# Memory Efficient Ray Tracing with Hierarchical Mesh Quantization

Benjamin Segovia* Intel Labs

Manfred Ernst† Intel Labs

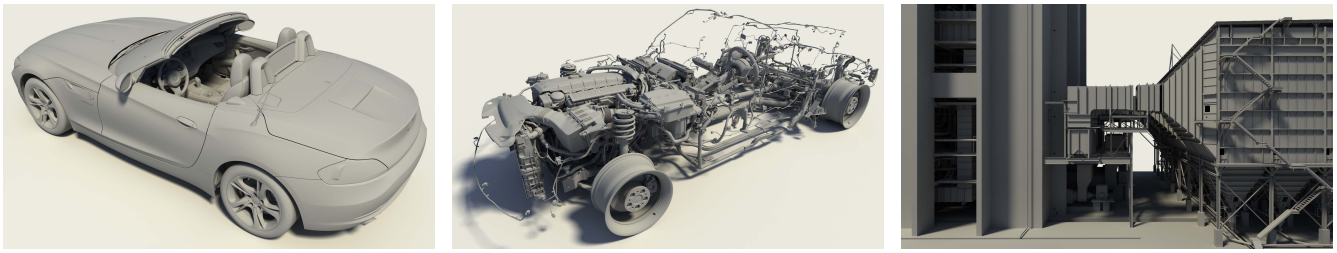


Figure 1: Three images rendered with path tracing and hierarchical mesh quantization. Left: car model (22 M triangles). Middle: mechanical and electrical parts of a car (13 M triangles). Right: Close-up of the power plant (13 M triangles). Hierarchical mesh quantization achieves an average compression rate of 5.7 : 1 in comparison to a BVH and an indexed face set. Path tracing with the compressed acceleration structure is only 17% slower than using a fast uncompressed representation. With ray packets, the difference is even smaller: only 11% on average.

## Abstract

We present a lossily compressed acceleration structure for ray tracing that encodes the bounding volume hierarchy (BVH) and the triangles of a scene together in a single unified data structure. Total memory consumption of our representation is smaller than previous comparable methods by a factor of 1.7 to 4.8, and it achieves performance similar to the fastest uncompressed data structures. We store quantized vertex positions as local offsets to the leaf bounding box planes and encode them in bit strings. Triangle connectivity is represented as a sequence of strips inside the leaf nodes. The BVH is stored in a compact quantized format. We describe techniques for efficient implementation using register SIMD instructions (SSE). Hierarchical mesh quantization (HMQ) with 16 bits of accuracy achieves an average compression rate of 5.7 : 1 in comparison to a BVH and an indexed face set. The performance impact is only 11 percent for packet tracing and 17 percent for single ray path tracing on average.

**Index Terms:** Computer Graphics [I.3.6]: Methodology and Techniques—Graphics data structures and data types; Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Raytracing; Coding and Information Theory [E.4]: Data compaction and compression—

## 1 Introduction

Ray tracing is becoming increasingly important for both off-line rendering and interactive applications. Recent work has shown that real-time ray tracing is feasible on both CPUs and GPUs, with performance sufficient for applications such as visualization of prototypes in computer aided design. However, memory capacity often places significant constraints on the size of models that can be rendered.

One possible solution to this issue is to render directly from a compressed representation of the model. However, most compression algorithms have limitations that make them unsuitable for this purpose. For example, they are too slow, or they do not permit nearly-random access to the geometry data, as is required in many ray tracing scenarios. Recently developed algorithms have begun to overcome these limitations, but they either achieve only moderate compression rates [11] or they require blockwise decompression of the data structures with caching [9].

We present a novel compressed ray tracing acceleration structure that is more compact than all previous comparable methods, and achieves performance similar to the fastest known uncompressed representations. A complete car model consisting of 22 million triangles and its BVH fit into 128 MB of memory and can be rendered at real-time frame rates in full HD resolution on a single CPU with four cores (primary rays only). The most important conceptual differences between this technique and previous ones are that:

1. It encodes the bounding volume hierarchy and the triangle data together in a single unified data structure.
2. Vertex positions are quantized locally in the BVH leaves with global snapping and stored as bit strings.
3. Triangle connectivity is encoded with tabulated strips.

In addition, the data structure is designed to permit highly efficient on-the-fly decompression using certain capabilities of register SIMD instruction sets such as Intel's SSE.

We evaluate the method with SIMD packet tracing and with single ray path tracing. The performance impact is very low for both rendering methods: on average 11 percent and 17 percent, respectively. Note that these values are in comparison to a highly optimized *in-core* ray tracer, with the uncompressed data sets fitting into main memory.

## 2 Background

Ray tracing of large data sets has been an active area of research for several years. Early approaches used out-of-core techniques to handle geometry that does not fit into main memory. Pharr et al. [13] streamed the rays over the geometry sets, which requires ray reordering. Wald et al. [22] used sophisticated caching techniques to achieve interactive ray tracing performance for models that do not fit into main memory. Level-of-detail (LOD) methods have also been applied to ray tracing to allow for rendering of complex scenes [23, 2, 18].

*e-mail: benjamin.segovia@intel.com

†e-mail: manfred.ernst@intel.com

Graphics Interface Conference 2010
31 May - 2 June, Ottawa, Ontario, Canada

Mahovsky reduced the memory consumption of bounding volume hierarchies with quantization of the bounding box geometry [12]. The triangle meshes are not compressed in any way. Cline et al. used a similar approach and extended it with wider fan-outs and a pointer free memory layout of the hierarchy [3]. Our approach also uses quantization for the bounding boxes but we store the nodes in a more flexible block-wise allocation scheme and we compress the triangle data. Reshetov introduced vertex culling [14] to handle large leaf nodes in shallow trees more efficiently.

