EUROGRAPHICS 2006 / E. Gröller and L. Szirmay-Kalos
(Guest Editors)

Volume 25 (2006), Number 3

Cache-Efficient Layouts of Bounding Volume Hierarchies

Sung-Eui Yoon[1] Dinesh Manocha[2]

[1]Lawrence Livermore National Laboratory [2]University of North Carolina at Chapel Hill

Abstract

We present a novel algorithm to compute cache-efficient layouts of bounding volume hierarchies (BVHs) of polygonal models. Our approach does not make any assumptions about the cache parameters or block sizes of the memory hierarchy. We introduce a new probabilistic model to predict the runtime access patterns of a BVH. Our layout computation algorithm utilizes parent-child and spatial localities between the accessed nodes to reduce both the number of cache misses and the size of the working set. Our algorithm also works well for spatial partitioning hierarchies including kd-trees. We use our algorithm to compute layouts of BVHs and spatial partitioning hierarchies of large models composed of millions of triangles. We compare our cache-efficient layouts with other layouts in the context of collision detection and ray tracing. In our benchmarks, our layouts consistently show better performance over other layouts and improve the performance of these applications by 26%–300% *without any modification of the underlying algorithms or runtime applications.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Hierarchy and Geometric Transformations

1. Introduction

Bounding volume hierarchies (BVHs) are widely used to accelerate the performance of geometric processing and interactive graphics applications. The applications include ray tracing, visibility culling, collision detection, and geometric computations on large datasets. Most of these algorithms precompute a BVH and traverse the hierarchy at runtime to perform intersection tests or culling.

The leaf nodes of a BVH correspond to the triangles of the original model. The intermediate nodes are the bounding volumes (BVs) such as spheres, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), and convex polytopes. The memory requirements of BVHs can be high for large datasets. For example, the storage cost of a hierarchy of OBBs (i.e., an OBB-tree) is approximately 64 bytes per node. As a result, BVHs of large datasets composed of tens of millions of triangles can require gigabytes of space.

Our goal is to compute cache-efficient layouts of BVHs to reduce the number of cache misses and improve the performance of BVH-based algorithms. As the gap between the processor speed and main memory speed widens, system designers increasingly use caches and memory hierarchies to reduce memory latency. The access times of different levels of a memory hierarchy can vary by orders of magnitude. As a result, the running time of an algorithm varies as a function of its cache access pattern.

Many mesh representations and algorithms have been proposed to improve the cache access patterns of geometric models for specific applications. These representations and algorithms include rendering sequences (e.g., triangle strips), processing sequences (e.g., streaming meshes), layouts computed using space filling curves, and minimum linear arrangement (MLA). However, these representations and algorithms may not improve the cache access patterns for different geometric applications.

Main Results: We present a novel algorithm to compute cache-efficient layouts of BVHs of large models. Our approach is cache-oblivious as it does not require any knowledge of cache parameters or block sizes of the memory hierarchy and is applicable to all kinds of BVHs and spatial partitioning hierarchies that can be represented as a tree. We represent a BVH as two separate linear sequences of BVs and triangles. Our problem is reduced to computing cache-efficient layouts of the BVs and the triangles. We introduce a new probabilistic model to predict the runtime access patterns of BVHs based on localities. Specifically, we utilize two types of localities during traversal of a BVH: parent-child and spatial localities between the accessed nodes. Our approach also uses the tree decomposition algorithm [GI99] and cache-oblivious mesh layout algorithms [YLPM05, YL06] to compute a layout that reduces the number of cache misses and the size of the working set.

We use our algorithm to compute layouts of OBB trees and kd-trees of large models composed of millions of triangles. Based on these layouts, we accelerate the performance of collision detection and ray tracing without any modifications to the underlying algorithms or runtime application. We also compare the performance of our layouts with other layouts including depth-first layout, breadth first layout, van Emde Boas layout, cache-oblivious mesh layout, and cache-aware layouts. We have observed 26%–2600% improvement in performance based on our cache-efficient layouts. Moreover, in some applications the performance of our cache-oblivious layouts is comparable to that of cache-aware layouts. Overall, our approach offers the following benefits:

1. **Generality:** Our algorithm is general and applicable to a wide range of BVHs and spatial partitioning hierarchies. It does not require any knowledge of cache parameters or of the block sizes of a memory hierarchy.
2. **Applicability:** Our algorithm does not require any modi-

Figure 1: ***Ray Tracing the Lucy model:*** We apply a standard kd-tree based ray tracing algorithm to the Lucy model consisting of 28 million triangles. A reflective plane is placed behind the Lucy model and the scene also has shadows. We compute a cache-efficient layout of the kd-tree of the Lucy model using our algorithm. Our layout improves the performance of ray tracing by up to two times over previous layouts, without any change to the underlying algorithm.

fication of BVH-based algorithms or the runtime application. layouts.

3. **Improved performance:** Our layouts reduce the number of cache misses during traversals of BVHs and spatial partitioning hierarchies. We are able to improve the performance of standard collision detection and ray tracing algorithms.

Organization: The rest of the paper is organized in the following manner. We give a brief survey of related work in Section 2 and an overview of memory hierarchies and BVHs in Section 3. Section 4 describes the localities that are used by our algorithm. We present a novel probabilistic model to predict the runtime access patterns of BVHs in Section 5 and describe our layout algorithm in Section 6. We highlights its performance in Section 7 and compare its performance with prior approaches in Section 8.

2. Related Work

In this section, we give a brief overview of related work on cache-efficient algorithms and layouts of BVHs and geometric models.

2.1. Cache-Efficient Algorithms

Cache-efficient algorithms have received considerable attention over last two decades in theoretical computer science and compiler literature. These algorithms include theoretical models of cache behavior [Vit01,SCD02] , and compiler optimizations based on tiling, strip-mining, and loop interchanging to minimize cache misses [CM95].

At a high level, cache-efficient algorithms can be classified as either cache-aware or cache-oblivious. Cache-aware algorithms utilize knowledge of cache parameters, such as cache block size [Vit01]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [FLPR99]. There is considerable literature on developing cache-efficient algorithms for specific problems and applications [ABF04, Vit01].

2.2. Layouts of BVHs

The impact of different layouts of tree structures has been widely studied. There is considerable work on cache-coherent layouts of tree-based representations including work on accelerating search queries. Given the cache parameters, Gil and Itai [GI99] cast cache-coherent layout computation as an optimization problem. They propose a dynamic programming algorithm to minimize the number of cache misses during traversals of search queries. However, the computed layout may not be storage efficient. Alstrup *et al.* [ABFC*03] propose a method to compute cache-oblivious layouts of search trees by recursively partitioning the trees.

There is relatively less work on cache-coherent layouts of BVHs. We refer the readers to a recent book on real-time collision detection [Eri04]. Opcode [Ter03] uses a blocking method that merges several bounding volumes nodes together to reduce the number of cache misses. The blocking is based on *van Emde Boas* layout of complete trees [vEB77]. However, it is not clear that van Emde Boas layouts can minimize the number of cache misses during traversal of general BVHs. Havran analyzes various layouts of BVHs in the context of ray tracing and improves the performance by using a compact layout representation of BVHs [Hav97]. Yoon et al. [YLPM05] propose a cache-oblivious mesh layout algorithm to compute layouts of geometric meshes and bounding volume hierarchies. We compare our approach with this algorithm in Section 8.2.

