2005.

[95] M. Isenburg and P. Lindstrom, "Streaming meshes," in *IEEE Visualization*, 2005, pp. 231–238.

[96] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space," *Inf. Process. Lett.*, vol. 6, pp. 80–82, 1977.

[97] S.-E. Yoon and D. Manocha, "Cache-efficient layouts of bounding volume hierarchies," *Computer Graphics Forum (Eurographics)*, vol. 25, no. 3, pp. 507–516, 2006.

[98] C. Silva, Y.-J. Chiang, W. Correa, J. El-Sana, and P. Lindstrom, "Out-of-core algorithms for scientific visualization and computer graphics," in *IEEE Visualization Course Notes*, 2002.

[99] M. Isenburg and S. Gumhold, "Out-of-core compression for gigantic polygon meshes," in *SIGGRAPH*, 2003, pp. 935–942.

[100] D. Salomon, *Data Compression.* Springer, 2007.

[101] S.-E. Yoon, D. Manocha, P. Lindstrom, and V. Pascucci, "Openccl," 2005. [Online]. Available: http://gamma.cs.unc.edu/COL/OpenCCL

[102] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, 1998.

[103] J. Klosowski, M. Held, J. S. B. Mitchell, K. Zikan, and H. Sowizral, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *IEEE Trans. Visualizat. Comput. Graph.*, vol. 4, no. 1, pp. 21–36, 1998.

[104] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1176–1185, 2005.

[105] 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.

[106] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," in *SIGGRAPH*, 1984, pp. 137–145.

[107] B. Mirtich and J. Canny, "Impulse-based simulation of rigid bodies," in *Symposium on Interactive 3D Graphics*, 1995, pp. 181–188.

[108] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts.* Jon Wiley and Sons, 2003.

[109] J. loup Gailly and M. Adler, "zlib," 2005. [Online]. Available: http://www.zlib.net

[110] L. Seiler, D. Carman, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, no. 3, 2008.

[111] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Streaming compression of triangle meshes," in *Symposium on Geometry Processing*, 2005, pp. 111–118.

[112] Y. Kim, J. Yu, X. Yu, and S. Lee, "Line-art illustration of dynamic and specular surfaces," *ACM Trans. on Graphics (SIGGRAPH ASIA)*, vol. 27, no. 5, pp. 156:1–156:10, Dec. 2008.

[113] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of Detail for 3D Graphics.* Morgan-Kaufmann, 2002.

[114] H. W. Jensen, *Realisic image synthesis using photon mapping.* A K Peters, 2001.

[115] M. Lin and D. Manocha, "Collision and proximity queries," *Handbook of Discrete and Computational Geometry*, 2003.

[116] S. Gottschalk, M. Lin, and D. Manocha, "OBB-Tree: A hierarchical structure for rapid interference detection," *Proc. of ACM Siggraph*, pp. 171–180, 1996.

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

[118] 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.

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

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

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

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

[123] 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.

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

[125] 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.

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

[127] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer, "Using latency to evaluate interactive system performance," in *Proceedings of the second USENIX symposium on Operating systems design and implementation*, ser. OSDI '96, 1996, pp. 185–199.

[128] T. A. Funkhouser and C. H. Sin, "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments," in *Computer Graphics (SIGGRAPH '93 Proceedings)*, J. T. Kajiya, Ed., vol. 27, Aug. 1993, pp. 247–254.

[129] F. Rousselle, C. Knaus, and M. Zwicker, "Adaptive rendering with non-local means filtering," in *SIGGRAPH Asia*, 2012.

[130] J. Ou, O. Karlík, J. Křivánek, and F. Pellacini, "Toward evaluating progressive rendering methods in appearance design tasks," *IEEE Computer Graphics and Applications (accepted for publication)*, 2012.

