Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing

Christian Lauterbach[1] Sung-Eui Yoon[2] Dinesh Manocha[1]
[1] University of North Carolina at Chapel Hill [2] Korea Advanced Inst. of Sci. and Tech.

ABSTRACT

We present a novel hierarchical representation, Ray-Strips, for interactive ray tracing of complex triangle meshes. Prior optimized algorithms for ray tracing explicitly store each triangle in the input model. Instead, a Ray-Strip takes advantage of mesh connectivity for compact storage, efficient traversal and ray intersections. As a result, we considerably reduce the memory overhead of the original model and the hierarchical representation. We also present efficient algorithms for single ray and ray packet traversal using Ray-Strips. Furthermore, we demonstrate that our representation can utilize the SIMD capabilities of current CPUs for incoherent ray packets and single rays. We show the benefit of Ray-Strips on models with tens of thousands to tens of millions of triangles. In practice, our approach can reduce the storage overhead of interactive ray tracing algorithms by up to five times compared to standard approaches. Moreover, we improve the runtime performance of ray tracing on large models.

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

Keywords: ray tracing, bounding volume hierarchies, compact representation, triangle strips

1 INTRODUCTION

Ray tracing has recently emerged as an alternative rendering method to rasterization for interactive rendering. Ray tracing has been extensively studied for almost three decades due to its ability to simulate the physical process of light propagation and to generate various rendering effects like transparency, shadows and indirect illumination. There has been renewed interest in real-time ray tracing due to the exponential growth rate of processing power and recent hardware trends of using multiple cores. Since ray tracing algorithms are embarrassingly parallel and easily map to multi-core and multi-processor systems, it is expected that the performance of ray tracing will continue to improve significantly.

One of the main benefits of ray tracing is that its asymptotic time complexity increases as a logarithmic function of the number of triangles of the model. This makes ray tracing an attractive approach for rendering large data sets composed of tens or hundreds of millions of primitives. However, the model representation and acceleration data structures used to perform ray tracing such massive models can take tens of giga-bytes of storage. This large storage overhead can significantly affect the performance of ray tracers [41].

Recent algorithmic improvements such as ray coherence techniques [29, 34] are able to exploit the coherent hierarchy traversal behavior of packets or groups of rays and improve the runtime performance. These algorithms have been shown to work well with architectural models, where most visible triangles cover large areas in terms of number of pixels in the image space. As a result, groups of rays can maintain high spatial coherence in terms of hierarchy traversal behavior in such models. However, these methods have decreased effectiveness on highly complex and detailed models since coherence decreases with model complexity and small triangles [41]. In particular, recent advances in 3D scanning and scientific simulation result in datasets composed of tens or hundreds of millions of small triangles.

Most ray tracing algorithms assume that input models are "triangle soup", i.e. all triangles are considered unordered and independent. However, many meshes used in computer games, CAD, scanned data, and scientific simulations have well-defined connectivity between the triangles. This property has been exploited to compute a sequential order of triangles and using that order for higher rasterization throughput. This includes representations based on triangle strips or rendering sequences [9, 18], which are designed to reduce the memory requirements and transformation cost during triangle rasterization. Unfortunately, such compact and efficient representations developed for improving the rasterization performance are not directly applicable to ray tracing. This is due to the fact that triangle strips or rendering sequences are mainly designed for sequential access during rasterization. On the other hand, current ray tracing algorithms access the underlying data representations including the triangles in a random manner. Therefore, most optimized ray tracing methods store each triangle and its geometric coordinates separately to facilitate such random accesses. However, these representation can have high storage overhead.

Main Contributions: In this paper, we address the problem of representing triangle meshes for fast ray tracing. We present a novel mesh and hierarchy representation, *Ray-Strips*, for interactive ray tracing. A Ray-Strip includes a list of vertices, which implicitly represents a list of triangles like a triangle strip. Moreover, our representation also includes a light-weight hierarchy defined on the list. Therefore, our Ray-Strips allow efficient tree traversal as well as efficient ray intersection tests between a ray and the triangles represented by the Ray-Strip. We also present techniques to improve the SIMD utilization for incoherent ray packets based on Ray-Strips. To evaluate our method, we apply our method to different benchmarks including complex scanned, CAD, and simulation data sets. Also, the model complexity of our benchmarks varies from tens of thousands of triangles to a few hundred million triangles.

As compared to prior approaches, our representation offers the following advantages:

- **Memory efficiency:** Ray-Strips require much less memory for both the hierarchy as well as the geometric primitives by compactly storing each of them. Compared to previous approaches, Ray-Strips are able to reduce memory requirements by up to 5 times. This can considerably improve the performance of out-of-core ray tracers and makes it possible to handle massive models on commodity hardware such as laptops with limited main memory and disk space.

- **SIMD utilization:** Ray-Strips enable us to efficiently handle incoherent packets by switching the usage of SIMD units to primitive intersection tests from hierarchy traversal.

- **Faster single-ray performance:** Due to the improved memory efficiency and SIMD utilization, we are able to improve the overall the run-time performance of ray tracing on massive models significantly.

Organization: The rest of the paper is organized as follows: In Section 2, we briefly survey related work on mesh representations and ray tracing. We discuss issues in ray tracing massive models in Section 3 and give an overview of our approach. Section 4 presents our mesh representation in detail and Section 5 describes efficient traversal and intersection algorithms based on our representation. We discuss our implementation and highlight the performance on different benchmarks in Section 6.

2 Previous work

In this section we give a brief overview on previous work on ray tracing of massive models and mesh representations.