Layouts of geometric meshes: Many algorithms and representations have been proposed to compute coherent layouts for specialized applications. Rendering sequences (e.g., triangle strips) [Dee95,Hop99] are used to improve rendering throughput by optimizing the vertex cache hits in the GPU. Isenburg and Gumhold [IG03] propose processing sequences, including streaming meshes [IL04], as an extension of rendering sequences for large-data processing. In these cases, global mesh access is restricted to a fixed traversal order. Many algorithms use space filling curves [Sag94] to compute cache-friendly layouts of volumetric grids or height fields. These layouts are widely used to improve performance of image processing [VG91] and terrain or volume visualization [PF01, LP01]. However, it is unclear whether space filling curves would extend to compute layouts of unstructured models and their hierarchies. In graph theory, minimum linear arrangement (MLA) [DPS02] has been widely researched to minimize the sum of edge lengths of all the edges in a graph layout. However, there is no direct relationship between reducing the sum of edge lengths and minimizing the number of cache misses. Recently, Yoon and Lindstrom [YL06] show that the MLA metric is a cache-oblivious metric assuming that all the cache block sizes are employed at runtime. Moreover, they also showed that the sum of a log function of edge lengths is a practical cache-oblivious metric derived from assuming power-of-two block sizes.

3. Memory Hierarchies and BVH Layouts

In this section, we give an overview of memory hierarchies, BVHs, and their layouts. We also introduce some of the terminology used in the rest of the paper.

3.1. Memory Hierarchy and Caches

Most modern computer architectures use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. The memory hierarchies have two main characteristics [AV88]. First, higher levels in the hierarchy are larger

in size, farther away from the processor, and have slower access times. Second, data is moved in large cache blocks between different memory levels. The BVH layout is initially stored in the highest memory level, typically the disk. The portion of the layout accessed by the application is transferred in large blocks into the next lower level, such as the main memory. A transfer is performed whenever there is a cache miss between two adjacent levels of the memory hierarchy. The number of cache misses depends on the layout of the BVH in memory and the access patterns of the application. If the nodes of a BVH are packed in blocks in a cache-coherent manner, the number of cache misses can be reduced. For the sake of clarity, we will use the term of cache block to indicate blocks of different caches including L1/L2, main memory, and disk.

3.2. Bounding Volume Hierarchies

BVHs are widely used in various applications to accelerate the performance of intersection or culling tests. The leaf node of a BVH corresponds to the triangulated primitives and the intermediate nodes are the bounding volumes (BVs). Each BV conservatively encloses its geometry contained in the node. Some of the commonly used BVHs include sphere-trees [Hub93], OBB-trees [GLM96], and k-DOP-trees [KHM*98]. In the rest of this paper, we use collision or intersection queries as the driving application to explain the concepts behind computing cache-efficient layouts of BVHs. These algorithms typically take two inputs: two 3D objects or one 3D object and a ray. The runtime algorithm traverses the BVHs of each object using a depth-first or a breadth-first order. The depth-first order is typically used when we need to check for ray-object intersection or to check whether two objects overlap. The breadth-first traversal order is preferred when the runtime algorithm can be interrupted and may return approximate results, e.g. time-critical computations or constant frame-rate rendering of large models.

Extensive work has been done on evaluating the performance of different BVHs for ray-tracing and proximity queries. These include the cost equations for ray-tracing [WHG84] and collision detection [GLM96, KHM*98]. These cost equations take into account the tightness of fit for a BV and the relative cost of computing intersections or overlaps with those BVs based on the traversal pattern. However, these formulations do not take into account the cost of memory accesses or of cache misses incurred while traversing the BVHs. If the underlying model and its BVH cannot fit into the main memory, the cost of memory accesses and cache misses can become a significant factor.

3.3. Layout of BVH

We use the following notation to represent the BVs of a BVH. We define n_i^1 as the ith BV node at the leaf level of the hierarchy and n_i^k as a BV node at the kth level of the hierarchy. We also define $Left(n_i^k)$ and $Right(n_i^k)$ to be the left and right child nodes of the n_i^k. A parent node and a grandparent node of the n_i^k are denoted by using $Parent(n_i^k)$ and $Grand(n_i^k)$.

Formally speaking, a BVH is a directed acyclic graph, $G(N,E)$, where N is a set of BV nodes, n_i^k, and E is a set of directed edges from a node, n_i^k, to each child node, $Left(n_i^k)$ and $Right(n_i^k)$, in the BVH. A layout of a BVH is composed of two parts: a BV layout and a triangle layout. A BV layout of a BVH, $G(N,A)$, is a one-to-one mapping of BVs to positions in the layout, $\varphi : N \rightarrow \{1,\ldots,|N|\}$. Our goal is to compute a mapping, φ, that minimizes the number of cache misses and the size of the working set during the traversal of the BVH at runtime. Similarly, we also compute a triangle layout to minimize the cache misses and the working set size during BVH traversals.

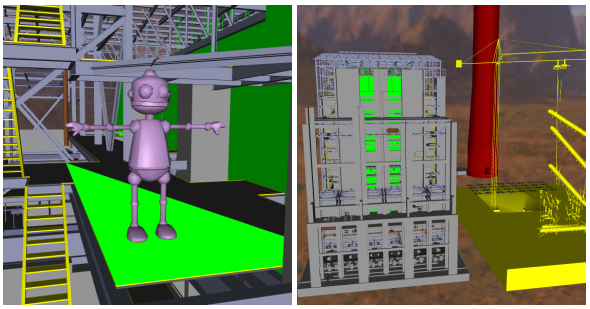

Figure 2: ***Hugo and 1M Power Plant Models:*** The Hugo robot model is placed in the top left of the power plant model, whose overall shape is shown on the right. We are able to achieve 35%–2600% performance improvement in collision detection by using our cache-efficient layouts of the OBB-tree over other tested layouts.

4. Localities in BVH Traversal

In this section, we define two localities that are used to compute a cache-efficient layout of a BVH. We also give a brief overview of prior work on packing trees and cache-oblivious mesh layout algorithms, which are used by our novel layout algorithm.

4.1. Access Patterns during BVH Traversal

Collision queries traverse BVHs as long as each query between two BVs reports a collision between them. We decompose the access pattern during a traversal into a set of search queries. We define a *search query*, $S(n_i^k)$, to be the traversal from the root node of the BVH to the node, n_i^k, which can be either a leaf or an intermediate node. Let us assume that the traversal of a collision query starts from the root node and ends at nodes, $n_{i(1)}^{k(1)},\ldots,n_{i(m)}^{k(m)} (= BV_1,\ldots,BV_m)$. In this case, the nodes, $(BV_1,\ldots,BV_m)$, define a front of the BVH for this traversal. We represent this traversal as the union of traversals of m different search queries, $S(BV_j)$. An example of an access pattern between two colliding objects is shown in Fig. 3. In frame i, the collision query ends at n_1^3 and n_5^3 starting from the root node, n_1^4, of the BVH of object 1. We can represent the access patterns of this collision query with two search queries ending at n_1^3 and n_5^3.