Hierarchical quantization of point clouds was realized with bounding sphere hierarchies by Rusinkiewicz and Levoy [15] and with kd-trees by Hubo et al. [5]. In both approaches, the geometry of a child node is quantized locally in the coordinate system of its parent node. In this way, the quantization error is not propagated down to the leaf nodes and precision is increased. Unfortunately, this approach cannot be applied to the vertices of a triangle mesh, because it would result in gaps between adjacent triangles that are stored in different leaf nodes. We solve this problem with the global snapping of vertices and bounding boxes.

Lauterbach et al. presented Ray Strips and ReduceM [10, 11], two hierarchical acceleration structures that store triangle strips in the leaf nodes. The vertices of the strips implicitly define a bounding interval hierarchy [19], but their positions are not compressed in any way.

Aggressive compression algorithms have been developed to encode tree structures but none of them allows for random access to the hierarchy nodes. We refer the interested reader to Katajainen and Mäkinen's survey [8]. The situation is quite similar for triangle meshes. Most methods are optimized for high compression rates to reduce storage size on disk or during network transfer. Alliez and Gotsman presented a comprehensive survey on this topic [1].

Recently, Yoon et al. introduced random accessible compressed meshes [24] and bounding volume hierarchies [9]. The mesh compression is based on Isenburg's algorithms [6]. It uses entropy coding to compress the residuals of predicted and quantized vertex positions. Very high compression rates are achieved with this approach but random access is only simulated with caching of large decompressed clusters. The algorithm for bounding volume hierarchies is similar in spirit. It predicts the geometry of the child bounding boxes from the parent node and compresses the residuals, in this case with a dictionary coder. The method is also not suitable for random access to individual BVH nodes. Detailed comparisons to RACBVH and to ReduceM are presented in Section 7.

## 3 Overview

Our method stores a triangle mesh and its bounding volume hierarchy in one compact hierarchical data structure. We build our method around three main components:

**Local vertex quantization with global snapping** (Section 4). Given a BVH leaf node and a set of enclosed triangles, we express the triangle vertices as *local* offsets to the bounding box planes. After quantization, these values can be stored with a small number of bits. Naïve quantization of the vertex positions inside the leaf nodes would, however, result in gaps between adjacent triangles, if they are located in different leaves. We solve this problem with a *global* quantization grid (see Figure 2). All leaf nodes are aligned on this grid. The triangle vertices are snapped to the same grid and their positions are stored as an integer offset to the closest bounding box plane for each axis.

**Tabulated triangle strip sets** (Section 5). In addition to the vertex positions, it is also crucial to compress the vertex connectivity. We store the triangles locally inside the BVH nodes as strips or indexed strips, whichever is smaller. As many strip configurations are possible inside the leaves, we use a precomputed table which enumerates and describes all possible leaf layouts. The vertex indices, like the quantized positions, finally have a small range and can be compressed very well.

**Quantized BVH with block allocation** (Section 6). The bounding volume hierarchy is stored in a compact format. We follow Mahovsky's [12] approach and quantize the bounding boxes of the BVH nodes. In addition, the nodes are stored in clusters. In this way, the child pointers represent a local offset in the cluster that requires only a small number of bits. Overall, the size of a BVH node is reduced from 32 bytes to 4 bytes. To limit the performance impact due to bounding box quantization, we store the top nodes of the BVH in an uncompressed format. As our data layout implies no restrictions on the construction step, we use an SAH-based builder as described in [20] since it provides high quality trees.

## 4 Local Vertex Quantization with Global Snapping

Although simple, the hierarchical vertex encoding is the crucial part of our method. It actually relies on four points:

1. All geometric data is quantized to make it amenable for compression.
2. As the BVH is an approximation of the mesh, we store the vertices as local offsets to the bounding box planes of the leaf nodes and encode them in bit-strings.
3. Cracks in the geometry are prevented by snapping all vertices *and* leaf bounding boxes to the same global grid (see Figure 2).
4. The globally snapped bounding boxes need not be stored, because we can (re-)snap them on-the-fly while decompressing the vertex positions.

Therefore, given a bounding box with origin $\mathbf{O}$ and a set of $k$ enclosed triangle vertices $\mathbf{A_i}$, $i \in [1 \dots k]$ to encode, we first compute the integer delta vectors $\mathbf{D_i} = (x_i, y_i, z_i)$ from $\mathbf{O}$ to $\mathbf{A_i}$. Then, we compute the numbers of bits $n_x$, $n_y$ and $n_z$ required to encode $\max(x_i)$, $\max(y_i)$, $\max(z_i)$. Finally, $n = n_x + n_y + n_z$ is the number of bits required for each vertex. This leads to the following variable bit encoding scheme:

- Encode in the bit string the numbers $n_x$, $n_y$, and $n_z$ of bits required per vertex for each axis.
- Encode the quantized positions of the $k$ vertices in the bit string afterward.

Decoding the vertex data is then straightforward since it mostly consists in reading the number of bits required for each component and to decode the vertex data. An efficient implementation, however, is tricky. Decoding a bit string requires to handle the common case where the vertices to decode overlap several aligned quad-words. Usually, this requires many expensive logical operations and branches. Our implementation makes use of unaligned loads and SSE instructions to overcome these issues. A detailed description is given in Appendix A.