Coherent ray tracing: Current interactive ray tracers use packets of rays to amortize the costs of hierarchy traversal and primitive intersection tests over multiple rays by assuming there is strong coherence between the rays in each packet. The packet traversal algorithm was first introduced for kd-trees [34] and has been applied to different structures such as bounding volume hierarchies [21, 35, 43], s-kd-trees [27, 40], and grids [37]. Recently, Reshetov et al. [29] proposed a multi-level ray tracing method, which can split ray packets and adapts ray packet size to the geometric complexity of the model.

Out-of-core techniques: One of the major challenges when dealing with massive models is the high memory footprint during rendering. To address this issue, many out-of-core techniques have been proposed in visualization [5] and rasterization [8, 14]. Particularly, Wald *et al.* [36] improved the performance of ray tracing by employing an explicit memory handling scheme and using precomputed light-field-like representations of the scene to hide the latency incurred during the loading process. Pharr *et al.* [28] proposed an algorithm to optimize memory coherence in ray tracing by reordering rays so that they access the primitives in a coherent manner. They mainly showed their performance improvement for offline ray tracing. Our approach is complementary to these algorithms and can be combined with them.

Simplification: Levels-of-detail (LODs) have been widely used to accelerate the performance of rasterization of large polygonal data sets [23] and have been applied to improve the performance of ray tracing [6, 32, 38, 41]. However, these LOD techniques may introduce visual artifacts in the final images and their representations can require bigger disk storage than those of non-LOD based methods. On the other hand, our method reduces memory and disk requirement without introducing any visual artifacts.

Mesh and Hierarchical Representations: There is considerable work on computing compact mesh representations for triangulated models, especially for GPU rasterization. Since modern GPUs maintain a small buffer to reuse recently accessed vertices, triangle ordering can improve the frame rates. This approach was pioneered by Deering [9]. The resulting ordering of triangles is called a *triangle strip* or *a rendering sequence*. There has been considerable work on designing improved algorithms to generate these sequences [10, 18, 26] and extending them to view-dependent rendering [19, 42].

In the ray tracing literature, there is work on improved representations for subdivision meshes [6,20,25,32] and tessellated versions of surfaces such as Bézier patches [4]. Amanatides and Choi [2] presented an edge-based ray-mesh intersection method for regular meshes using Plücker coordinates. All of these methods utilize the fact that connectivity between vertices is implicit and therefore does not need to be explicitly stored. In contrast, there is relatively little work on efficiently handling triangular meshes. As one example, Galin and Akkouche [13] use a triangle fan decomposition of the model and show an efficient method for intersecting a ray against triangle fans.

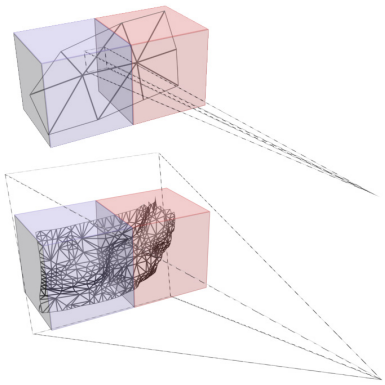


Figure 1: **Ray coherence for packets of rays:** *Consider a group of rays organized as a ray packet (visualized here by the enclosing frustum). In the upper image, the complexity of the geometry proportional to the frustum size is low, so rays will likely traverse the same part of the hierarchy and will hit the same or close primitives. In the lower image, the geometric complexity is very high relative to the frustum size. Traversal and intersection overhead are high since rays will traverse very different paths in the tree.*

Compression: There is considerable work on compactly representing triangle meshes [1]. Most previous mesh compression schemes were mainly designed to achieve maximum compression as they were targeted for archival use or transmission. However, it is not clear whether these compression algorithms can be directly used to improve the performance of ray tracing algorithms, as they may not provide random access to the mesh primitives.

Coherent layouts: Since the stored order of triangles and vertices can affect the performance of applications, coherent layouts minimizing the number of cache misses have been researched. These includes space filling curves [22, 30], cache-aware layouts [17], cache-oblivious mesh layouts [42], etc.. Our algorithm is complementary to all the layout techniques and can potentially be combined with some of them to further improve the performance of ray tracing.

3 Overview

In this section we briefly explain the two main issues that arise during ray tracing of complex models: low ray coherence and high memory requirements. Next, we present an overview of our approach that addresses both these issues. We also further assume that the basic input primitives are triangulated models.

3.1 Ray Coherence

Ray coherence techniques such as ray packet traversal and intersection tests [34] and multi-level ray tracing [29] can significantly improve the performance of ray tracing. These approaches simultaneously perform intersection tests for rays in a group against a hierarchy (e.g. kd-tree or bounding volume hierarchy) and geometric primitives of a model. Moreover, we can utilize the SIMD functionality of current CPUs or GPUs to perform hierarchy traversal and intersection tests on multiple rays simultaneously.

One major disadvantage of ray packet traversal and intersection tests is its behavior for incoherent packets, when all the rays in a group are forced to traverse different parts of the hierarchy. For complex models, this can result in a low utilization ratio of SIMD vector resources. Moreover, this can make packet traversal slower than standard ray tracing method with single rays due to the overhead of preparing data for SIMD usage. This behavior is more likely to happen as the number of primitives in the model increases (see Fig. 1). Therefore, a large part of the computational resources

during ray tracing is not fully utilized in these kinds of models. As the SIMD functionality will be more widely used in futures CPU and future many-core architectures, such incoherent packets can result in significant performance issues.