There are two different localities, parent-child locality and spatial locality, which arise during the traversal.

1. **Parent-child locality:** Once a node of a hierarchy is accessed by a search query, it is likely that its child nodes will be accessed soon. For example, in frame i of Fig. 3, if the root node of the BVH is accessed, its two child nodes, n_1^3 and n_5^3, are likely to be accessed soon. Moreover, after n_1^3 is accessed during frame i, its child nodes are likely to be accessed in the next frame.
2. **Spatial locality:** Whenever a node is accessed by a search query, other nodes in close proximity are also highly likely to be accessed by other search queries. For example, collisions or contacts between two objects occur in small localized regions of a mesh. Therefore, if a node of a BVH is

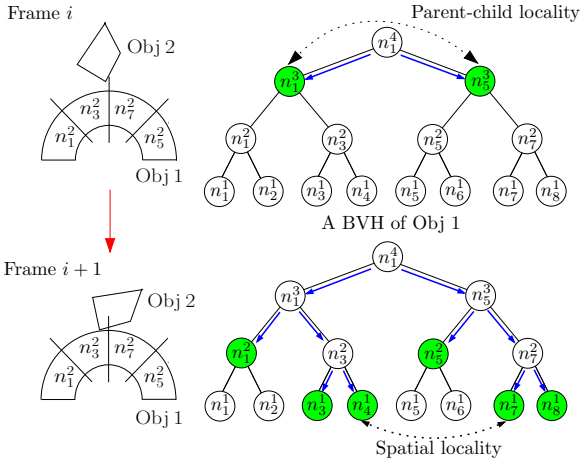

Figure 3: ***Two localities within BVHs:*** We show two successive frames from a dynamic simulation and the change in access patterns (shown with blue arrows) of a BVH. In this simulation, object 2 drops on object 1, as shown on the left. The access pattern of the BVH of object 1 during each frame is shown on the right. The BVs from the 2nd level in the BVH are shown within object 1 on the left. We also illustrate the front traversed within each BVH during each frame in green. The top BVH shows the parent-child locality, when the root node, n_1^4, of the BVH of object 1 collides with the BVs of objects 2. During frame $i+1$, object 2 is colliding with object 1. In this configuration, the BVs n_3^2 and n_7^2 (and their sub-nodes) are accessed due to their close spatial locality.

accessed, other nearby nodes are either colliding or are in close proximity and may be accessed soon. In frame $i+1$ of Fig. 3, if one of two nodes, n_4^1 and n_7^1, is accessed, the other node is also likely to be accessed during that frame or subsequent frames.

We consider each of these two localities and use them to compute the layout of a BVH. In the remainder of this section, we briefly summarize several known results related to these localities.

4.2. Parent-Child Locality

We use several results presented by Gil and Itai [GI99] to compute a cache-coherent layout of a BVH. Gil and Itai address the problem of computing a good layout for search queries on a tree. They define two different measures for the cache-coherence of a layout of a tree. The two measure are:

1. **The number of cache misses (or page faults):** $PF^1(BV_i)$ is defined as the number of cache misses, given a cache that can hold only single cache block during the traversal of a search query ending at BV_i.
2. **The size of working set:** The working set during the traversal of the search query is a set of the different cache blocks that are accessed. $WS(BV_i)$ is defined as the size of the working set.

Intuitively speaking, $PF^1(BV_i)$ measures the number of times that accessing BVs crosses boundaries of cache blocks of the layout during the traversal. Additionally, [GI99] introduced a virtual probability function, $Pr(BV_i)$, that can measure how many times BV_i is accessed during any search query on the tree. The expected size of working set, WS, of the layout can be formulated as:

$$WS = \sum_{BV_i \in \text{BVH}} Pr(BV_i) WS(BV_i),$$

for all nodes BV_i in the hierarchy. Similarly, we can define the expected number of cache misses, PF^1, of a layout by multiplying $Pr(BV_i)$ by $PF^1(BV_i)$ for all nodes BV_i in the tree. If a tree layout is optimal given the PF^1 or WS measure, the tree layout is defined as PF^1-optimal or WS-optimal, respectively.

Lemma 1 **(Convexity):** *If a layout of a tree is PF^1-optimal or WS-optimal, the layout is convex [GI99].*

The layout of a tree is convex if all the intermediate BVs between BV_0 and BV_k are stored in the same block when a node BV_0 and its descendant BV_k are stored in the same cache block.

Lemma 2 **(Equivalence):** *A layout of a tree is PF^1-optimal if and only if it is WS-optimal [GI99].*

Lemma 3 **(NP-Completeness):** *Computing a layout of a tree that is a WS-optimal with a minimum storage is NP-Complete [GI99].*

We use these properties and lemmas to design our layout algorithm that considers parent-child locality during the traversal of search queries.

4.3. Spatial Locality

Recently, Yoon et al. proposed approaches to compute cache-coherent layouts of meshes and graphs [YLPM05, YL06]. They construct a graph to represent cache-coherent access patterns on an input mesh . Each vertex of the graph represents a data element (e.g. a vertex of the mesh) that an application accesses. Depending on the spatial locality between two vertices, an edge connecting two vertices is created with a weight that is proportional to the spatial locality. Also, to compute cache-coherent layouts of meshes, Yoon et al. perform multi-level optimization given the cache-oblivious metric to measure the expected number of cache misses for each edge.

We summarize two major results that are relevant to our work. First, Yoon et al. show that the expected number of cache misses during accessing an edge consisting of two vertices has high correlation with a log function of the edge length, the index gap of the two vertices in the layout. Therefore, the multi-level layout construction seeks a layout that has a lower sum of the log functions of edge lengths to minimize the expected number of cache misses. Second, any layout computed by the multi-level layout construction method that recursively divides each chunk of the mesh into k subset as proposed in [YLPM05] is implicitly sub-optimized to construct cache-oblivious layouts of meshes. These two results are used as part of our layout algorithm with our probabilistic model.

5. Probabilistic Model

In this section we present our probabilistic model, which is used to predict the runtime access patterns on BVHs. We derive our formulation based on the geometric relationship between the nodes of BVHs.