### 4.1 Min-Max Trick

The basic technique to encode the vertex data inside a bounding box consists in computing the integer delta vector from the bottom-left corner $(x_{min}, y_{min}, z_{min})$. However, the leaf bounding boxes have a crucial property: they are the *actual* bounding boxes of the vertices they contain. The vertex positions are not uniformly distributed inside the volume of the bounding box. Many vertices are instead

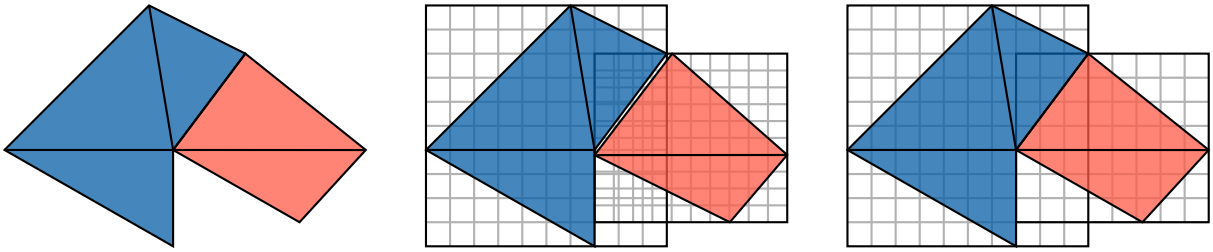


Figure 2: Vertex quantization. Left: unquantized vertices. Center: Local quantization in leaf nodes without global snapping; Gaps appear between adjacent triangles. Right: Local quantization with snapping to a global grid results in a watertight mesh.

close to the bounding box faces. Computing the delta values from the *closest* plane therefore leads to much smaller delta vectors (see Figure 3).

For this reason, we add one extra bit for each component of each delta vector to refer the closest plane from which the distance is expressed. Although we have to store three extra bits per vertex, exploiting the non-uniform distribution of the vertices reduces the final size of the data structure by 5 percent on average. It works particularly well for scenes with large triangles, such as the Power Plant.

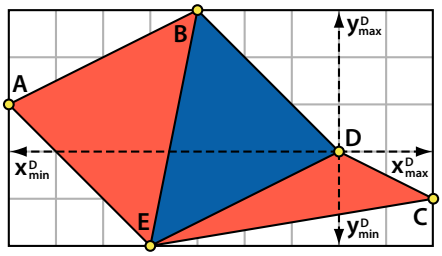



Figure 3: Min-max trick. Instead of only using the "minimum" bounding box planes, we propose to add one more bit to choose, for *each* dimension, the closest plane. In this 2D case, we will therefore encode point **D** as $(x^D_{max}, y^D_{min})$

### 4.2 Index Encoding

We also propose to use a variable bit rate for vertex indices. As we did with vertex data, we first compute the number of bits required to encode the largest index. Then, using this number of bits, we store all the indices.

## 5 Tabulated Triangle Strip Sets

Another important question is how to handle the triangle connectivity. Strips are a compact representation for triangle meshes. However, generating a single non-degenerated strip per leaf is often impossible. Therefore we designed an extended strip system that can handle a *sequence* of strips for each leaf node. Instead of storing the length and the other properties of the strips, we keep all the data in an automatically generated look-up table (LUT) and only store an index into this table for each leaf. The table is designed as follows:

- Set a maximum triangle number per leaf (in our case 14).
- Each entry of the LUT handles one mesh layout. Entry STRIP3×3_STRIP2×3_TRI×4 will for example handle a layout consisting of 3 strips of 3 triangles then 3 strips of 2 triangles and then 4 triangles.
- To limit the number of entries, *sort* the strips according to their sizes. Therefore, STRIP4×1_STRIP3×3 is a valid entry while STRIP3×3_STRIP4×1 is not.
- For each entry, we store the number of triangles and the offsets of their first vertex. For example, Entry STRIP3×1_TRI×2 is composed by five triangles and the five triangle offsets {0,1,2,5,8}.
- Since strips alternate clockwise and counter-clockwise triangles, store each triangle orientation.

This strategy leads to an efficient, compact and versatile way to handle any sequence of strips. Using at most 14 triangles per leaf provides a surprisingly small number of entries: in this case, 507. Therefore, using 9 bits is sufficient to index any leaf layout containing from 1 to 14 triangles.

In addition, the LUT stays the same whether we are using strips or indexed strips. If the leaf stores triangle strips, the offsets provided by the table give the offsets of the vertex *positions*. Conversely, if the leaf stores indexed strips, the table will provide the offsets of the vertex *indices*. This is the core of the last optimization. During the compression, we compute strip and indexed strip sequences and keep the smaller one. Usually, indexed strips are more compact for a large number of triangles. The leaves are then flagged with an additional bit to indicate the presence of strips or indexed strips.

We also experimented with triangle soup and local indexed face sets instead of triangle strips in the leaf nodes. On average, the resulting data structures were 26 percent and 8 percent larger, respectively. Performance was almost identical for all three methods.

## 6 Quantized BVH with Block Allocation

Many techniques may be used to reduce the size of the BVH nodes and the overall size of the tree. Some methods like lightweight BVH [3] require specific hierarchy layouts such as balanced trees to completely remove child indices. This unfortunately impacts the final ray tracing performance because the SAH cannot be used. Other techniques reduce the memory size of the bounding volumes using either fixed rate quantization or hierarchical encoding schemes like in [12]. These techniques unfortunately suffer from the relative sizes of the child indices. To achieve a good trade-off between speed penalty due to changes in the tree hierarchy and size penalty due to children indices, we base our method on three techniques:

1. A two-level BVH which uses both compressed and uncompressed nodes.
2. Hierarchical quantization to reduce the bounding box memory consumption.
3. Block allocation and block-relative indices to reduce the size of all references to child nodes.

### 6.1 Hierarchical Encoding

Following Mahovsky's approach [12], we express the minimum and maximum positions of the child bounding boxes relatively to their parents. Since we only use 4 bits to encode each bounding box

component, the bounding box memory footprint is equal to 24 bits. Using 4 bytes per node finally offers 8 bits to index the child nodes and to encode extra information as described in the next section.

As shown in Appendix B, we also designed a fast decoding routine using both look-up tables to convert the packed min/max components and the SSE4.1 _mm_insert_epi32 instruction (which allows the insertion of a single element inside a SIMD vector) to enable efficient SIMD computations. Compared to the sole use of look-up tables as described in [12], we obtained a 2× speed-up for this decoding step.