3.2 Memory Requirements

Ray tracing algorithms typically need two major data structures: a hierarchical acceleration data structure such as kd-trees or bounding volume hierarchies (BVHs) and a geometric representation of the primitives associated with the hierarchy. The acceleration structure is used to improve the performance of ray tracing by allowing a quick search for the primitives that a ray or group of rays can potentially intersect. However, the acceleration structure adds a significant memory overhead in addition to storing the actual primitives of an input mesh. Given a model consisting of n triangles, an optimized BVH with one triangle per leaf has $2n-1$ BVH nodes and each node requires 32 bytes [21, 35]. Also, most optimized software implementations store additional information such as the normal and a projection plane. In particular, the system described in [33] uses 48 bytes per triangle for storage. Note that this does not apply to hardware ray tracers [39] which have fast triangle intersection units and can therefore use indexed triangle lists. As an example, the Stanford Lucy model consists of approximately 14 million vertices and 28 million triangles. In this case, the minimal indexed triangle list including vertices requires a total of 482 MB of data, and the optimized triangle representation $1,284$ MB. A BVH for the model adds another $1,712$ MB.

Compared to efficient mesh representations such as triangle strips for rasterization, data representations for ray tracing take much more memory space. Moreover, prior research [36, 41] has shown that the performance of ray tracing reduces drastically when the acceleration data structures and the primitives do not fit into the main memory. Considering that one of the attractions of ray tracing is its logarithmic performance with regard to the number of input primitives, it is important that we design compact memory representations for interactive ray tracing.

3.3 Our Approach

We introduce a novel mesh representation called Ray-Strips. This representation is similar to triangle strips used to accelerate rasterization. However, this representation is designed such that it is amenable to ray tracing while maintaining compactness. Each Ray-Strip represents a mesh that is a subset of the overall input model, and all of them are organized in a high-level scene hierarchy, where each leaf node references a Ray-Strip. Ray-Strips represent a balanced hierarchy of the triangles as encoded by the strips; therefore, Ray-Strips themselves represent a low-level scene hierarchy.

The choice of a two-level hierarchy stems from the observation that the high levels of the hierarchy have a major impact on ray tracing performance. Therefore, it is desirable to use an optimized tree structure such as one computed with the surface-area heuristic (SAH) [15]. On the other hand, the tree structure closer to the leaf nodes in the hierarchy is likely to be evenly distributed in space, assuming that the underlying geometric primitives are connected and close. Therefore, such lower levels of the hierarchy can be split evenly without introducing significant bad split decisions in terms of the SAH metric. This means that at some point it is possible to switch to splitting the hierarchy at the object median without sacrificing the global quality of the hierarchy. Once we split the hierarchy at the object median, then we can represent the hierarchy as a balanced hierarchy, which requires less memory because no child pointers are necessary.

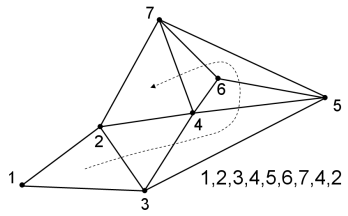


Figure 3: **Example triangle strip:** *This figure shows a simple mesh consisting of seven triangles, which can be represented as a triangle strip of nine vertices shown on the right bottom.*

3.3.1 High-level hierarchy

Our representation consists of a high-level hierarchy and a strip representation for the triangle mesh. The high-level hierarchy is built on the triangular primitives, but only up to a certain triangle count or other subdivision criterion. Therefore, each leaf of the high-level hierarchy references a subset of triangle primitives. This hierarchy can be any acceleration structure such as a kd-tree, BVH, grid or spatial kd-tree. Note that the size of the tree is significantly smaller and the bounding boxes of the tree are large since each node of the tree contains many triangle primitives. Therefore we can achieve higher ray coherence during traversal the high-level hierarchy. We use a BVH of axis-aligned bounding boxes and construct the hierarchy using the SAH for best performance. We perform single ray and ray packet traversal on our high-level hierarchy in the same manner as described in [21, 35].

3.3.2 Ray-Strips

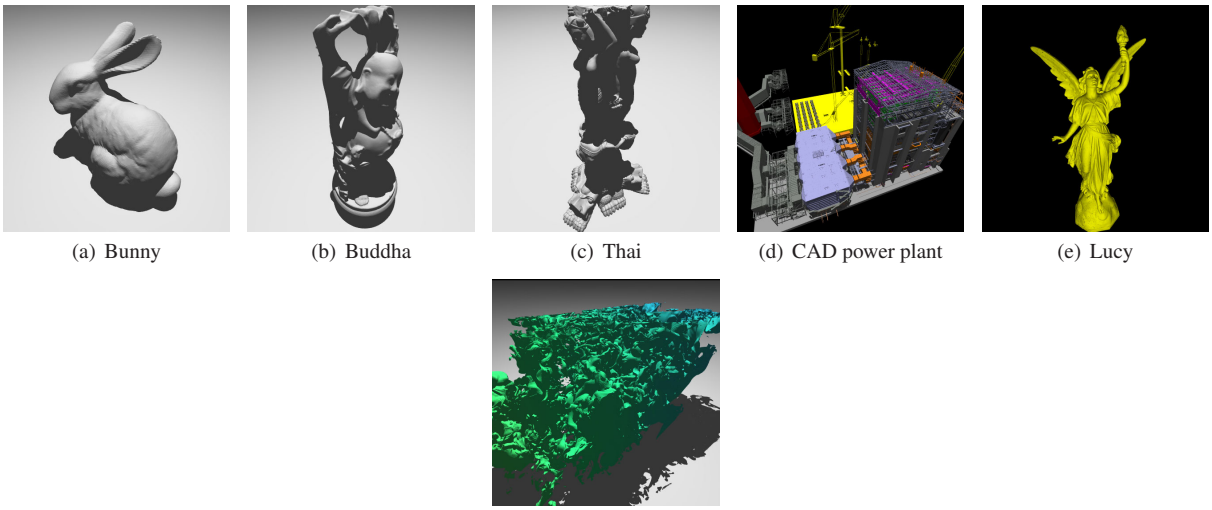
Our low-level hierarchy represents a triangle mesh and is intended both to reduce the memory footprint of the geometric primitives in the mesh as well as to allow efficient traversal and intersection. Ray-Strips are based on triangle strips, but contain an object hierarchy on the strip that is similar to spatial kd-trees and allows hierarchy traversal for efficient culling. One major aspect of this representation is that at any step during the traversal it is possible to perform intersection tests with all the triangles contained in the current sub-tree instead of traversing further. Later, we show that this property can speed up ray tracing for incoherent packets and single rays by efficiently using SIMD instructions, which perform intersection tests on a single ray with multiple primitives at the same time. This is in contrast to prior SIMD algorithms that intersect multiple rays with a single triangle. The traversal and intersection algorithms are described in Section 5.