We assign to a BV, n_i^k a probability, $Pr(n_i^k)$, that the node would be accessed during the traversal as part of a search query. Suppose the parent node, $Parent(n_i^k)$, of a node, n_i^k, of an object collides with a BV node, BV_{Obj2}, of another object. In this case, the two children of $Parent(n_i^k)$ are fetched and tested to further localize the colliding region. Therefore, $Pr(n_i^k)$ can be computed by multiplying two factors: 1) the probability that $Parent(n_i^k)$ is accessed, and 2) the probability

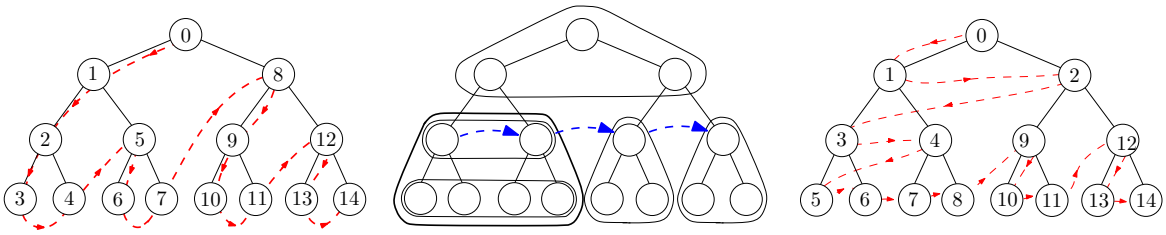

Figure 4: ***Layout computation of a BVH:*** A depth-first layout of a BVH is shown in the leftmost figure and a cache-oblivious layout of the same tree is shown in the rightmost figure. The number within each BV node in the leftmost and the rightmost figures is an index of the ordering of BVs in the layout. The middle figure shows the output of the clustering step. The topmost cluster is the root cluster and the rest are child clusters. Directed edges (shown in blue) indicate ordering between clusters. Also, the middle figure indicates that leftmost cluster is merged with its neighboring cluster.

that $Parent(n_i^k)$ collides with BV_{Obj2}. If there is a collision between the two nodes, each node is further refined with its two child nodes. Thus, the second probability can be computed by assuming that there was also a collision between $Grand(n_i^k)$ and BV_{Obj2}. The probability that n_i^k is accessed during the traversal can be recursively formulated as following:

$$Pr(n_i^k) = Pr(Parent(n_i^k))Pr(n_i^k, X_p = 1 | X_g = 1) \qquad (1)$$

where X_p and X_g are two binary random variables indicating whether there are collisions between the $Parent(n_i^k)$ and BV_{Obj2} and between $Grand(n_i^k)$ and BV_{Obj2}, respectively.

5.1. Probability Computation

Our goal is to efficiently compute $Pr(n_i^k, X_p = 1 | X_g = 1)$ given the recursive probability formulation presented in Eq. (1). Since we compute probabilities for nodes of the BVH as a preprocess, we do not know anything about the size or BV type of BV_{Obj2}, a BV node of another object. Instead of assuming any particular BV for BV_{Obj2}, we enumerate all possible configurations of BVs for BV_{Obj2} and compute the probability. Let $S_g(n_i^k)$ be the set that represent all possible configurations of BV_{Obj2} that collide with $Grand(n_i^k)$. We can similarly define $S_p(n_i^k)$. For example, if BV_{Obj2} is a sphere, $S_p(n_i^k)$ can be constructed by *Minkowski sum*: $S_p(n_i^k, r) = Parent(n_i^k) \bigoplus Sphere(r)$ where $Sphere(r)$ is a sphere with a radius, $r \in [0, \infty)$ and $\bigoplus$ is the Minkowski sum operator. In the more general case, BV_{Obj2} would correspond to a box or a convex shape and could have arbitrary orientation. As a result, both $S_g(n_i^k)$ and $S_p(n_i^k)$ can be represented as high dimensional *configuration-space*. Given the formulation of $S_p(n_i^k)$ and $S_g(n_i^k)$, $Pr(n_i^k, X_p = 1 | X_g = 1)$ can be defined as:

$$\begin{aligned} Pr(n_i^k, X_p = 1 | X_g = 1) &= \frac{Pr(X_p = 1 \cap X_g = 1)}{Pr(X_g)} \\ &= \frac{Vol(S_p(n_i^k) \cap S_g(n_i^k))}{Vol(S_g(n_i^k))}, \end{aligned} \qquad (2)$$

where $Vol(A)$ represents the volume of A. Intuitively speaking, the probability is the ratio of the volume of the intersected space between $S_p(n_i^k)$ and $S_g(n_i^k)$ to the volume of $S_g(n_i^k)$. We refer to the intersected volume ratio between $S_p(n_i^k)$ and $S_g(n_i^k)$, as $V_{intersected}(n_i^k)$.

It is, however, complex and expensive to construct the Minkowski sum or the configuration space in general [VM04]. The combinatorial complexity is high and the resulting algorithms are susceptible to degeneracies and robustness problems. As a result, exact computation of $Pr(n_i^k)$ is non-trivial.

5.2. Approximate Probability Computation

We propose a simple method to approximate the probability function described in Eq. (2). We observe that the intersected volume ratio computed when BV_{Obj2} is considered to be a point–therefore, BV_{Obj2} has zero extent–is a good approximation of the probability, which is the intersected volume ratio, $V_{intersected}(n_i^k)$, between $S_p(n_i^k)$ and $S_g(n_i^k)$. In other words, we use an intersected volume ratio, $V_{intersected}(n_i^k, 0)$, between $Parent(n_i^k)(= S_p(n_i^k, 0))$ and $Grand(n_i^k)(= S_g(n_i^k, 0))$ for the probability defined in Eq. 2. This approximation is based on the following observations:

- **Relative importance of probabilities during layout computation:** Suppose that our layout algorithm considers two nodes, n_1 and n_2, to decide which node should be ordered first. Our layout algorithm will choose a node that has a higher probability.
- **Importance of $S_p(n_i^k, 0)$ and $S_g(n_i^k, 0)$ for probability computation:** Suppose that an intersected volume ratio, $V_{intersected}(n_1, 0)$, between $Parent(n_1)$ and $Grand(n_1)$ is bigger than its counterpart, $V_{intersected}(n_2, 0)$, of $Parent(n_2)$ and $Grand(n_2)$, when r is zero. It is then likely that the intersected volume ratio, $V_{intersected}(n_1, r)$, between $S_p(n_1, r)$ and $S_g(n_1, r)$ is also bigger than its counterpart, $V_{intersected}(n_2, r)$, of n_2, when r is non-zero. Therefore, we can approximate the relative importance of the intersected volume ratio, $V_{intersected}(n_i^k)$, between $S_p(n_i^k)$ and $S_g(n_i^k)$ as the relative importance of the intersected volume ratio, $V_{intersected}(n_i^k, 0)$, between BVs of $Parent(n_i^k)$ and $Grand(n_i^k)$.

In order to quantitatively verify our approximation, we selected two nodes, n_1 and n_2, during layout computation of the dragon model and measured $V_{intersected}(n_1, r)$ and $V_{intersected}(n_2, r)$ as r geometrically increases from zero. We observed that less than 5% of relative importance between $V_{intersected}(n_1, 0)$ and $V_{intersected}(n_2 2, 0)$ is reversed as compared to $V_{intersected}(n_1, r)$ and $V_{intersected}(n_2, r)$, when we used a higher radius for BV_{Obj2} as a sphere.

Discretization: In order to approximate the volume ratio of the intersected area between $Parent(n_i^k)$ and $Grand(n_i^k)$ to $Grand(n_i^k)$, we overlay a uniform grid on the BV of

$Grand(n_i^k)$ and measure the number of cells of the grid that are contained in the BV of $Parent(n_i^k)$. We generate a sample point in each cell to perform this containment test. In practice, we found that using 64 samples to compute the probability was sufficient.