### 6.2 Block Indexing

To index the child nodes with at most 8 bits, we cluster nodes into blocks and use two interleaved indexing systems. *Inside* a block, only a small number of bits indexes the children. In the example of Figure 4, the small indices are used to index nodes N from root nodes R. To refer a child *outside* the current block, we store a 32 bit offset in a dedicated forward node. In Figure 4, forward nodes F point to block roots R while they are themselves pointed to by bottom nodes B.

Using this layout, we built a *4 byte* node format such that 24 bits are used for the bounding box (see Appendix B) and 6 bits are used to index the child nodes. The remaining 2 bits distinguish leaf nodes and the three internal node types X, Y, Z, required to sort the two children as described in [21].

### 6.3 Leaf Layout

Leaf layout is also crucial since most of the nodes are actually leaves and using one or several 4 byte integers for triangle or vertex indices will largely increase the size. As shown in Figure 4, our idea consists in interleaving leaf data and tree nodes to avoid references to external arrays. By simply storing the leaf data after each block, we can refer the leaf data relatively to the location of the current leaf in memory.

### 6.4 Two-Level BVH

According to [12], hierarchically quantizing the BVH nodes leads to a significant overhead and a performance drop while using single rays or small ray packets. A simple improvement actually consists in storing the top of the tree in an uncompressed format and the bottom nodes in a compressed one. As top nodes are traversed by most of the rays, this limits the performance impact while maintaining a small memory footprint.

### 6.5 Alternative Layouts

We tried other BVH layouts. In particular, we implemented a one-level BVH using 8 byte nodes and 48 bit bounding boxes. However, we found that the best trade-off regarding size and speed was obtained with a two-level BVH using 4 byte nodes. Indeed, in all cases, for identical sizes, two-level BVHs with 4 byte nodes were faster. Conversely, for equivalent ray tracing performance, two-level BVHs with 4 byte nodes were smaller. In the rest of the paper, we will use 4 byte two-level BVHs letting 5% of the top nodes uncompressed. The resulting data structure is only 6–11 percent larger than a fully compressed hierarchy, but ray tracing performance is improved by 12–24 percent.

We may also notice that all the ideas presented here are generic. Therefore, they may be applied to other hierarchy layouts like MBVH [4] or ReduceM [11].

## 7 Results and Discussion

We evaluated our method on an Intel® Core™ i7 Extreme CPU (3.2 GHz) with 6 GB of main memory running 64-bit Linux. Two algorithms were implemented: a state-of-the-art packet tracer for real-time rendering [21] and a single ray path tracer [7] for offline rendering.

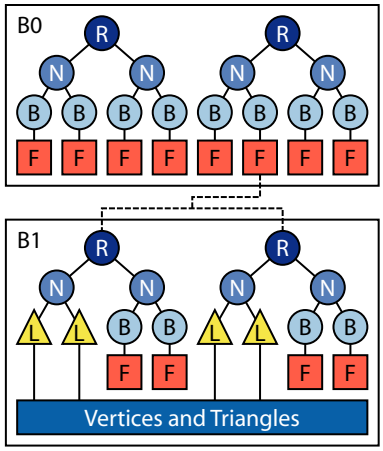



Figure 4: BVH block layout. In this simple example, we allocate the BVH using two blocks B0 and B1. In block B0, we have regular inner nodes (N) and bottom nodes (B). Bottom nodes indirectly point to "child" blocks with the help of "forward" nodes (F). Block B1 is a child block of B0. It contains leaves (L) with vertices and triangles enqueued at its end. Our structure directly interleaves primitives, bounding boxes and child indices.

All timings are given for eight threads running on four cores of the CPU. In comparison to a single threaded implementation, our algorithm scales almost linearly with the core count. Actual numbers are omitted for the sake of brevity. Memory consumption is given for the entire data structure including the vertices, triangle strips and the bounding volume hierarchy. The compression rates are measured in comparison to an indexed face set and a binary SAH-based bounding volume hierarchy with 32 byte nodes.

### 7.1 Memory Consumption and Rendering Performance

In comparison to an indexed face set and a regular BVH with a maximum leaf size of 12, our method reduces the memory consumption by a factor of 4.9 to 7.2. Large leaf nodes not only reduce the absolute memory footprint but also improve the compression rates, as can be seen in Table 2. The impact of compression on rendering performance is as low as 4.6 to 14.4 percent for primary ray packets and 10.8 to 15.7 percent for single ray path tracing. The influence of larger leaf nodes is diverse, even resulting in better performance for some cases. See Table 2 for the details.

### 7.2 Compression Time

Our implementation was mainly focused on rendering performance. Therefore, the compression algorithms are not multi-threaded and the compression speed is low, in particular while using strips or indexed strips (SIS). For example, compressing Lucy using quantized indexed face sets takes 7 seconds while compressing it using SIS takes 210 seconds. The current implementation therefore limits our algorithm to static scenes. However, handling large animated scenes is an interesting future work. First, it would be crucial to *stream* the decompression. Indeed, decompressing the whole model in main memory before the animation is not an option since it would require a large amount of available memory on the rendering device. Being able to apply the animation directly on the compressed mesh is an appealing challenge.

Secondly, as for the BVH refitting algorithm [21], it should be noticed that the topology may stay unchanged during an animation and therefore stripping algorithms are not required anymore. However, as the vertex data size may change, the implementation of a compressed BVH refitting technique is more difficult since it cannot operate in-place.