4 Ray-Strip Representation

In this section we explain our Ray-Strip representation and its relationship with our two levels of hierarchy. We also present our algorithm to build Ray-Strips.

4.1 Representation

The Ray-Strip representation is based on triangle strips. Each triangle strip represents a set of connected triangles and is stored as a list of vertices, where connectivity is given implicitly by the order of vertices. Fig. 3 shows an example of a simple mesh and a corresponding triangle strip. However, even though a triangle strip is an efficient way to represent a triangle mesh, intersecting a ray with each triangle in the strip can be rather slow for ray tracing. To address this issue, we generate a memory-efficient spatial kd-tree hierarchy on top of the strip. The hierarchy allows efficient traversal for determining the visible triangles in the Ray-Strip.



(a) Bunny (b) Buddha (c) Thai (d) CAD power plant (e) Lucy

(f) Isosurface

Figure 2: **Benchmark scenes:** *We test our Ray-Strip representation on a range of complex scenes of different types including scanned, scientific visualization and CAD models. From upper left: Bunny (scanned, 69K), Buddha (scanned, 1.2M), Thai (scanned, 10M), Power plant (CAD, 12.7M), Lucy (scanned, 30M), Iso-surface (10M).*

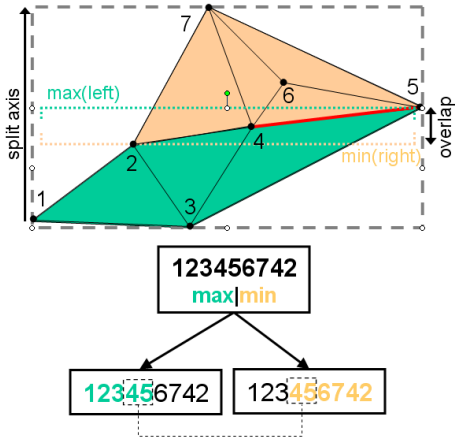


Figure 4: **Strip hierarchy:** *We subdivide a strip at the center edge (shown in red) and then record the minimum and maximum bounds on one axis for the resulting two children. This defines a balanced spatial kd-tree on the mesh such that only the two bounds need to be stored. The interval in the strip is defined implicitly. Please note that intervals overlap since both children reference the center edge 4-5.*

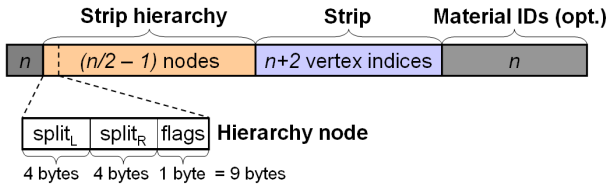


Figure 5: **Ray-Strip structure:** *The data structure for storing a Ray-Strip consists of the header recording the number of triangles n, the hierarchy defined on the strip as well as the triangle strip representing the geometry. We also allow optional per-triangle information, such as material references in this case. Note that the size of the Ray-Strip is known a priori and only depends on n.*

Given n triangles in a mesh, our Ray-Strip consists of a list of $n+2$ vertex indices to represent the triangles. We recursively split the strip at the median edge into two parts of equal number of triangles (or two sub-trees whose height only differs by one) in order to define a balanced tree on the strip. An example of this operation is shown in Fig. 4: The red edge is the one that defines the split and will be a part of the two resulting sub-strips.

In order to use the balanced tree on the strip, we also need to know the spatial bounds of the nodes during traversal. Instead of storing the full bounding box, we select the axis of the least absolute overlap between children. Then, we store the maximum bound for the left and the minimum bound for the right child. Effectively, we store the split axis as well as the coordinates for the two bounds, which is very similar to spatial kd-trees (s-kd-trees) [40]. Currently, we split down to sub-strips of length 4, i.e. representing 2 triangles, which has performance advantages for SIMD usage. Because we always perform median splits, the hierarchy is balanced (or the left sub-tree has at most one level more than the right) and can therefore be stored in an array without the need for child pointers. In addition, each node directly corresponds to a consecutive sequence of vertices on the vertex list of the triangle strip. Therefore, it is not necessary to actually store the leaf nodes as the traversal algorithm can detect whenever it has reached a sub-strip of sufficient size. Because the spatial kd-tree is an object hierarchy, the number of nodes in the tree is also known: for n triangles, we need a total of $\lceil n/2 \rceil - 1$ nodes for the hierarchy.

Our decision to use s-kd-trees was motivated by picking the most light-weight structure to save memory and the fact that s-kd-trees have previously been shown to have similarly fast traversal to kd-trees [40]. B-KD-trees [39] and more sophisticated related hierarchies [16] may have better behavior where empty space subdivision is important, but this is not as much of an concern for fully connected meshes .

