EUROGRAPHICS 2010 / T. Akenine-Möller and M. Zwicker
(Guest Editors)

Volume 29 (2010), Number 2

HCCMeshes: Hierarchical-Culling oriented Compact Meshes

Tae-Joon Kim[1], Yongyoung Byun[1], Yongjin Kim[2], Bochang Moon[1], Seungyong Lee[2], and Sung-Eui Yoon[1]

[1] KAIST [2] POSTECH
Project URL: http://sglab.kaist.ac.kr/HCCMesh

Abstract

Hierarchical culling is a key acceleration technique used to efficiently handle massive models for ray tracing, collision detection, etc. To support such hierarchical culling, bounding volume hierarchies (BVHs) combined with meshes are widely used. However, BVHs may require a very large amount of memory space, which can negate the benefits of using BVHs. To address this problem, we present a novel hierarchical-culling oriented compact mesh representation, HCCMesh*, which tightly integrates a mesh and a BVH together. As an in-core representation of the HCCMesh, we propose an i-HCCMesh representation that provides an efficient random hierarchical traversal and high culling efficiency with a small runtime decompression overhead. To further reduce the storage requirement, the in-core representation is compressed to our out-of-core representation, o-HCCMesh, by using a simple dictionary-based compression method. At runtime, o-HCCMeshes are fetched from an external drive and decompressed to the i-HCCMeshes stored in main memory. The i-HCCMesh and o-HCCMesh show 3.6:1 and 10.4:1 compression ratios on average, compared to a naively compressed (e.g., quantized) mesh and BVH representation. We test the HCCMesh representations with ray tracing, collision detection, photon mapping, and non-photorealistic rendering. Because of the reduced data access time, a smaller working set size, and a low runtime decompression overhead, we can handle models ten times larger in commodity hardware without the expensive disk I/O thrashing. When we avoid the disk I/O thrashing using our representation, we can improve the runtime performances by up to two orders of magnitude over using a naively compressed representation.*

1. Introduction

There has been extensive research on designing mesh representations optimized for different applications. For simple rendering, a mesh is often represented by a list of vertices and a list of three vertex indices defining each triangle, i.e. indexed triangle format. For applications requiring mesh traversals (e.g., iso-surface extractions), connectivity information is stored in a mesh representation or computed on the fly during streaming processing [ILS05].

Ray tracing and collision detection are widely used for providing high-quality visualizations and user interactions. In these algorithms, we need to detect intersecting primitives between two input objects (e.g., a ray and a 3D object in ray tracing and two 3D objects in collision detection). In order to efficiently detect these intersecting primitives, hierarchical traversal and culling by using bounding volume hierarchies (BVHs) are commonly used [TKH*05, YGKM08]. BVHs are constructed from meshes and leaf nodes of BVHs contain one or more triangles of the mesh. An intermediate node of a BVH has a bounding volume (BV) that encloses all the triangles contained in the sub-tree rooted at the node. Axis-aligned bounding boxes (AABBs) are commonly used as BVs, because of their simplicity and compactness [TKH*05, YGKM08].

Due to advances of model acquisition and computer-aided design techniques, massive models are easily generated these days. Such massive models can consist of hundreds of millions of triangles and thus use several gigabytes of memory. In addition, BVHs constructed from these massive models can use additional gigabytes of memory space. 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 [HPG07].

Only a few techniques have been proposed to design compact mesh and BVH representations in order to reduce the data access time and memory requirements during the hierarchical traversal [CSE06, LYTM08]. None of them supports various tree structures of BVHs, while providing efficient hierarchical culling and a low runtime access overhead. Furthermore, these prior techniques do not provide enough compression ratios to handle large-scale models consisting of hundreds of millions of triangles on commodity hardware.