6. Layout Computation

In this section, we present a simple greedy algorithm to compute a cache-efficient layout of a BVH. We use the properties, lemmas, and the probability model described in the previous sections to compute cache-efficient layouts of BVHs.

6.1. Overall Algorithm

At the top level, our algorithm decomposes a BVH into clusters. If we knew the cache parameters and the block size, we could compute how many BV nodes fit into the cache block. Given this information, we could decompose the BVH into a set of clusters, such that the size of each cluster is equal to the size of the cache block. However, our algorithm does not assume any particular cache size and constructs a layout that works well with any cache parameter. In order to achieve this goal, we recursively compute the clusters. We first decompose the BVH into a set of clusters and recursively decompose each cluster. In this case, the cache block boundaries can lie anywhere within a layout that corresponds to the nodes of these clusters. Therefore, we need to compute a cache-efficient ordering of the clusters computed at each level of recursion.

Our algorithm has two different components that handle parent-child and spatial localities. In particular, the first part of our algorithm decomposes a BVH into a set of clusters that minimize the cache misses for parent-child locality. The clusters are classified as a root cluster and child clusters. The root cluster contains the root node of the BVH and child clusters are created for each node outside the root cluster whose parent node is in the root cluster (see the middle image in Fig. 4). The second part of the algorithm computes an ordering of the clusters and stores the root cluster at the beginning of the ordering. The ordering of child clusters is computed by considering their spatial locality. Then, we can merge two child clusters if it can further decrease the size of the working set. We recursively apply this two-fold procedure to compute an ordering of all the BVs in the BVH.

Cluster size: For each level of recursion, we decompose the BVH into a set of clusters that have approximately the same number of BV nodes. Suppose that a root cluster has B BV nodes. Then, the root cluster has $B+1$ child clusters and we decompose the BVH into $B+2$ clusters. Assuming that each cluster is reasonably balanced in terms of the number of BV nodes belonging to each cluster, $B \times (B+2)$ should be bigger than n, the number of nodes in the BVH, to contain all the nodes in the BVH. Therefore, B is set to be $\lceil \sqrt{n+1} - 1 \rceil$.

6.2. Cluster Decomposition

Before computing clusters from the BVH, we first compute and assign a probability, $Pr(n_i^k)$, to a BV, n_i^k, as described in the previous section. Then, we partition the BVH into $B+2$ clusters, where B is the number of nodes in the root cluster.

Our goal in this step is to store the BV nodes, which are accessed together due to the parent-child locality, into the same cluster in order to minimize the number of cache misses. According to our probability model shown in Eq. (1), the probability assigned to each node can also be considered the probability that the node is accessed, given that a root node of a cluster is already accessed. Therefore, we can achieve our goal by maximizing the sum of probabilities of BVs belonging to the root cluster. Moreover, maximizing this sum to the root cluster also minimizes the probability of accessing the nodes belonging to the child clusters. This formulation also minimizes the number of times that a search query accesses the data across the boundaries of cache blocks of the layout, quantified by PF^1 measure. According to Lemma 2, computing an optimal layout for the PF^1 metric is equivalent to computing an optimal layout that minimizes the expected size of working set, WS. Therefore, maximizing the sum of probabilities of BVs belonging to the root cluster minimizes the expected size of the working set during collision queries in the end.

However, computing a layout that minimizes the working set and the number of cache misses for all possible search queries with minimum space of a layout is NP-complete (as per Lemma 3). As a result, we employ a greedy algorithm to efficiently compute a cache-oblivious layout of the BVH. Our algorithm greedily traverses the BVH and merges nodes from the root node of the BVH into the root cluster by locally choosing a node that has the highest probability. Once the root cluster has B nodes, we stop merging the nodes into the root cluster. Then, each child node of the nodes inside the root cluster whose child nodes are outside the root cluster consists of a child cluster containing all the nodes of its sub-tree. The layout computed by our greedy approach also maintains the convexity of the layout as defined by Lemma 1.

6.3. Layouts of Clusters

Given the computed clusters at each level of the recursion, we compute a cache-oblivious ordering of the clusters by considering their spatial locality. During each recursive step of the algorithm, the number of BV nodes belonging to each cluster roughly reduces by a factor of $B+2$, based on our cluster computation algorithm. This causes considerable differences between the sizes of clusters created during the previous level of the recursion and the current level of the recursion. Therefore, it is important to compute a cache-coherent ordering of the clusters in order to further reduce the cache misses. This is because there is high likelihood that the size of a cache block may lie between the cluster size of the previous level and the current level of recursion.

We place the root cluster at the beginning of the ordering of clusters, since the traversal typically starts at the root node of the BVH. In order to compute an ordering of child clusters, We test two different layout methods based on two results summarized in Sec. 4.3.

Our first approach is to construct a cache-oblivious layout of child clusters by computing an undirected graph that represents access patterns between clusters. To construct the graph, we construct an edge between two clusters if they are in close proximity, that is, if their BVs overlap. Then, we compute a probability that a BV of a cluster has collided given that a BV of another cluster has collided based on the probability formulation described in Eq. 2. Once the graph is computed, a cache-oblivious layout of child clusters is computed using the cache-oblivious mesh layout algorithm [YLPM05]. We found that this method effectively reduces the size of working set. However, we also found that the runtime performance of the application can increase, especially, when the performance is dominated by disk access time. This is mainly because the layout of child clusters is constructed solely based on spatial locality between the clusters, although a runtime application is likely to access child clusters from left to right clusters since the application typically uses depth-first or breadth-first traversal.

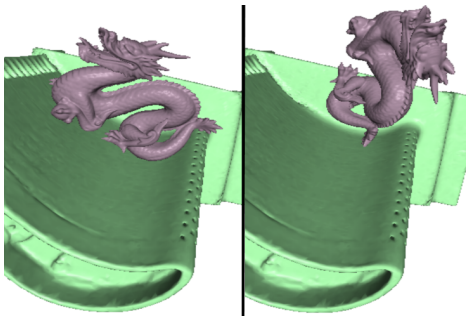

Figure 5: ***Dynamic Simulation between Dragon and Turbine Models:*** This image sequence shows discrete positions from our dynamic simulation between dragon and CAD turbine models. We are able to achieve 38%–215% performance improvement in collision detection by using our cache-efficient layouts of the OBB-tree over other tested layouts.

To address this issue, we layout child clusters from leftmost to rightmost following their position in the BVH as our second approach. Note that this order can be viewed to be constructed through a multi-level layout construction method since root nodes of child clusters can be considered to be recursively divided from their common root node of the child clusters. Therefore, this simple layout of child clusters can be also considered as a sub-optimized cache-oblivious layout based on the second results summarized in Sec. 4.3, We are also able to observe that the working set size during runtime traversal based on the layout computed by our second approach is within 5% of that of the first layout approach. Moreover, we also found that the runtime performance of applications is further improved with the second layout method since the layout order is more coherent to breadth-first and depth-first runtime traversal.