The spatial kd-tree always assumes that the left sub-strip in a split is the one on the left with regard to the split axis. This can be true, but in general will only sometimes be the case. Since we cannot modify the strip order, we need an additional flag per split that indicates whether the split is the normal split order or in reverse. Overall, a node in the hierarchy needs to store two split coordinates, the split axis and the reverse flag. We encode these two data into one byte for simplicity, although technically they only need 3 bits (2 for the axis and one for the flag). Fig 5 shows the actual memory representation of a Ray-Strip: a header stores the num-

ber of triangles in the strip, then all the hierarchy nodes are stored and finally the actual triangle strip as a sequence of vertex indices. We also store additional per-triangle information, which currently is used for saving a material reference (as a 16-bit integer) for shading purposes. If the material properties are not used or per-vertex colors are used, then this information does not need to be stored.

4.2 Strip Generation

The decomposition of an input mesh to triangle strips is a well-studied problem called stripification and several algorithms are known in the literature [10, 18, 26]. For a given input model, we can subdivide the model into triangle strips and build the high-level hierarchy on those strips. Unfortunately, typical stripification algorithms tends to generate very long and thin triangle strips. However, the triangle strips optimized for high GPU vertex cache utilization may not be good for ray tracing since they do not consider any spatial relationships such as spatial distribution of geometric primitives or overlaps, which have a decisive effect on the run-time performance of ray tracing.

Our approach for finding suitable triangle strips is as follows: we first run the SAH construction algorithm to decompose an input mesh into smaller sub-meshes, each of which has a group of triangles below a threshold count. Then, we build a high-level hierarchy from the sub-meshes. As a next step, we attempt to build a triangle strip from each chunk of the mesh. For strip computation, we use the Stripe library [12] for computing a stripified version of the mesh. Note that even for a connected mesh, the existence of a triangle strip fully spanning a mesh is not guaranteed and that the problem of computing an optimal decomposition has been shown to be NP-complete [11]. Therefore, if we are not able to build a strip for a sub-mesh, we instead partition it into as many sub-meshes as necessary to get a decomposition. By performing this partition, we create new child nodes for a node containing the sub-mesh in the high-level hierarchy.

5 Ray Tracing using Ray-Strips

In this section, we present how to perform fast ray tracing using Ray-Strips. This includes two operations: hierarchy traversal and triangle intersection. We present an methods for single rays as well as ray packets.

5.1 Hierarchy Traversal

The Ray-Strip hierarchy is essentially a spatial kd-tree. Therefore, the Ray-Strip can be traversed in a very similar manner to the spatial kd-tree (see [40], also illustrated in Fig. 6). Starting at the root node, the bounding box of the mesh is split in the axis stored in the node. Given a ray, the distances d_L and d_R to both split planes can be tested to determine whether it hits the left, the right or both children, where the order is given by the ray's direction.

If the ray intersects both the children, the near one – as determined by the ray's direction – is traversed first and the far one is pushed on a stack. The children are processed in the same manner by updating the bounding box from the parent with the child's respective split plane. Since the hierarchy is subdivided according to fixed rules and the number of triangles in the mesh is known, the traversal just needs to keep track of how many triangles are left in the current sub-tree to know when it encounters a leaf. This is performed by updating the interval in the triangle strip representing the triangles whenever the traversal jumps to another node. As mentioned above, this also involves the reverse flag for the node to find out which side of the interval corresponds to the left and right children, respectively. The pseudo-code implementing this traversal is shown in Algorithm 1.

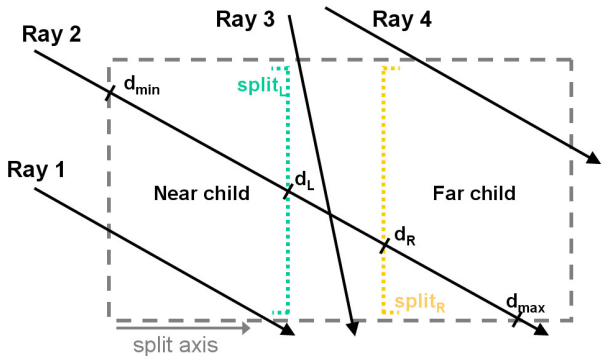


Figure 6: **Traversing a node:** *Given an inner node of the mesh hierarchy, there are two split planes associated with it. The ray hits the left child if the distance to the left plane d_L is larger than the node entry distance d_{min} and hits the right child if the node exit distance d_{max} is larger than d_R. For ray 1, only the near child is traversed, for ray 2 both need to be traversed and ray 4 only traverses the far child. Note the special case for ray which traverses neither of the two children.*

Algorithm 1 Traversal of the Ray-Strip hierarchy

```
node ← root, left ← 0, right ← #indices - 1
[d_min, d_max] = ray.intersectWithBoundingBox()
while node != NULL do
  while (right - left + 1) > 2 do
    d_L ← ray.distToNear(node.axis, node.split1, node.split2)
    d_R ← ray.distToFar(node.axis, node.split1, node.split2)
    if d_L < d_min then
      if d_R > d_max then
        [node,left,right,d_min,d_max] ← stack.pop()
      else
        [node,left,right] ← node.farChild(ray,node.reverse)
      end if
      continue
    else if d_R ≤ d_max then
      stack.push(node.farChild(ray,node.reverse),
      max(d_R, d_min), d_max)
    end if
    d_max ← min(d_L, d_max);
    [node,left,right] ← node.nearChild(ray,node.reverse)
  end while
  intersectStrip(left, right)
  [node,left,right,d_min,d_max] ← stack.pop()
end while
```

5.2 Intersection Computation

At every step in the traversal, the ray tracer can decide to intersect with the triangles contained in the node, which are defined by the current interval in the triangle strip sequence. The naïve approach would be to test every triangle by itself using a standard triangle intersection algorithm, but that would ignore information available as part of the strip representation. Instead, we take advantage of the connectivity information. Specifically, we use an edge-based intersection that tests the ray for containment using Plücker coordinates [31], which allows us to test the orientation of a ray relative to an edge. Given three edges defining a triangle in a consistent order (clockwise/counter-clockwise), the ray intersects the triangle if and only if the signs of the Plücker edge tests match. Given an interval in the triangle strip, we can compute all the edge tests directly and then test each consecutive set of three edge results for intersection with the respective triangle. Since there are shared edges, this is more efficient than testing each of the triangles. In particular, we can easily test 4 edges at the same time by using SIMD instructions

on current CPUs. This is particularly effective when tracing just one ray because we utilize data parallelism.