Main results: In this paper, we propose a novel hierarchical-

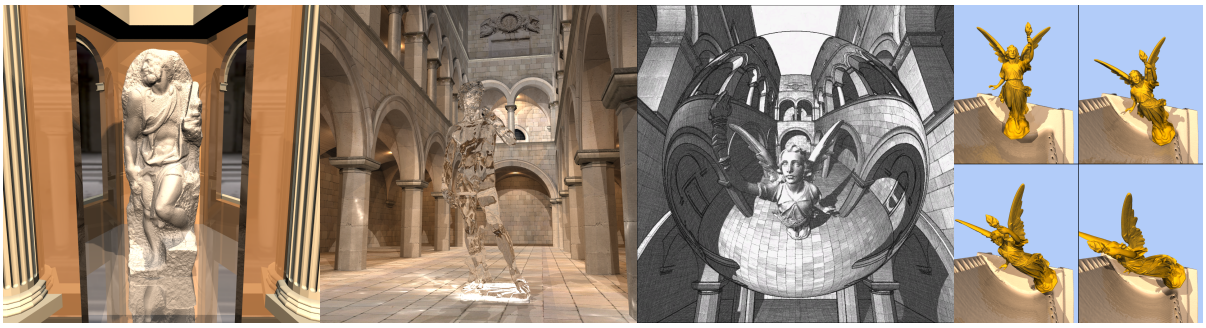

Figure 1: ***Applications:*** *These figures show images of applications using our HCCMesh representations. From left, we show a Whitted-style ray tracing of the St. Matthew, photon mapping on a transparent David model in the Sponza scene, a line-art style rendering of the Lucy model reflected on a sphere, and collision detection between the Lucy and a CAD turbine model*

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), and then further compress it into our out-of-core representation, *o-HCCMesh* (Sec. 5). Given a general out-of-core data access framework (Sec. 3), 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. 1 and Sec. 6.1) 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. 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.

2. Related Work

In this section we review prior work, mainly on compression methods of meshes and BVHs, to handle massive models.

2.1. Efficient Handling of Massive Models

At a high level, techniques that enable efficient handling of massive models are classified as out-of-core techniques minimizing the number of expensive I/O operations [SCC*02, CRMS03], multi-resolution techniques reducing the amount of necessary data [LRC*02], or data compression methods reducing the storage requirement [AG04, GGK02].

Mesh compression:

Mesh compression techniques have been well studied and excellent surveys are available [AG04, GGK02]. Most previous mesh compression schemes are designed to achieve a maximum compression ratio as they aim for archival use or for transmitting massive models. However, these techniques do not support efficient random hierarchical traversal and culling for the compressed meshes.

2.2. Tree and BVH Compression

Tree compression techniques have been studied in many different fields [KM90]. These techniques include linearizing the structure [Jac89] and transforming the tree into a predefined tree [Zer85]. Relatively little research has been done on compressing BVHs. A BVH has two types of information: the BV information and its tree structure.

Encoding BVs: In order to compress BVs, fixed-rate quantization methods are frequently used [CSE06, Ter03]. Also, hierarchical encoding schemes were developed to achieve a higher compression ratio [RL00, HMHB06, Kar07]. We propose a novel BV encoding scheme that tightly integrates the mesh and the BVH, to achieve a higher compression ratio while supporting efficient hierarchical traversal and culling.

Encoding tree structures: Many techniques assume a particular tree structure (e.g., complete tree) in order to remove most of cost related to encoding the tree structure [CSE06, LYTM08, Hav97]. Lauterbach et al. [LYTM08] introduced the 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 [YGKM08, TKH*05]. Compact sub-trees [Hav97] reduce the space for tree structures given an assumption of the complete sub-trees. Lefebvre and Hoppe [LH07] employed local offsets to encode the location of child nodes given the pre-ordered layout of the tree. Jacobson [Jac89] 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. Our proposed method also supports various kinds of tree structures and shows a

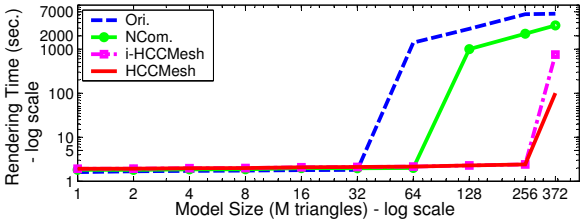

Figure 2: ***Ray Tracing Time vs. Model Complexity:*** *This graph shows the rendering time with various model complexities of the St. Matthew model shown in Fig. 1. We measure the performance of ray tracing with our HCCMesh, the original (**Ori.**), and naively compressed (**NCom.**) representations.*

compression ratio near the optimum in practice, while providing much faster runtime tree traversal than the succinct tree.

2.3. Random Access and Compression

Random access has been directly supported by a few compression techniques on the compressed single-resolution meshes [YL07, CKLL09] and multi-resolution meshes [KCL06]. However, it is unclear how to apply these techniques to our problem. Several octree based compression techniques are proposed [MCT08, CRMS03] but octrees are known to show low culling ratios than BVHs. Recently, Kim et al. [KMKY10] proposed a random-accessible compressed BVH representation. This method improved the performance of an out-of-core ray tracer by using compact out-of-core BVH representation. Our method compresses even an in-core representation and further improves the performance of various applications.