### 7.3 Impact of Quantization Parameters

Quantization always introduces a certain amount of error to the geometry. This is acceptable as long as the error is controllable and can be adjusted such that the final images are indistinguishable from a reference image. Depending on the input data, our implementation allows the user to either specify the number of quantization bits $q$ or an upper limit for the quantization error $\varepsilon$. The latter is particularly useful for tessellated CAD models where $\varepsilon$ can be chosen to match the tessellation accuracy. A visual comparison of different quantization parameters and the corresponding size of the data structure is given for the car model in Figure 5 and for Lucy in Figure 6. For all other measurements in this paper, we use $q = 16$ for Lucy and the power plant. The CAD models are quantized with $\varepsilon = 12.5\mu m$ which corresponds to roughly 16-18 quantization bits per axis on the global grid. For these parameters, the final renderings are almost identical to a reference image.

### 7.4 Comparison with Higher Order Primitives

Higher order primitives, such as NURBS or subdivision surfaces, are an alternative to compression of triangulated models in some scenarios. It is, however, a common misconception that higher order surfaces are always more compact. In the case of CAD models, the NURBS representation typically consumes *more* memory than the tessellated models. Direct ray tracing of trimmed patches is also more expensive, in particular for incoherent rays. The only reason why NURBS are attractive for rendering of CAD models is their arbitrary accuracy.

The situation is different for subdivision surfaces. The control mesh is always more compact than the refined mesh. In this situation, a direct visualization is a very attractive way to reduce the memory footprint and also allows per-pixel adaptive tessellations. The same is true for displacement maps. A one-byte-per-pixel displacement map may represent a mesh with about one byte per triangle, while our technique requires between 4.5 and 6.6 bytes per triangle. However, our representation can be directly intersected and is likely to perform better, in particular while using single ray tracing or path tracing techniques. For such cases, it is more difficult to factorize the tessellation steps if done on-the-fly or the cache accesses if triangle caches are used. Christensen et al. [2] describe techniques to handle the problem.

For these reasons, we believe that a compact and intersectable representation of a tessellated mesh and higher order primitives remain complementary.

### 7.5 Comparison to RACBVH [9]

Both, our technique and the RACBVH use uniform quantization for the vertex positions, but contrary to RACBVH we compress the BVH and the triangle mesh *together*. This allows us to achieve similar compression rates with less aggressive but more efficient encoding / decoding algorithms. Kim et al. report an average compression rate of 10:1 when comparing to a BVH and triangle soup. We achieve 8:1 in this case.

Although their compression rates are higher, we still achieve a significantly lower total memory consumption. Indeed, RACBVH does not support BVHS with more than one triangle reference per leaf node whereas our technique does. For Lucy, our representation only requires 159 megabytes of memory while RACBVH consumes 266 megabytes with the same quantization accuracy of 16 bits per axis. For larger leaf nodes, Kim et al. have to resort to a global index list for the triangle references but the complete memory requirements for the combination of RACBVH and an index list are unfortunately not given in the paper. We would, however, expect that this number is larger than our total memory consumption.

The performance is very difficult to compare, because we compare our compressed representation to a state-of-the art ray tracer with all data sets fitting into main memory. Kim et al. only present performance data for a mesh that does not fit into memory.

We may also note that RACBVH requires decompressing data to a cache in large clusters. This approach can lead to cache thrashing with very incoherent rays from a path tracer. Our method has no such limitations because all data is decompressed on the fly. This is also beneficial for parallel scalability. Our method scales linearly with the number of processor cores whereas the RACBVH only achieves a speed-up of $2\times$ when switching from one to four cores.

| | Our technique | RACBVH | ReduceM |
|---|---|---|---|
| Lucy | 159 MB | 266 MB | 568 MB |
| Power Plant | 57 MB | 121 MB | 272 MB |

Table 1: Size comparisons with RACBVH and ReduceM. Our hierarchical encoding scheme improves compression rates by a factor of up to $4.8\times$. Note that we present here the smallest BVHs using at most 12 triangle per leaf since performance changes were not significant (see Table 2).

### 7.6 Comparison to ReduceM [11]

Our technique is similar to ReduceM in two ways: both methods operate in-core and they use a two-level approach to store the hierarchy. In ReduceM, the top-level may be a kd-tree or any other acceleration structure. Lower levels are strips which implicitly define a balanced acceleration structure for ray tracing.

As ReduceM authors chose to use plain vertex data, this makes ReduceM more suitable for animation. Adding quantization to ReduceM should be relatively easy. As the acceleration structure at the bottom levels is implicitly defined by vertex data, traversing the strips would however require to sequentially uncompress quantized data for each traversal step. This strategy is probably much more expensive than ours (where vertex data decoding only occurs in leaves), but the idea is appealing for future work.

Regarding performance, ReduceM is competitive to a highly optimized kd-tree. Our method's performance is similar to a state-of-the-art BVH. We think that both methods are more or less equivalent with regard to performance.

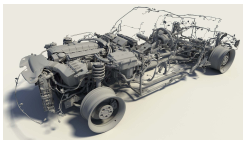
Table 1 shows the memory footprints of ReduceM, the RACBVH and our method. Our hierarchical quantization system clearly outperforms the other approaches. As a consequence, it is the best method when memory consumption is the major bottleneck.

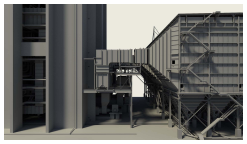
### 7.7 Encoding Other Attributes

In this paper, we only focused on the compression of vertex positions. We believe that the technique can be easily extended to support other attributes as well. Shading normals could be added by encoding one normal for the first vertex in a leaf node and expressing the remaining normals as delta vectors to the first one. The same approach should work for texture coordinates and vertex colors. Material IDs are a bit different, but we can expect that most leaves use only one material and handle the multi-material situation as a special case.