5.3 Ray Packets

Both the traversal and the intersection algorithms described above can be extended to handle ray packets. For traversal, the algorithm given above changes in that a sub-tree is traversed if the bounding box is intersected by any of the rays. For intersection, the edge tests have to be performed for each ray in the packet. However, if the rays in the packet share the same origin, the Plücker intersection can be optimized to reuse the coordinate computation, as presented in [3].

An important issue for ray packet tracing on massive and complex models is that rays can be very incoherent at the lower levels of the hierarchy due to small triangles and the geometric detail. This leads to traversal and intersection steps to have one or very few "active" rays, i.e. rays intersecting the current sub-tree. Ray-Strips allow us to detect those cases and switch to edge intersection whenever appropriate. Thus, when the number of active rays becomes smaller than a certain threshold, we switch to single ray intersection for all the active rays. This improves performance because of data parallelism as well as generally more coherent memory accesses.

6 Implementation and Results

We now present results from our system that uses Ray-Strips and compare its performance to prior methods. We also analyze the performance of our method and discuss its limitations.

6.1 Results and Comparison

We have implemented the Ray-Strip representation and traversal methods on a Intel Core 2 architecture Xeon system at 2.5 GHz and 2 GB of RAM. Although the system has multiple cores, we only use one thread for rendering. We use the Intel SSE instruction set which allows to perform traversal and intersection of 4 rays (a 2×2 ray packet) simultaneously similar to other interactive ray tracing implementations. All performance numbers are given in frames per second at a resolution of 512×512 pixels. We test our method on several benchmark scenes of varied properties and complexity, ranging from 67 thousand to 30 million triangles (see Fig. 2.)

Memory complexity: Table 1 summarizes the results for memory reduction when using Ray-Strips for our benchmark scenes. We compare the results to a current BVH implementation as used in interactive ray tracing that stores one [21] or just a small number [35] of triangles per node for performance. To compare the geometry representation, the table also shows memory cost for storing the triangles both as a minimal list of 3 vertex indices and one material index (14 bytes per triangle, plus global vertex list), as well as the performance optimized triangle representation used in [33] (48 bytes per triangle). To better compare the results, we also split up the total memory footprint into the memory taken by hierarchy and actual geometry (triangles and vertices) for all approaches. The results show that Ray-Strips reduce the memory cost for all models significantly, for both hierarchy as well as geometry. Note that the impact of Ray-Strips is slightly lower for architectural and CAD scenes such as the power plant. This is due to lesser mesh connectivity compared to the meshes of 3D scanned and the iso-surface models.

We also compare the memory efficiency of Ray-Strips to several other hierarchies that were previously used to reduce the cost for storing the acceleration structure. Table 2 shows results for a subset of scenes from Table 1. kd-trees and spatial kd-trees [40] are both relatively light-weight structures that are optimized for fast ray tracing of static scenes. Previous approaches also include compressed BVH structures such as limited precision BVHs [24] as well as light-weight BVHs [7]. Both methods quantize the coordinates of the axis-aligned bounding boxes and are therefore able to reduce the memory cost for storing each node. They also store multiple references to triangles in each leaf node in order to use less nodes overall. As the results show, Ray-Strips represent the hierarchy much more efficiently than both kd-trees and spatial kd-trees, but may take somewhat more space than the compressed representations. However, the compressed values have to be decoded during traversal and bounding boxes may be enlarged due to the conservative quantization, leading to an overhead of traversed nodes. Therefore, rendering using the compressed structures can incur a larger overhead than using Ray-Strips. In addition, neither of the approaches above reduces the memory footprint for storing the geometry. However, the Ray-Strips approach is largely orthogonal to compression approaches, and could potentially be combined for a cumulative effect.

Model	Triangles	Ray-Strip (fps)	BVH (fps)
Bunny	67k	2.39	4.52
Buddha	1.2M	2.26	3.72
Thai	10M	2.29	3.04
Iso-surface	10M	1.73	2.62
Power plant	12.7M	0.70	0.39
Lucy	30M	1.85	0.04

Table 3: **Results: Rendering performance for single rays.** *This table shows rendering performance using single rays for a standard BVH implementation and the same implementation running on Ray-Strips. The resolution for rendering was* 512^2 *using one core, and only primary rays were used.*

Model	Triangles	Ray-Strip (fps)	BVH (fps)
Bunny	67k	7.04	10.42
Buddha	1.2M	5.21	5.10
Thai	10M	3.89	3.10
Iso-surface	10M	4.20	5.52
Power plant	12.7M	1.76	0.71
Lucy	30M	2.95	0.05

Table 4: **Results: Rendering performance for** 2×2 **rays.** *This table shows rendering performance using* 2×2 *ray packets, for a standard BVH implementation and the same implementation running on Ray-Strips. The resolution for rendering was* 512^2 *using one core, and only primary rays were used.*

Rendering performance: The performance results of using Ray-Strips are presented for single rays in Table 3. Although the pure rendering performance is slightly lower than a standard BVH in small models, the behavior of Ray-Strips for complex models shows that the SIMD utilization and memory complexity has effects on the frame rates. Note that in the Lucy scene the full BVH renderer had to operate out of core and therefore was slower than would otherwise be expected. We also tested the performance when using ray packet traversal for 2×2 packets (see Table 4) and found that Ray-Strips gain similar speed-ups by using SIMD units compared to standard BVH traversal and still perform very well on complex models where the standard packet traversal becomes slow due to lack of coherence.