3. 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. We do not assume 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 [TKH*05, YGKM08]. We further assume that each leaf node of a BVH contains a single triangle of a mesh. It is straightforward to extend our method to rooted binary trees and k-ary trees. In Sec. 7 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.

3.1. 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 [SCC*02]. 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. 3), using a simple clustering method. For simplicity, 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). 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.

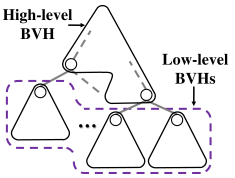

Figure 3: *Decomposition of a BVH to high-level and low-level BVHs*

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 [SCC*02].

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.

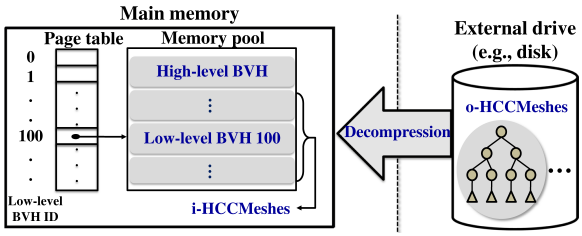

Figure 4: ***Out-of-core Runtime Access Framework:*** *Given the framework, our main contributions (shown with blue colors) are compact in-core and out-of-core HCCMesh representations and compression methods that allow fast runtime decompression performance.*

3.2. 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 shows the overall structure of the out-of-core access framework with our main contributions.

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. i-HCCMesh Representation

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

4.1. 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

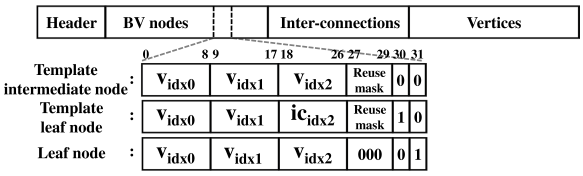

Figure 5: ***i-HCCMesh of a Low-Level BVH:*** *The i-HCCMesh consists of a header, a BV array, an inter-connection array, and a vertex array. There are three different BV node types in the i-HCCMesh and two bits are used to encode the type of a node.*

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. 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.

4.2. Encoding Bounding Volumes

The BV of a node is constructed to tightly enclose all the triangles contained in the sub-tree of the node [YGKM08, TKH*05]. 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. 6-(a)). This observation [LYTM08] 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. 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

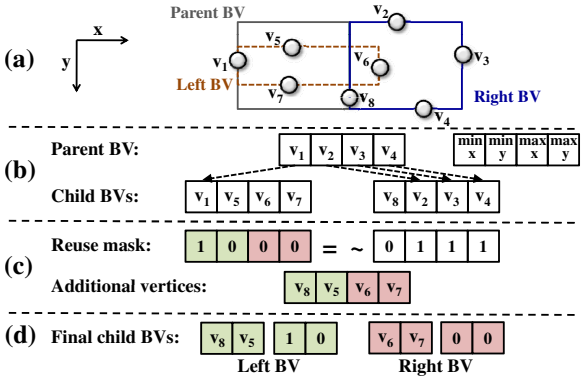

Figure 6: ***2D Example of In-core BV Encoding:*** *(a) The figure shows AABBs of a node, its two child AABBs, and vertices defining extents of the BVs. (b) Vertices defining BVs are shown. (c) Reuse mask and vertices that are additionally specified are shown. (d) The final BV encoding for two child nodes.*

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 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. 6. It is worth mentioning that an inheritance property and a reuse mask similar to ours have been introduced in an earlier work [Kar07].

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. 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

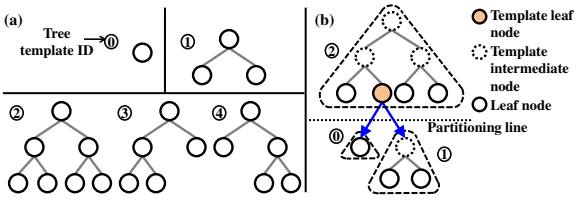

Figure 7: ***Tree Templates:*** *The left figure shows possible tree templates of a height of 1, 2, and 3, with its ID. The right figure shows a tree partitioning example with sub-trees that have up to a height of 3 and the computed tree template IDs.*

4 bytes; the unused two bits are used to encode three different types of a node, which will be explained later in Sec. 4.3.