Summary

Interactive global illumination of non-deformable massive models

모델링 기술의 발전으로 최근 대용량 모델이 많이 생성되고 있다. 이러한 대용량 모델은 수억 개의 요소로 이루어져 있으며 수십 기가바이트 이상의 공간을 차지하기에 이른다. 이는 높은 비용의 데이터 접근을 야기하고 결과적으로 가시화 또는 렌더링의 심각한 성능 저하를 일으킨다. 또한 전역 조명(global illumination)에서는 데이터 접근 패턴을 응집시키기(coherent) 쉽지 않아 이러한 성능 저하를 더욱 부추기게 된다. 본 논문에서는 이러한 고비용의 데이터 전송 시간을 줄이기 위해 메모리에 상주하는 형식과 외부 저장장치에 저장된 형식의 두 단계의 압축 데이터 형식을 제안한다. 제안된 형식은 임의 접근을 요구하는 다양한 응용프로그램의 성능을 두드러지게 향상시킨다. 또한 성능을 더욱 향상시키기 위해서 본 논문에서는 CPU와 GPU 같은 다른 형태의 연산 장치를 최대한 활용하는 새로운 프레임워크를 제안한다. 제안된 프레임워크는 다른 형태의 연산 장치 사이에 데이터 전송을 극적으로 줄인다. 그 결과 본 논문에서는 전역 조명 효과가 포함된 대용량 모델의 렌더링이 실시간으로 가능하게 된다. 끝으로 렌더링 처리량과 대화형 반응성을 동시에 최대화 시키는 최적화 방법을 제안한다. 제안된 방법은 컴퓨터의 성능에 적응하여 성능에 상관없이 작동한다. 이러한 최적화 기법의 효용을 보이기 위해 본 논문에서는 사용자 연구(user study)를 시행한다.

감 사 의 글

박사학위는 혼자의 힘으로 할 수 없다는 것을 새삼 느끼고 있습니다. 이 논문을 준비하기 까지 수많은 사람들로부터 도움을 받았고 항상 감사하며 기억할 것입니다. 먼저 6년동안 부족한 저를 이끌어 주신 윤성의 지도교수님께 감사 드립니다. 항상 높은 목표를 지향하고 목표를 위해 선택과 집중을 해서 최선을 다해 꼼꼼히 준비하시는 모습은 저에게 큰 귀감이 되었습니다. 교수님의 지도가 없었다면 저의 성장은 지금에 크게 미치지 못하였을 것입니다. 남은 학생들에게도 변함없는 지지를 부탁드립니다. 그리고 바쁘신 와중에도 논문을 심사해 주시고 좋은 말씀을 해주신 박진아 교수님, 신인식 교수님, 김민혁 교수님, Xin Sun 박사님께도 감사드립니다. 특히 마이크로소프트 아시아 연구소에서 인턴으로 일할 때 저를 이끌어주시고 이후 연구에도 같이 참여해 좋은 의견을 많이 주신 Xin Sun 박사님께 깊은 감사를 드립니다. 그리고 우리 연구실 학생들... 논문 한 번 써보겠다고 시험 공부도 마다하고 같이 씨름해준 덕수형, 보창이형 그 땐 정말 고마웠어요. 항상 옆자리에 있었던 Pio, 매번 논문 영어 봐줘서 고마웠고 내가 영어 회화에 거부감이 적어진 것은 네 덕분이야. 그리고 포톤매핑을 구현하고 졸업한 용영이! 그 코드는 고쳐가면서 아직도 쓰고 있단다. 회사 잘 다니고 있는 근호, 종윤이, 종협아. 너희들이 있을 때는 참 든든했는데 가고 나니까 그렇게 빈자리가 크게 느껴질 수가 없네. 그리고 오성이 형과 Lin, 다른 층에 살고 연구 분야도 달라 함께할 시간이 적어서 아쉬웠어요. 사람 잘 챙기고 언제나 유쾌한 정환이형, 제게 없는 장점을 많이 가지고 있어서 항상 배우고 있어요. 그리고 언제나 궂은 일을 도맡아서 하는 재필이, 정인아, 앞으로 같이 일하게 될 사람들이 너희 같기만 하다면 뭐든지 믿고 맡길 수 있을 것 같아. 우리 렌더링 팀 후배들, 명환, 수민, 명배야, 내가 연구에 막혔을 때 항상 같이 고민해줘서 뚫고 나아갈 수 있었다. 정말 고마워. 연구실 분위기 메이커 가연, 밍양, 너희들의 큰 웃음 소리는 근심도 잊게 만드는 힘이 있어. 그리고 마이크로소프트 아시아 연구소의 이미란 이사님과 딸 민지, 제가 많이 힘들 때 좋은 조언 해주시고 엄마와 같은 마음으로 챙겨주셔서 정말 큰 힘이 되었습니다. 북경에서 같이 고생했던 재모형, 준영이형, 정태형, 형묵이형, 임현, 성철이, 진한이, 민정이, 대성이, 심지은이, 태성이, 인턴 생활 동안 많이 의지가 되었고 반 년간 함께 생활했던 그 경험은 결코 잊지 못할거에요. 그리고 눈치 없이 아직까지 찾아가도 반갑게 맞이해주는 우리 동틀무렵, 밴드 활동은 정말 최고의 활력소였고 평생의 자부심입니다. 또한 모자란 실력에도 계속 풋살 경기에 불러주시는 NS랩 사람들, 정말 감사 드립니다.