## 8 Conclusion

We have presented a compressed acceleration structure for ray tracing that combines a triangle mesh and its BVH in one unified data structure. The method quantizes vertex positions locally in the leaf nodes and encodes them as bit-strings. Vertex connectivity is represented by tabulated triangle strip sets. The BVH is stored in a clustered layout with quantized bounding boxes, requiring only 4 bytes per node. With 16 bits of quantization accuracy for the vertices, we achieve total compression rates between $4.9\times$ and $7.2\times$, when comparing to a BVH and an indexed face set. The final memory footprint is significantly lower than that of any comparable method.



| | Car | | | Engine | | | Lucy | | | Power plant | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Mesh** | | | | | | | | | | | | |
| Vertices | 12,458,625 | | | 16,567,597 | | | 14,027,872 | | | 11,070,509 | | |
| Triangles | 22,364,947 | | | 12,763,717 | | | 28,055,742 | | | 12,748,510 | | |
| **BVH** | | | | | | | | | | | | |
| Max leaf size | 4 | 8 | 12 | 4 | 8 | 12 | 4 | 8 | 12 | 4 | 8 | 12 |
| BVH nodes (millions) | 14.1 | 7.6 | 5.1 | 8.3 | 4.4 | 3.0 | 16.8 | 9.1 | 6.1 | 7.5 | 4.0 | 2.9 |
| **Memory (MB)** | | | | | | | | | | | | |
| BVH + IFS | 913.0 | 716.0 | 640.4 | 638.4 | 520.1 | 475.2 | 1101.3 | 865.4 | 773.9 | 548.9 | 442.3 | 409.2 |
| HMQ | 199.8 | 144.9 | 127.8 | 129.8 | 96.0 | 84.5 | 245.5 | 181.3 | 158.5 | 98.1 | 67.9 | 57.0 |
| Ratio | 4.6 : 1 | 4.9 : 1 | 5.0 : 1 | 4.9 : 1 | 5.4 : 1 | 5.6 : 1 | 4.5 : 1 | 4.8 : 1 | 4.9 : 1 | 5.6 : 1 | 6.5 : 1 | 7.2 : 1 |
| **Path tracing (Mrays/s)** | | | | | | | | | | | | |
| BVH + IFS | 3.2 | 3.1 | 2.9 | 3.8 | 3.6 | 3.3 | 8.4 | 8.0 | 7.0 | 2.2 | 2.2 | 2.1 |
| HMQ | 2.6 | 2.5 | 2.5 | 2.7 | 2.8 | 2.9 | 7.5 | 7.0 | 6.3 | 1.8 | 1.9 | 1.8 |
| Slow-down | 18.7% | 18.8% | 15.7% | 29.0% | 22.5% | 14.1% | 10.2% | 12.3% | 10.8% | 19.3% | 14.3% | 12.9% |
| **Packet tracing (Mrays/s)** | | | | | | | | | | | | |
| BVH + IFS | 26.5 | 27.4 | 26.2 | 21.6 | 23.0 | 23.2 | 12.8 | 13.9 | 13.9 | 33.8 | 32.6 | 31.0 |
| HMQ | 23.9 | 24.6 | 24.1 | 18.6 | 19.5 | 19.9 | 11.1 | 12.1 | 12.4 | 30.9 | 29.3 | 29.6 |
| Slow-down | 9.8% | 10.3% | 8.2% | 14.0% | 15.4% | 14.4% | 13.1% | 12.9% | 10.8% | 8.6% | 9.9% | 4.6% |

Table 2: Results for total memory footprints (megabytes), path tracing and packet tracing speed (millions of rays per second). BVH + IFS: uncompressed binary BVH and indexed face set. HMQ: hierarchical mesh quantization.

In addition, the performance impact compared to an in-core real-time ray tracer is only 4.6 to 15.7 percent, resulting in the fastest ray tracing performance for large models.

In the future, we are planning to implement the algorithm on Intel's Larrabee architecture [17] since it provides a practical instruction set for our technique. Porting our method to OpenCL is also interesting for future work but may require different technical choices to be efficient.

## Acknowledgements

The authors wish to thank BMW, the Stanford Computer Graphics Laboratory and the Walkthru Group at UNC for providing the models. We also thank Carsten Benthin, Bill Mark, Alexander Reshetov and Ingo Wald for fruitful discussions and the anonymous reviewers for their valuable comments.

## References

[1] P. Alliez and C. Gotsman. Recent Advances in Compression of 3D Meshes. In *Advances in Multiresolution for Geometric Modeling*, pages 3–26. Springer, 2004.

[2] P. H. Christensen, D. M. Laur, J. Fong, W. L. Wooten, and D. Batali. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. In *Proceedings of Eurographics 2003*, pages 543–552, 2003.

[3] D. Cline, K. Steele, and P. Egbert. Lightweight Bounding Volumes for Ray Tracing. *Journal of Graphics Tools*, 11(4):61–71, 2006.

[4] M. Ernst and G. Greiner. Multi Bounding Volume Hierarchies. In *Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing 2008*, pages 35–40, 2008.

[5] E. Hubo, T. Mertens, T. Haber, and P. Bekaert. The Quantized kd-Tree: Efficient Ray Tracing of Compressed Point Clouds. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006*, pages 105–113, 2006.

[6] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming Compression of Triangle Meshes. In *Procedings of the 2005 Symposium on Geometry Processing*, 2005.

[7] J. T. Kajiya. The rendering equation. In *Proceedings of SIGGRAPH '86*, pages 143–150, 1986.

[8] J. Katajainen and E. Mäkinen. Tree Compression and Optimization with Applications. *International Journal of Foundations of Computer Science*, 1(4):425–447, 1990.

[9] T. Kim, B. Moon, D. Kim, and S. Yoon. RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 2009.

[10] C. Lauterbach, S. Yoon, and D. Manocha. Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pages 19–26, 2007.

[11] C. Lauterbach, S. Yoon, M. Tang, and D. Manocha. ReduceM: Interactive and Memory Efficient Ray Tracing of Large Models. In *Rendering Techniques 2008*, pages 1313–1322, 2008.

[12] J. A. Mahovsky. *Ray tracing with reduced-precision bounding volume hierarchies*. PhD thesis, University of Calgary, 2005.