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.

4.3. 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. 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. 7-(b)). The number of k-height tree templates, $T(k)$, is computed as the following (see the supplementary report for the proof 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. 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. 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. 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. 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.

4.4. Encoding Meshes

We quantize vertex geometry and normals of the mesh using simple quantization methods [RL00] 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.

5. 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 compressing more.

5.1. 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 [ILS05], since it runs quite fast while achieving a high compression ratio. We modify the streaming mesh compression method that employed a statistical compression method [Sal07] to use our simple dictionary-based compression method, which will be explained later in Sec. 5.2. 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 [AG04] 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 [ILS05]. 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.

5.2. 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 the 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 [Sal07], 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 $B(\cdot)$.

6. 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 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) [Wal04]. 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 to be **Ori.** and **NCom.** respectively. We use cache-oblivious layouts [YGKM08] 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 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.

Model	T (M)	Type	OS	NS	i-HCCMesh BV	Tr	M	CS	CR	CR'	o-HCCMesh 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 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.*

6.1. 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 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.

6.1.1. Whitted-Style Ray Tracing

We implement a Whitted-style BVH-based ray tracer [LYTM06] that can use HCCMeshes, **Ori.**, and **NCom**. Moreover, in order to compare the performance of the ray tracer on various model complexities (Fig. 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. 1.

When **Ori.** and **NCom.** fit into the available main memory, the performance of 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. 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.**.

6.1.2. 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 [KYYL08]. 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. 6.1.1) using our HCCMesh representations. We use 25 point lights in the Sponza scene with the Lucy model (Fig. 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.

6.1.3. Multi-Resolution Ray Tracing

Multi-resolution techniques are widely used to improve the performance of many rendering algorithms [LRC*02]. 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 [YLM06]. 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 [YLM06] 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 interconnection array of each low-level BVH.

The original multi-resolution representation takes 8.7 GB [YLM06] 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 [YLM06].

6.1.4. 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 [Jen01]. 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. 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.

6.1.5. Collision Detection

Collision detection is an essential technique for enabling user interaction. In practice, BVHs are widely used in practice [LM03]. We implement a rigid body simulation and drop the Lucy model on the top of a CAD turbine model (Fig.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 [YM06], 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.

7. 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 [GLM96] 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. 6.1.4.

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. 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 [LYTM08] and the LBVH [CSE06] respectively) and, more importantly, supports various tree structures that are constructed from different optimized hierarchy construction methods [YGKM08, TKH*05]. Since an optimized hierarchy can show 2 or more performance improvement than a naively constructed hierarchy [Wal04], our method can perform better and handle bigger data sets. The RACBVHs [KMKY10] 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 [YL07]. 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 [Jac89]. 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 more by 3 times to 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 Sec. 2.1, 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. 6.

8. 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 architecture. Second, we would like to further improve the decompression performance of our method by exploiting data-level parallelism of GPUs and Larrabee architectures.

Acknowledgments

We would like to thank Christian Lauterbach and anonymous reviewers for their constructive feedbacks. We also thank members of KAIST SGLab. for their helpful feedbacks. The St. Matthew, David, and Lucy models are courtesy of Stanford University. The Sponza model and CAD turbine models are courtesy of an anonymous donor and of Kitware respectively. This project was supported in part by MKE/MCST/IITA[2008-F-033-02,2008-F-030-02], MCST/KEIT [2006-S-045-1], MKE/IITA u-Learning, MKE digital mask control, MCST/KOCCA-CTR&DP-2009, KRF-2008-313-D00922, KMCC, and MSRA E-heritage.

References

[AG04] ALLIEZ P., GOTSMAN C.: Recent advances in compression of 3d meshes. *Advances in Multiresolution for Geometric Modelling* (2004), 3–26.

[CKLL09] CHOE S., KIM J., LEE H., LEE S.: Random accessible mesh compression using mesh chartification. *IEEE Trans. on Visualization and Computer Graphics 15*, 1 (2009), 160–173.

[CRMS03] CIGNONI P., ROCCHINI C., MONTANI C., SCOPIGNO R.: External memory management and simplification of huge meshes. *IEEE Trans. on Vis. and Com. Gra. 9*, 4 (2003), 525–537.

[CSE06] CLINE D., STEELE K., EGBERT P. K.: Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools 11*, 4 (2006), 61–71.