Construction: Performing the stripification of the input model adds some overhead to the construction of the hierarchy, which is shown in Table 5. Almost all the additional time is spent in the stripification library, which is not optimized for speed. If necessary, a faster library can be used instead. However, since the main application for Ray-Strips is rendering complex models, the construction is usually performed as a preprocessing step and the hierarchies can be loaded from disk for runtime rendering. Note that the CAD models in general only have much smaller triangle strips since there is a lower degree of connectivity in the input data.

Model	Tris	Memory total (MB) / Reduction			Memory geometry (MB)			Memory hierarchy (MB)	
		Ray-Strip	minimal	optimized	Ray-Strip	minimal	optimized	Ray-Strip	BVH
Bunny	67k	1.5	5.5 (-73%)	7.4 (-80%)	1.2	1.4	3.2	0.3	4.2
Buddha	1.2M	23	87 (-74%)	116 (-80%)	11	20	49	11	66
Thai	10M	215	801 (-72%)	1068 (-79%)	163	190	457	106	610
Power plant	12.7M	395	1075 (-63%)	1361 (-71%)	196	296	583	198	778
Lucy	30M	581	2247 (-74%)	2996 (-81%)	301	535	1284	279	1712
Iso-surface	10M	220	822 (-73%)	1095 (-80%)	110	195	469	108	626

Table 1: **Results: Memory complexity.** *This table shows the memory requirements for our approach as well as the two usual other triangle representations combined with a BVH using axis-aligned bounding boxes. The total memory cost can be split up into memory requirements for the hierarchy as well as for the actual geometry. The comparison structures are a* minimal *representation (which stores just 3 vertex references and a material pointer per triangle), as well as the triangle representation optimized for intersection speed as used in [Wald 2004]. For Ray-Strips and indexed triangle lists, the size of the global vertex list is included.*

Model	Triangles	Ray-Strip	kd-tree [40]	spatial kd-tree [40]	LBVH [7]	LPBVH [24]
Bunny	67k	0.3	4.1	0.9	0.5	0.3
Buddha	1.2M	11	29	17	8.3	n/a
Lucy	30M	279	n/a	n/a	214	103

Table 2: **Results: Hierarchy memory comparison:** *We compare the relative memory complexity of several acceleration structures to Ray-Strips (all values in MB, "n/a" signifies that the model was not used in the respective paper) The spatial kd-tree is an object-level hierarchy similar to our Ray-Strip hierarchy, LBVH and LPBVH are both compressed BVH structures storing multiple primitives at leaf nodes and values were taken for the highest compression ratio in the respective papers, which usually leads to a significant degradation in performance. Note that our approach could use the compression from those approaches for further memory reduction. Neither of the techniques above compresses the geometry data.*

6.2 Analysis and Limitations

The results above show that Ray-Strips are an efficient representation for our tested complex benchmarks. In practice, the memory improvements gained from using Ray-Strips are highly dependent on finding sufficiently long triangle strips to build the hierarchy on. Obviously, this makes our approach unsuitable for models without any mesh connectivity (e.g. without any shared vertices). In that case, the Ray-Strip representation shown in Section 4 becomes an indexed triangle list with a standard BVH (just with the addition on the 2-byte header storing the triangle count) and performance gains are lost. However, the memory overhead added is only very small. Therefore, it is unlikely that there will be a significant performance loss compared to using an indexed triangle list to start with.

Our current implementation uses the stripification library Stripe [12], which was designed for rasterization and unlike many newer approaches works on general meshes without limitations on the input. Since the computed strips can be sub-optimal for ray tracing, it should be quite possible to further improve the performance of our approaches by designing a stripification algorithm that chooses strips based on ray tracing criteria.

As presented in Section 3 and 4, our system uses a BVH with axis-aligned bounding boxes as the high-level hierarchy, but is important to note that in principle any acceleration structure can be used for the Ray-Strips. We find that a BVH usually provides a reasonable compromise between rendering speed, flexibility and ease of use. It is also easily updateable so that dynamic scenes can be handled efficiently. If maximum performance for a static scene is desired, a kd-tree may be a better choice and might reduce the memory footprint slightly. Note that Ray-Strips can be updated in the same manner as BVHs, but have the limitation that mesh connectivity cannot change in the animation, e.g. objects cannot "break".

7 Future Work and Conclusion

We have proposed a novel compact representation, Ray-Strip, for ray tracing triangular meshes. By using our representation, we are able to reduce up to 80% of the memory footprint over prior approaches. We are also able to obtain the performance improvement on ray tracing of large data sets due to the reduced memory requirement, increased SIMD utilization, and more coherent memory access compared to a standard acceleration structure. This makes our representation an attractive candidate for future ray tracing systems and hardware.

There are many interesting issues for future work. It would be useful to extend the mesh representation to include geometry compression and hierarchy compression, which can further lower the memory overhead. Another promising avenue is integration with a LOD technique such as R-LODs [41] for out-of-core rendering of more complex models. We are also interested in investigating stripification algorithms that are optimized towards generating strips suitable for ray tracing with Ray-Strips. Finally, we plan to investigate the effects of using Ray-Strips on other data parallel architectures such as GPUs.