[13] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *Proceedings of SIGGRAPH '97*, pages 101–108, 1997.

[14] A. Reshetov. Faster ray packets - triangle intersection through vertex culling. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pages 105–112, 2007.

[15] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of SIGGRAPH '00*, pages 343–352, 2000.

[16] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[17] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions Graphics*, 27(3):1–15, 2008.

[18] G. Stoll, W. R. Mark, P. Djeu, R. Wang, and I. Elhassan. Razor: An architecture for dynamic multiresolution ray tracing. Technical Report 06-21, University of Texas at Austin Department of Computer Sciences, 2005.

[19] C. Waechter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006*, pages 139–149, 2006.

[20] I. Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[21] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes

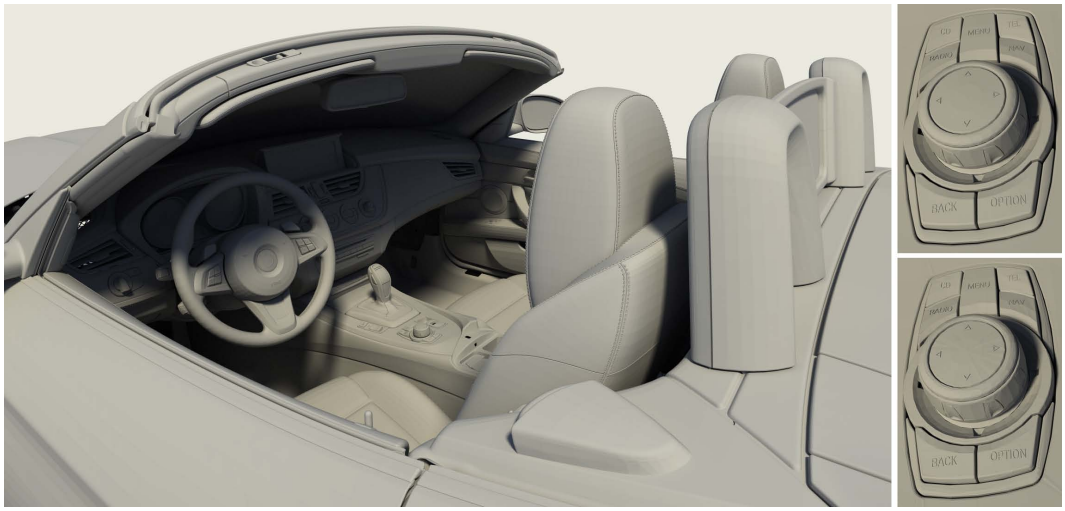


Figure 5: Path traced rendering of a car model with 22 million triangles and 12 million vertices. The close-up renderings on the right side show the effect of quantization on this model. The top image shows a maximum quantization error of 5 $\mu m$. It is indistinguishable from a rendering with the uncompressed floating point data. The close-up below uses a quantization error of 500 $\mu m$ which results in unacceptable artifacts. We use 12.5 $\mu m$ for the large image without any noticeable errors. Note that the quad-like structures are not a compression artifact; they appear because we use flat shading.


using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[22] I. Wald, A. Dietrich, and P. Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Rendering Techniques 2004*, pages 81–92, 2004.

[23] S. Yoon, C. Lauterbach, and D. Manocha. R-LODs: fast LOD-based ray tracing of massive models. In *The Visual Computer (Pacific Graphics) 2006*, pages 772–784, 2006.

[24] S. Yoon and P. Lindstrom. Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1536–1543, 2007.


## A Fast Vertex Data Decoding

To speed up the decoding step, we propose, as shown by Figure 7, to explicitly use the unaligned load capabilities of our target architecture (x86 with SSE support). As x86 chips are particularly lenient regarding memory loads, we will store quad-words aligned to *byte* boundaries in the following manner:

- For a bit number $n$ per vertex, compute the maximum number $p_n = \lfloor 64/n \rfloor$ of vertices that fit into a quad-word.
- Store the vertices using sequences of $p_n$ vertices with padding to the next *byte* boundary between two sequences.

For a faster decoding, we compute a look-up table (LUT) which stores two values for every vertex size $n \in [1 \dots 64]$: The number of vertices $p_n$ we may store inside a quad-word and the offset $s_n$ in *bytes* to the next location of the vertices to decode (including the padding).

The decoding step becomes now simpler and faster. If $q$ is the quad-word from where we are currently decoding:

1. Decode $p_n$ vertices from $q$. As all vertices fit into $q$, this operation only requires a straightforward sequence of SHIFT and AND instructions.
2. Load the next quad-word located $s_n$ bytes after $q$. Note that this is only aligned to a byte boundary, not a quad-word boundary.

To improve the decoding step, we finally use integer SIMD operations offered by SSE for all int-to-float conversions and other operations. Indeed, once we get the delta values from the bit string, a branch-less code using SIMD operations retrieves the integer snapped positions and then computes the world space floating point locations using the origin and the cell size of the snapping grid. The whole pseudo-code of the vertex data decoding is given below.

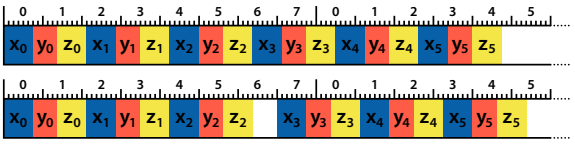



Figure 7: Vertex data decoding. This figure shows two layouts to store six vertices. For both layouts, the number of bits per component inside each leaf is fixed but it may vary from leaf to leaf. In this example, component $x$ (resp. $y$ and $z$) uses 6 bits (resp. 5 and 6 bits). The top layout simply enqueues the vertex bits one after the other. The bottom layout uses the unaligned load capabilities of x86s. In this example, we enqueue 3 vertices and we leave afterward 5 bits blank to align the next set of vertices to a byte boundary (not a quad-word boundary) such that all vertices may be loaded from *unaligned* quad-words. The numbers on the scale indicate byte boundaries.