[GGK02] GOTSMAN C., GUMHOLD S., KOBBELT L.: Simplification and compression of 3d meshes. In *Tutorials on Multiresolution in Geometric Modelling*. Springer, 2002, pp. 319–361.

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph* (1996), 171–180.

[Hav97] HAVRAN V.: Cache sensitive representation for the bsp tree. In *Proc. of Compugraphics* (1997).

[HMHB06] HUBO E., MERTENS T., HABER T., BEKAERT P.: The quantized kd-tree: Efficient ray tracing of compressed point clouds. In *IEEE Symp. on Interactive Ray Tracing* (2006), pp. 105–113.

[HPG07] HENNESSY J. L., PATTERSON D. A., GOLDBERG D.: *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 2007.

[ILS05] ISENBURG M., LINDSTROM P., SNOEYINK J.: Streaming compression of triangle meshes. In *Symposium on Geometry Processing* (2005), pp. 111–118.

[Jac89] JACOBSON G.: Space-efficient static trees and graphs. In *Symp. on Found. of Comp. Science* (1989), pp. 549–554.

[Jen01] JENSEN H. W.: *Realisic Image Synthesis using Photon Mapping*. A K Peters, 2001.

[Kar07] KARRENBERG R.: *Memory Aware Realtime Ray Tracing: The Bounding Plane Hierarchy*. Bachelor thesis, Saarland University, Saarbrucken, Germany, 2007.

[KCL06] KIM J., CHOE S., LEE S.: Multiresolution random accessible mesh compression. *EG 25*, 3 (2006), 323–332.

[KM90] KATAJAINEN J., MAKINEN E.: Tree compression and optimization with applications. *International Journal of Foundations of Computer Science 1*, 4 (1990), 425–447.

[KMKY10] KIM T.-J., MOON B.-C., KIM D., YOON S.-E.: RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Trans. on Vis. and Com. Gra.* (2010). to appear.

[KYYL08] KIM Y., YU J., YU X., LEE S.: Line-art illustration of dynamic and specular surfaces. *ACM Trans. on Graphics (SIGGRAPH ASIA) 27*, 5 (Dec. 2008), 156:1–156:10.

[LH07] LEFEBVRE S., HOPPE H.: Compressed random-access trees for spatially coherent data. In *Eurographics Symposium on Rendering* (2007), pp. 339–349.

[LM03] LIN M., MANOCHA D.: Collision and proximity queries. *Handbook of Discrete and Computational Geometry* (2003).

[LRC*02] LUEBKE D., REDDY M., COHEN J., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

[LYTM06] LAUTERBACH C., YOON S., TUFT D., MANOCHA D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs. *IEEE Symp. on Interactive Ray Tracing* (2006), 39–45.

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: ReduceM: Interactive and memory efficient ray tracing of large models. *Computer Graphics Forum (EG Symp. on Rendering) 27*, 4 (2008), 1313–1321.

[MCT08] MELERO F., CANO P., TORRES J.: Bounding-planes octree: A new volume-based LOD scheme. *Computers and Graphics 32*, 4 (2008), 385–392.

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: A multiresolution point rendering system for large meshes. *SIGGRAPH* (2000), 343–352.

[Sal07] SALOMON D.: *Data Compression*. Springer, 2007.

[SCC*02] SILVA C., CHIANG Y.-J., CORREA W., EL-SANA J., LINDSTROM P.: Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Vis. Course Notes* (2002).

[Ter03] TERDIMAN P.: Opcode: Optimized collision detection, 2003.

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 19*, 1 (2005), 61–81.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[YGKM08] YOON S.-E., GOBBETTI E., KASIK D., MANOCHA D.: *Real-Time Massive Model Rendering*. Morgan & Claypool Publisher, 2008.

[YL07] YOON S.-E., LINDSTROM P.: Random-accessible compressed triangle meshes. *IEEE Trans. on Vis. and Computer Graphics (Proc. Visualization) 13*, 6 (2007), 1536–1543.

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-LODs: Interactive LOD-based Ray Tracing of Massive Models. *The Visual Computer (Pacific Gra.) 22*, 9–11 (2006), 772–784.

[YM06] YOON S.-E., MANOCHA D.: Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum (Eurographics) 25*, 3 (2006), 507–516.

[Zer85] ZERLING D.: Generating binary trees using rotations. *Journal of ACM 32*, 3 (1985), 694–701.