Merging clusters: If we recursively apply the cluster decomposition method and the cluster layout method proposed above, all the nodes contained in one cluster are stored before or after nodes of another cluster. However, if there is more overlap between root nodes of two clusters than between each root node and its child nodes, two root nodes of the child cluster are likely to be accessed sequentially. By doing this, we can further reduce the size of working set. We formalize this observation like this: if the probability that two root nodes are accessed due to spatial locality is bigger than the probability that their children nodes are accessed due to the parent-child locality, we merge two clusters into one. Once clusters are merged into one bigger cluster, the two root nodes of the clusters are stored consecutively in the final computed layout within our layout algorithm.

6.4. Triangle Layout

Once a set of BV pairs is computed during the runtime traversal of the BVHs of two objects, exact query computation based on the triangles of leaf nodes is performed. We extract a triangle layout from the BV layout of the BVH for efficient layout computation. If we encounter leaf nodes of the BVH during layout computation, we sequentially order the triangles stored in the BVs into the triangle layout since we perform the overlap tests at runtime in a sequential manner based on the stored order of the triangles within a leaf node.

7. Implementation and Performance

In this section we describe our implementation and highlight the performance of cache-oblivious layouts on different BVHs and spatial partitioning hierarchies. These include the kd-tree used by a ray tracing algorithm and OBB-tree used to perform collision queries in a dynamic simulation.

Model	Triangles (M)	Size of BVH (MB)	Mean and std of depth of leaves	Comp. time (min)
Hugo	0.02	2	16, 1.7	0.03
Bunny	0.07	8	17, 0.8	0.26
Dragon	0.8	108	21, 1.6	3
1M power plant	1.1	139	23, 2.9	6
Turbine	1.7	220	22, 0.7	8
Lucy	28	4,811	37, 3.4	34

Table 1: ***Benchmark Models:*** *Model complexity, sizes of BVHs, mean and standard deviation(std) of depth of leaf nodes, and computation time to compute cache-oblivious layouts are shown.*

7.1. Implementation

We have implemented our cache-oblivious layout computation algorithm as well as the two applications on a 2.4GHz Pentium-IV PC with 1GB of RAM. Our cache-oblivious layout algorithm can handle very large datasets in an out-of-core manner. Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files.

7.2. Benchmark Models

Our algorithm has been applied to different polygonal models. These include the Lucy model composed of more than 28 million polygons (Fig. 1), 1M version of power plant model, a Hugo model consisting of 16K polygon (Fig. 2), the CAD turbine model consisting of a single object with 1.7 million triangles (Fig. 5), the dragon model consisting of 800K polygons, and the Stanford bunny model consisting of 67K polygons (Fig. 7). The details of these models are shown in Table 1.

7.3. Performances

We applied our out-of-core layout computation algorithm to compute cache-oblivious layouts of BVHs of the models. Table. 1 presents the layout time for each model. An unoptimized implementation of our out-of-core algorithm can process up to 14K triangles per second.

7.3.1. Collision Detection

We have implemented an impulse based rigid body simulation [MC95] for dynamic simulation. We use OBB-trees [GLM96] to perform collision queries. To compute OBB-trees in an out-of-core manner, we decompose the input mesh into chunks of geometry [YSGM04]. We compute cache-efficient layouts of the OBB-trees of different models and use these layouts with the same underlying algorithm, i.e. RAPID [GLM96], to perform collision queries. In our current implementation, each OBB node takes 64 bytes.

We compared the performance of our cache-oblivious layout of BVHs (COLBVH) with different layouts including depth-first layout(DFL) of the BVH, breadth-first layout(BFL), van Emde Boas layout (VEB) [vEB77], cache-oblivious mesh layout (COML) [YLPM05], and a cache-aware layout obtained by explicitly setting cache size into our cache-oblivious layout algorithm (CALBVH). The OBBs are precomputed and only the ordering of the hierarchy is modified. The COML, as explained in Sec. 4.3, is computed by constructing an undirected graph. This is accomplished by generating edges between parent and child nodes and between nearby nodes on the same level of the BVH. We use *OpenCCL*

library [YMLP05] to compute cache-oblivious layouts of the graph representing access patterns on the BVH. The VEB layout is computed recursively. The tree is partitioned with a horizontal line so that the maximum height of the tree is divided into half. The resulting sub-trees are linearly stored by first placing the root sub-tree followed by other sub-trees from leftmost to rightmost. This process is applied recursively until it reaches a single node of the tree.

We have tested the performance of the OBB-tree collision detection algorithm with our layouts in three different benchmarks:

1. **Bunny and Dragon:** A bunny moves towards a dragon (Fig. 7).
2. **Dragon and Turbine:** A dragon drops onto the CAD turbine model(Fig. 5).
3. **Power plant and Hugo:** A Hugo robot model is placed in the top left side of the power plant model (Fig. 2. This is our benchmark 3-a. Also, the robot model is placed in the middle of the power plant model, specifically in the furnace room. This is our benchmark 3-b. This particular benchmark has much larger overlaps between the BVs since the robot model is placed inside the power plant model.

Our first and second benchmarks are performed during a rigid body simulation.

We collected timing data after making sure that there is no loaded data in the main memory. Moreover, we also made sure there is no file fragmentations since the fragmentations can slow down the performance of I/O accesses. Dynamic simulations of the first and second benchmarks are shown in the accompanying video. In the first and second benchmarks, we are able to achieve a 26%–215% improvement in the performance of collision queries by using COLBVHs over other layouts on our benchmarks. Also, the performance of cache-oblivious layout is comparable to that of cache-aware layout. This improvement is achieved by reducing the working set during collision queries and fewer cache misses. Moreover, in our benchmark 3-a and 3-b, we are able to achieve up to 26 times improvement over other layouts. Since the power plant model has very irregular distribution of geometry, our layout method considering geometric relationship between BVs is able to achieve higher performance improvement over other layouts in this particular model. Also, our layout consistently shows better performance over other layouts. In Fig. 6, we report the average collision query times and working set size in our benchmarks.

In benchmarks 1, 2, and 3a, VEB layout has slightly worse performance over our layouts. This is mainly because the OBB-trees are almost balanced trees and the ordering of child clusters from left to right during VEB layout computation maintains reasonably good cache-coherence as discussed in Sec. 4.3. However, in our benchmark 3-b, BFL layout has much smaller working set size compared to VEB. Since the robot model is placed inside the power plant model and BVs of the plant have high overlaps with other BVs, BFL is more suitable in that case.

7.3.2. Ray Tracing

We implemented an interactive ray tracer based on kd-trees [Wal04]. To allow different layouts of kd-nodes, we change intermediate kd-nodes to have the left and right child indices; therefore, the size of each kd-node is 16 bytes as opposed to 8 bytes used in the state-of-the-art kd-tree representation.