On the machine used for all measurements (see Section 7), the decoding step of a bit string using scalar instructions takes about 205 cycles per vertex, whereas the decoding step using SIMD instructions and unaligned loads takes about 39 cycles per vertex. This leads to a 5.2× speed-up. As indicated in Section 7, the vertex data decoding overhead is also relatively small compared to the costs of traversal steps and triangle intersections.

```
// AoS and SoA structures with overloaded operators
// 4 integers in a SIMD vector
struct soa_i { ....  __m128i vec; };

// 3d float vector packed in a SIMD vector
struct aos3f { ....  __m128  vec; };

// 3d integer vector
struct vec3i { ....  int x, y, z; };

// LUT which provides for each number of bits (from 0 to 64),
// the maximum number of vertices with this size we may
// decode and the number of bytes required to store
// this number of bits
```


Figure 6: Left: Path traced rendering of Lucy (28 million triangles, 14 million vertices). Close-ups with different quantization parameters: a) un-compressed data, 774 megabytes. b) HMQ with 16-bit quantization, 159 megabytes. c) HMQ with 14-bit quantization, 131 megabytes. d) HMQ with 12-bit quantization, 105 megabytes.

```
static const uint32_t maxToDecode[65][2] =
{ {0, 0}, {64,1}, .... {1, 8}, {1, 8} };

// Return the integer position of a vertex on the given grid
FINLINE soa_i
snap(const aos3f &from, const aos3f &org, const aos3f &rcpD)
{ return ((from - org) * rcpD + aos3f(0.5f)).to_soa_f(); }

// Give world-space position from the location on the grid
FINLINE aos3f
unsnap(const soa_i &from, const aos3f &org, const aos3f &dim)
{ return aos3f(soa_f(from)) * dim + org; }

// Uncompress the vertex data using unaligned loads
void uncompress(
  const uint8_t *data,   // Data to decode
  aos3f *v,              // Uncompressed vertices
  const AABB &aabb,      // Leaf bounding box
  const aos3f &org,      // Origin of the grid
  const aos3f &dim,      // Extents of the grid cells
  const aos3f &rcpDim,   // Reciprocal of dim
  uint32_t vertNum,      // Number of vertices to decode
  const uint32_t bit[3] // Numbers of bits for x,y,z
{
  // Snap the leaf bounding box and get the total number of bits
  // needed per vertex and the bit masks required for x,y,z
  const aos3f pMin(&aabb.pMin);
  const aos3f pMax(&aabb.pMax);
  const soa_i ivMin = snap(pMin, org, rcpDim);
  const soa_i ivMax = snap(pMax, org, rcpDim);
  const uint32_t n = bit[0] + bit[1] + bit[2];
  const uint64_t maskX = (1 << bit[0]) - 1;
  const uint64_t maskY = (1 << bit[1]) - 1;
  const uint64_t maskZ = (1 << bit[2]) - 1;
  uint32_t remaining = vertNum;
  uint32_t whereToGet = 0;

  while(remaining) {
    // Get the number of vertices to uncompress
    const uint32_t currNum = min(remaining, maxToDecode[n][0]);
    uint64_t curr = *(const uint64_t*) (data + whereToGet);

    // Directly uncompress the vertices from the quad-word
    for(uint32_t i = 0; i < currNum; ++i) {
      ALIGN(16) vec3i iDelta;
      iDelta.x = curr & maskX; curr >>= bit[0];
      iDelta.y = curr & maskY; curr >>= bit[1];
      iDelta.z = curr & maskZ; curr >>= bit[2];
      // Apply the min-max trick ...
      const soa_i packed(&iDelta);
      const soa_i which = packed << 31;
      const soa_i delta = packed >> 1;
      const soa_i lMin = ivMin + delta;
      const soa_i lMax = ivMax - delta;
      const soa_i vi = sel(lMin, lMax, which);
      // ... and retrieve the world space position
      *v++ = unsnap(vi, org, dim);
    }
    remaining -= currNum;

    // This is where we have to uncompress the next vertices
    whereToGet += maxToDecode[currNum * n][1];
  }
}
```

## B Fast Compressed Node Decoding

```
// Compressed AABB in 3 bytes. 4 bits are used for each component
struct CAABB {
  void decode(aos3f pMin,  aos3f pMax,
              aos3f &tMin, aos3f &tMax) const;
  // Store per component the min/max value
  uint8_t pMinMax[3];
  // Make the conversion from the 8 bit min/max to floats
  static const float dLUTMinMax[256][2];
};

// Fast gather using SSE4.1 instructions
FINLINE aos3f
aos3f::gather(const float *x, const float *y, const float *z) {
  aos3f to;
  to.vec = _mm_load_ss(x);
  to.vec = _mm_castsi128_ps(_mm_insert_epi32(
    _mm_castps_si128(to.vec), *((int *) y), 1));
  to.vec = _mm_castsi128_ps(_mm_insert_epi32(
    _mm_castps_si128(to.vec), *((int *) z), 2));
  return to;
}

// Fast decompression using SSE hand-coded gather
FINLINE void
CAABB4::decode(aos3f pMin, pMax, aos3f &tMin, aos3f &tMax) const
{
  const aos3f ext = pMax - pMin;
  const float *x = dLUTMinMax[this->pMinMax[0]];
  const float *y = dLUTMinMax[this->pMinMax[1]];
  const float *z = dLUTMinMax[this->pMinMax[2]];
  const aos3f deltaMin = aos3f::gather(x, y, z);
  const aos3f deltaMax = aos3f::gather(x + 1, y + 1, z + 1);
  const aos3f tmpMin = pMin + ext * deltaMin;
  const aos3f tmpMax = pMin + ext * deltaMax;
  tMin = tmpMin;
  tMax = tmpMax;
}
```