다른 누구보다 항상 저를 믿어주고 지지해주며, 연구 내용도 같이 고민해주고 방향도 잡아준 이지은에게 무한한 감사를 드립니다. 적당히 하고 싶어질 때 더욱 최선을 다할 수 있도록 북돋아주고 영어도 지도해 주는 등 나열을 하자면 끝이 없네요. 가끔씩 하는 말이지만 이러한 헌신적인 도움과 지지가 없었다면 저의 졸업은 불가능 했을 것입니다. 박사과정 동안 저에게 어떠한 영광이라도 있었다면 모두 이지은에게 돌리고 싶습니다. 끝으로 평생 저를 묵묵히 지켜봐 주시고 믿어주신 아버지, 어머니, 항상 해준 것이 없다고 미안해 하시지만 그렇게 생각하지 마세요. 지금까지 받아온 것만으로도 과분하고 더 바란 적은 한번도 없었습니다. 아버지 어머니는 저의 롤모델이자 가치 판단의 기준이십니다. 그리고 앞에서는 새침데기지만 밖에서는 오빠를 자랑하고 다니는 우리 동생 유미야, 항상 고맙고 계속 건강하길 바란다.

이 력 서

이　　름 : 김 태 준

생 년 월 일 : 1984년 4월 5일

E-mail 주 소 : tjkim@kaist.ac.kr

학　　력

2000. 3. – 2003. 2.　경북과학고등학교

2003. 3. – 2007. 8.　한국과학기술원 전산학과 (B.S.)

2007. 9. – 2009. 8.　한국과학기술원 전산학과 (Integrated(M.S.))

경　　력

2010. 9. – 2011. 2.　Microsoft Research Asia 연구인턴 (Internet Graphics 그룹, 사수:Xin Sun)

학 회 활 동

1. **Tae-Joon Kim**, Bochang Moon, Duksu Kim, and Sung-Eui Yoon, *RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies* , SIGGRAPH talk and poster, New Orleans (USA), August., 2009.
2. **Tae-Joon Kim**, Yongyoung Byun, Yongjin Kim, Bochang Moon, Seungyong Lee, and Sung-Eui Yoon, *HCCMeshes: Hierarchical-Culling oriented Compact Meshes* , Eurographics, Norrkoping (Sweden), May., 2010.
3. Bochang Moon, Youngyong Byun, **Tae-Joon Kim**, Pio Claudio, Hye-sun Kim, Yun-ji Ban, Seung Woo Nam, and Sung-Eui Yoon, *Cache-Oblivious Ray Reordering* , ACM SIGGRAPH, Vanouver (Canada), August., 2011.
4. **김태준**, 이종협, 김영현, 윤성의, *압축을 이용한 디지털 리소그래피 3차원 패턴의 효율적인 전송* , 차세대 리소그래피 학술대회, 대전 (대한민국), 2월., 2012.
5. 김영현, **김태준**, 윤성의, *대규모 전장공간의 실시간 렌더링 기법* , 한국군사과학기술학회, 경주 (대한민국), 6월., 2012.
6. Pio Claudio, Dohyeong Kim, **Tae-Joon Kim**, and Sung-Eui Yoon, *VDR-AM: View-Dependent Representation of Articulated Models* , International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), Pilsen (Czech), June., 2013.

연 구 업 적

1. **Tae-Joon Kim**, Bochang Moon, Duksu Kim, and Sung-Eui Yoon, *RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies* , IEEE Transactions on Visualization and Computer Graphics, vol. 16, no. 2, pp. 273-286, 2010.
2. **Tae-Joon Kim**, Yongyoung Byun, Yongjin Kim, Bochang Moon, Seungyong Lee, and Sung-Eui Yoon, *HCCMeshes: Hierarchical-Culling oriented Compact Meshes* , Computer Graphics Forum, vol. 29, no. 2, pp. 299-308, 2010.
3. Bochang Moon, Youngyong Byun, **Tae-Joon Kim**, Pio Claudio, Hye-sun Kim, Yun-ji Ban, Seung Woo Nam, and Sung-Eui Yoon, *Cache-Oblivious Ray Reordering* , ACM Transactions on Graphics, Vol. 29, No. 3, 2010 .
4. **Tae-Joon Kim**, Xin Sun, and Sung-Eui Yoon, *T-ReX: Interactive Global Illumination of Massive Models on Heterogeneous Computing Resources* , IEEE Transactions on Visualization and Computer Graphics, under revision (minor).