Acknowledgments

The Bunny, Buddha, Lucy and Thai models are courtesy of the 3D scanning repository at Stanford University. 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 a KAIST seed grant.

References

[1] P. Alliez and C. Gotsman. *Recent advances in compression of 3D meshes*, pages 3–26. Springer, 2005.

[2] John Amanatides and Kia Choi. Ray tracing triangular meshes. In *Proceedings of the Eighth Western Computer Graphics Symposium*, pages 43–52, 1997.

[3] Carsten Benthin. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Computer Graphics Group, Saarland University, 2005.

[4] Carsten Benthin, Ingo Wald, and Philipp Slusallek. Interactive ray tracing of free-form surfaces. In *AFRIGRAPH 2004*, pages 99–106, 2004.

[5] Yi-Jen Chiang, Jihad El-Sana, Peter Lindstrom, Renato Pajarola, and Cláudio T. Silva. Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization 2003 Course Notes*, 2003.

[6] Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching

Model	Tris	Construction time (ms)		Avg. strip length (triangles)	#strips
		Ray-Strip (s)	BVH (s)		
Bunny	67k	0.4	0.03	19.29	4260
Buddha	1.2M	5.5	0.7	29.58	46591
Thai	10M	48.1	7.1	24.95	502577
Iso-surface	10M	49.3	6.3	24.69	519468
Power plant	12.7M	68.9	29.6	7.75	1890760
Lucy	30M	1764	149.6	31.01	1122074

Table 5: **Results: Construction statistics** *The construction time for both plain BVHs (using surface-area heuristic splits) as well as for Ray-Strips is shown. Our approach is much slower here, which is mainly due to the stripification process. The timings for Lucy are particularly slow because the model does not fit into memory during construction. We also show the number of strips and the average strip length. Note that for CAD datasets, the stripification algorithm results in shorter strips due to limited connectivity.*

for distribution ray tracing in complex scenes. *j-CGF*, 22(3):543–552, September 2003.

[7] David Cline, Kevin Steele, and Parris K. Egbert. Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools: JGT*, 2006.

[8] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *IEEE Symp. PVG*, pages 1–8, 2003.

[9] Michael F. Deering. Geometry compression. In *ACM SIGGRAPH*, pages 13–20, 1995.

[10] Pablo Diaz-Gutierrez, Anusheel Bhushan, M. Gopi, and Renato Pajarola. Constrained strip generation and management for efficient interactive 3d rendering. In *Computer Graphics International*, pages 115–121, 2005.

[11] Regina Estkowski, Joseph S. B. Mitchell, and Xinyu Xiang. Optimal decomposition of polygonal models into triangle strips. In *Symp. on Computational geometry*, pages 254–263, 2002.

[12] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 319–326, 1996.

[13] Eric Galin and Samir Akkouche. Fast processing of triangle meshes using triangle fans. In *Proc. of Shape Modeling and Applications(SMI)*, pages 328–333, 2005.

[14] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph.*, 24(3):878–885, 2005.

[15] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[16] V. Havran, R. Herzog, and H. P. Seidel. On Fast Construction of Spatial Hierarchies for Ray Tracing. *IEEE Symposium on Interactive Ray Tracing*, pages 71–80, 2006.

[17] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[18] H. Hoppe. Optimization of mesh locality for transparent vertex caching. *ACM SIGGRAPH*, pages 269–276, 1999.

[19] Z. Karni, A. Bogomjakov, and C. Gotsman. Efficient compression and rendering of multi-resolution meshes. In *IEEE Visualization*, pages 347–354, 2002.

[20] Leif P. Kobbelt, Katja Daubert, and Hans-Peter Seidel. Ray tracing of subdivision surfaces. In *Rendering Techniques*, pages 69–80, 1998.

[21] C. Lauterbach, S. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. *IEEE Symposium on Interactive Ray Tracing*, 2006.

[22] Peter Lindstrom and V. Pascucci. Visualization of large terrains made easy. *IEEE Visualization*, pages 363–370, 2001.

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

[24] Jeffrey Mahovsky. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, September 2005.

[25] Kerstin Müller, Torsten Techmann, and Dieter W. Fellner. Adaptive ray tracing of subdivision surfaces. *Comput. Graph. Forum*, 22(3):553–562, 2003.

[26] D. Nehab and J. Barczakand P. V. Sander. Triangle order optimization for graphics hardware computation culling. *Symposium on Interactive 3D Graphics and Games*, 2006.

[27] BC Ooi, KJ McDonell, and R Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC*, 1987.

[28] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH*, pages 101–108, 1997.

[29] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[30] Hans Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[31] Ken Shoemake. Pluecker coordinate tutorial. *Ray Tracing News*, 11(1), 1998.

[32] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Technical Report TR-06-21, Dept. of CS, Univ. of Texas, 2006.

[33] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[34] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In *EUROGRAPHICS*, volume 20, pages 153–164, 2001.

[35] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 2006.

[36] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering*, 2004.

[37] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *Proc. of ACM SIGGRAPH*, 2006.

[38] Michael Wand and Wolfgang Straber. Multi-resolution point-sampled raytracing. In *Graphics Interface*, 2003.

[39] Sven Woop, Joerg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

[40] Carsten Wächter and Andreas Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006: Eurographics Symposium on Rendering.*, 2006.

[41] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-LODs: Interactive LOD-based Ray Tracing of Massive Models. *The Visual Computer (Pacific Graphics)*, 2006.

[42] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH*, 2005.

[43] Sungeui Yoon, Sean Curtis, and Dinesh Manocha. Ray tracing dynamic scenes using selective restructuring. *Proc. of Eurographics Symposium on Rendering*, 2007.