We applied our layout algorithm to compute cache-oblivious layouts of kd-trees. Since the implicitly computed

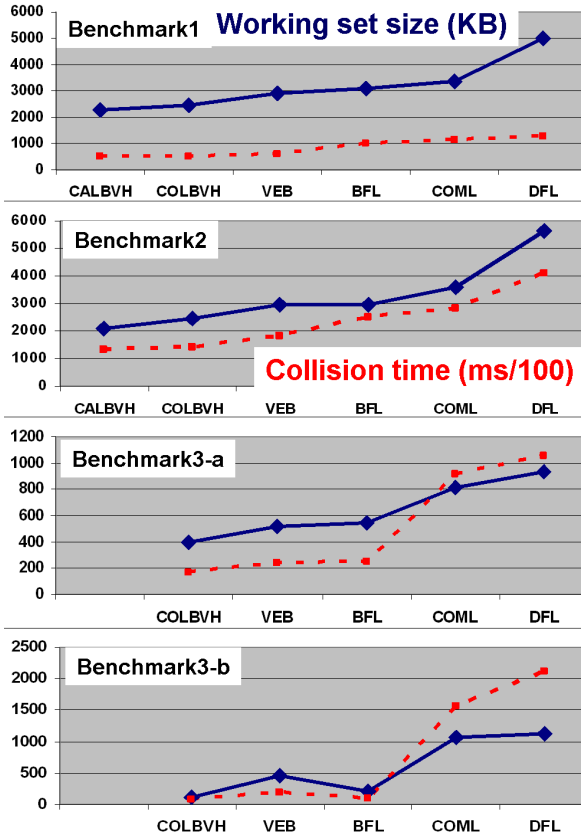

Figure 6: ***Performance of Collision Detection:*** Average collision query time and the size of working set for collision detection in our benchmarks. We highlight the performance of other layouts (i.e. VEB, DFL and BFL) and compare them with our layouts (COLBVH and CALBVH). VEB is the van Emde Boas layout, DFL and BFL are the depth-first and breadth-first layouts, respectively. We obtain 26%–2600% improvement in the performance of collision queries based on reduced working set size and fewer cache misses. Moreover, the performance of cache-oblivious layout (COLBVH) is comparable to that of cache-aware layouts (CALBVH) (in the first and second benchmarks) and consistently shows better performance over other layouts.

BV of each kd-node is fully contained in its parent BV and some BVs can have zero volumes, we use the surface areas of BVs as the volume for probability computation. Such techniques have also been used by kd-tree construction algorithms [MB90]. We compute the probability based on the ratio of surface areas and use our layouts of kd-trees without any modification of runtime ray tracer.

We tested different layouts of the Lucy model consisting of 28 million triangles. Please note that the kd-tree of the Lucy model is unbalanced, since the standard deviation of depth of leaf nodes is about 3. We also ensure that there is no fragmentations in the data files.

We are able to achieve 77%–180% improvement in the performance of ray tracing and able to achieve 7%–55% reduction in the size of working set compared to other layouts. In this case, the performance improvement cannot be directly measured by reduction in the working set size since the I/O access time is also affected by other factors including disk I/O sequential prefetching. Since the cache-oblivious layout stores coherent data in spatially close region on the disk, it is

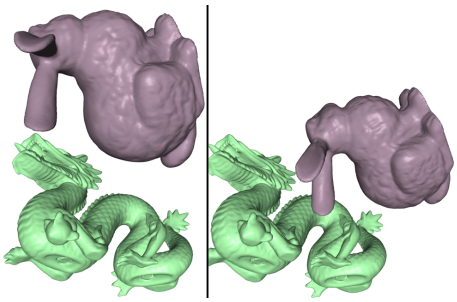

Figure 7: ***Dynamic Simulation between Bunny and Dragon Models:*** This image sequence shows discrete positions from our dynamic simulation between bunny and dragon models. We are able to achieve 26%–166% performance improvement by using our cache-oblivious layouts of OBB-trees as compared to other layouts. Moreover, the cache-oblivious layout has only 10% lower performance as compared to the cache-aware layout.

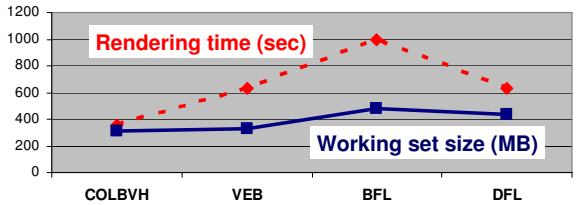

Figure 8: ***Performance of Ray Tracing:*** Average render time and the size of working set during ray tracing of the Lucy model with 28 million triangles are shown with different layouts. By using the cache-oblivious layout, we are able to achieve 77%–180% improvement in the performance of ray tracing and reduce the working set size by 7%–55%.

likely that its layout is well suited to reducing disk I/O access times. We report the rendering time and the working set size in Fig. 8. The ray tracing traverses the kd-tree in the depth-first order and performs intersection tests between the BVs of kd-tree and the rays. Moreover, there is no overlap between the BVs of kd-nodes that are not descendant to each other. Therefore, depth-first layout is likely to be more coherent at runtime traversal compared to van Emde Boas (VEB) layout and the breadth-first layout (BFL). Our experimental results also support this conjecture.

8. Analysis and Limitation

In this section, we analyze the performance of our algorithm and discuss some of its limitations.

8.1. Performance Analysis

We can achieve performance improvement by storing related data into one block since many current caching architectures employ a block fetching mechanism [AV88]. The performance of the cache-oblivious layouts of BVHs strongly depends on the size of each BV relative to the size of the cache block. We observe higher performance improvement when we have many disk I/O accesses, which typically have a block size of 4KB. On the other hand, we do not achieve significant improvement in terms of reducing L1/L2 cache misses, which have block size of 64 bytes. In the extreme case, when the block size is exactly the same as the size of each element used in the layout computation, there is very little improvement due to our layouts.

8.2. Comparison with Cache-Oblivious Mesh Layouts

Yoon et al. [YLPM05] presented a cache-oblivious mesh layout (COML) to minimize the number of cache misses during runtime accesses on the mesh. We were able to achieve 40%–2000% performance improvement over the cache-oblivious mesh layout (COML). We attribute the improvement of our new algorithm to the following reasons:

- **Clustering method:** The COML method uses a graph partitioning to compute layouts for any graph that may correspond to a polygonal mesh or a BVH. However, there is no guarantee that clustering the outputs of the graph partitioning on the input graph satisfy the convexity property, which is very important to compute cache-coherent layouts of trees. Therefore, the constructed layout of the BVH may be far from an optimal layout that minimizes the size of the working set during traversal of collision queries. Instead, our layout algorithm optimized for BVHs always guarantees that the clustering output satisfies the convexity property. At the same time, our layout maximizes the probabilities that BVs, which are accessed together due to parent-child locality, are stored in the same cluster.
- **Probability computation:** In order to construct an input graph for the COML algorithm, edges should be created to represent access patterns of traversals of collision queries. However, it is difficult to consistently compute weights for edges that represent parent-child or spatial localities in the graph. The edge creation methods for BVHs described in Yoon et al. [YLPM05] do not adequately represent access patterns of the traversals. On the other hand, our algorithm (COLBVHs) considers two different localities and quantifies the probability that a node is accessed during runtime traversal based on the geometric relationship between the BVs. As a result, we are able to capture more accurate runtime access behavior on the BVHs for layout computation.

8.3. Limitations

Our algorithm works well for our current set of benchmarks. However, it has certain limitations. Our greedy algorithm is based on greedy heuristics to compute cache-coherent layouts based on parent-child locality. Therefore, there is no guarantee that our cache-oblivious layouts of BVHs always reduce the number of cache misses or the size of the working set. Moreover, our current layout algorithm assumes that traversals of collision queries start from the root node of the BVH.

9. Conclusion and Future Work

We have presented a novel algorithm to compute cache-efficient layouts of BVHs. We do not make assumptions about the cache parameters or the memory hierarchy and take advantage of coherent data access patterns on BVHs. We describe a new probabilistic model to predict the runtime access patterns of applications on BVHs. We decompose the access patterns during the traversal of BVHs into a set of search queries and utilize parent-child and spatial localities between the accessed nodes. Our layout algorithm considers these two localities and reduces the number of cache misses and the size of working set. We have used cache-oblivious layouts of BVHs for collision detection between complex models and ray tracing of massive models. We were able to achieve 26%–2600% improvements on the performance over different layouts.

There are several areas for future work. We would like to extend our probability formulation that predicts runtime data access patterns of collision queries to consider other proximity queries such as minimum separation distance. We also would

like to compute cache-coherent layouts of other hierarchical representations such as multiresolution meshes (e.g. vertex hierarchies) by extending our layout algorithm. In this regard, we already applied our layout to an LOD hierarchy combined with a kd-tree for interactive ray tracing [YLM06] and were able to observe up to 60% improvement. Also, we would like to apply our probability formulation to construction of BVHs in order to reduce the number of intersection between two objects and further improve the runtime performance. Finally, we would like to design layout algorithms for deformable models [LYTM06].

Acknowledgments

We would like to thank Ivy Bigelow, Russ Gayle, Dawoon Jung, Ted Kim, Ming Lin, Peter Lindstrom, Brandon Lloyd, Valerio Pascucci, Stephane Redon, and anonymous reviewers for useful discussions and feedback. The Bunny, Dragon, and Lucy models are courtesy of Stanford University. The power plant model is courtesy of an anonymous donor. This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, ONR Contract N00014-01-1-0496, DARPA/RDECOM Contract N61339-04-C-0043, Intel, and, LOCAL LLNL LDRD project (05-ERD-018). Some of the work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

[ABF04] ARGE L., BRODAL G., FAGERBERG R.: Cache oblivious data structures. *Handbook on Data Structures and Applications* (2004).

[ABFC*03] ALSTRUP S., BENDE M. A., FARACH-COLTON E. D. D. . M., RAUHE T., THORUP M.: Efficient tree layout in a multilevel memory hierarchy. *Computing Research Repository (CoRR)* (2003).

[AV88] AGGARWAL A., VITTER J. S.: The input/output complexity of sorting and related problems. *Commun. ACM 31* (1988), 1116–1127.

[CM95] COLEMAN S., MCKINLEY K.: Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation* (1995), 279–290.

[Dee95] DEERING M. F.: Geometry compression. In *SIGGRAPH 95 Conference Proceedings* (Aug. 1995), Cook R., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 13–20. held in Los Angeles, California, 06-11 August 1995.

[DPS02] DIAZ J., PETIT J., SERNA M.: A survey of graph layout problems. *ACM Computing Surveys 34*, 3 (2002), 313–356.

[Eri04] ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann, 2004.

[FLPR99] FRIGO M., LEISERSON C., PROKOP H., RAMACHANDRAN S.: Cache-oblivious algorithms. *Symposium on Foundations of Computer Science* (1999), 285–297.

[GI99] GIL J., ITAI A.: How to pack trees. *Journal of Algorithms* (1999).

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96* (1996), 171–180.

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

[Hop99] HOPPE H.: Optimization of mesh locality for transparent vertex caching. *Proc. of ACM SIGGRAPH* (1999), 269–276.

[Hub93] HUBBARD P. M.: Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality* (October 1993).

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)* (2003), vol. 22, pp. 935–942.

[IL04] ISENBURG M., LINDSTROM P.: *Streaming Meshes*. Tech. Rep. UCRL-CONF-201992, LLNL, 2004.

[KHM*98] KLOSOWSKI J., HELD M., MITCHELL J., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1 (1998), 21–37.

[LP01] LINDSTROM P., PASCUCCI V.: Visualization of large terrains made easy. *IEEE Visualization* (2001), 363–370.

[LYTM06] LAUTERBACH C., YOON S.-E., TUFT D., MANOCHA D.: *RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs*. Tech. Rep. TR-06-010, Univ. of North Carolina at Chapel Hill, 2006, 2006.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer* (1990).

[MC95] MIRTICH B., CANNY J.: Impulse-based simulation of rigid bodies. In *Proc. of ACM Interactive 3D Graphics* (Monterey, CA, 1995).

[PF01] PASCUCCI V., FRANK R. J.: Global static indexing for real-time exploration of very large regular grids. *Super Computing* (2001), 225–241.

[Sag94] SAGAN H.: *Space-Filling Curves*. Springer-Verlag, 1994.

[SCD02] SEN S., CHATTERJEE S., DUMIR N.: Towards a theory of cache-efficient algorithms. *Journal of the ACM 49* (2002), 828–858.

[Ter03] TERDIMAN P.: Opcode: Optimized collision detection, 2003. http://www.codercorner.com/Opcode.htm.

[vEB77] VAN EMDE BOAS P.: Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.* (1977).

[VG91] VELHO L., GOMES J. D.: Digital halftoning with space filling curves. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (July 1991), Sederberg T. W., (Ed.), vol. 25, pp. 81–90.

[Vit01] VITTER J.: External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys* (2001), 209–271.

[VM04] VARADHAN G., MANOCHA D.: Accurate minkowski sum approximation of polyhedral models. In *Pacific Conference on Computer Graphics and Applications* (2004), pp. 392–401.

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

[WHG84] WEGHORST H., HOOPER G., GREENBERG D.: Improved computational methods for ray tracing. *ACM Transactions on Graphics* (1984), 52–69.

[YL06] YOON S.-E., LINDSTROM P.: *Mesh Layouts for Block-Based Caches*. Tech. Rep. UCRL-TR-220368-DRAFT, Lawrence Livermore National Lab., 2006.

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-LODs: Interactive LOD-based Ray Tracing of Massive Models. *The Visual Computer (Pacific Graphics)* (2006). To appear.

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH* (2005).

[YMLP05] YOON S.-E., MANOCHA D., LINDSTROM P., PASCUCCI V.: OpenCCL: Cache-Coherent Layouts of Meshes and Graphs, 2005. http://www.cs.unc.edu/ geom/COL/OpenCCL/.

[YSGM04] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick-VDR: Interactive View-dependent Rendering of Massive Models. *IEEE Visualization* (2004), 131–138.