

Ray Tracing Massive Models using Hierarchically Compressed Geometry

Ingo Wald

Abstract

We describe a scheme for interactively visualizing massively complex models by interactively ray tracing a hierarchically compressed scene structure that allows for (hierarchically) encoding complex scenes with as few as 6 bytes per triangle, without losing visual quality. In particular, our scheme combines high encoding efficiency with the ability to decode individual nodes and primitives (without having to resort to expensive caching schemes required when en-/decoding entire meshlets) and is therefore well suited for the highly parallel, low-memory accelerator architectures we see in today's GPUs and Xeon Phi coprocessor boards. We demonstrate the power of our scheme by interactively ray tracing (in core!) even the biggest publicly available models on a single Intel Xeon Phi equipped workstation.


1. Introduction

Ray tracing is widely considered advantageous in both its ability to produce highly realistic images, and in its efficient handling of even highly complex geometry. The key to both of these virtues is the use of some form of hierarchical acceleration structure: An acceleration structure not only enables the tracing of arbitrary individual rays (the key to high-quality imagery), it also makes the cost of tracing a ray roughly logarithmic in the number of triangles (the key to efficiency for complex geometry).

The flipside is that ray tracing requires some significant amount of memory: For the model data alone, having to be able to access any part of the model at any time precludes schemes like *level of detail* or *progressive meshes* that in particular for complex geometry have proven so useful for raster graphics. In addition, the acceleration structure requires some non-trivial amount of additional memory, and can easily more than double the total memory footprint. Though exact numbers vary, a usual rule of thumb for total footprint is something on the order of 100 bytes per triangle (and easily more for interactive ray tracers that store additional, per-triangle acceleration data).

This lavish use of memory counteracts ray tracing's advantages for rendering large models: in particular in an interactive setting, paging model data from disk is typically not an option, leading to the quip that "ray tracing can easily render arbitrarily large models...as long as they fit in main memory". As long as ray tracing is only done on CPUs that have full

access to (possibly quite large) main memory that is not too much of a problem: At today's prices, it is not unaffordable to put in enough memory to render even some quite non-trivial models, and that is exactly what most professional users of ray tracing do.



lucy	stmatthew	atlas	david (v2/v3)
28M tris	372M tris	506M tris	921M / 2G tris
189MB	2.2GB	3.0GB	5.3GB / 15.5GB
6.7 bytes/tri	6.0 bytes/tri	6.0 bytes/tri	5.8bpt / 7.5bpt

Figure 1: All of the typical "scanned" models are well handled by our system: Based on 20 bits initial quantization—which yields visually artifact-free results—we can encode these models in around 6 bytes per triangle, meaning that even the biggest publicly available versions of *david*—the 921 million and 2 billion triangle "v2" and "v3" versions—conveniently fit onto a 16GB Intel Xeon Phi accelerator board (the reason *david(v3)* has slightly lower rates because the model is fully cleaned up, and contains lots of "random", disconnected fragments).

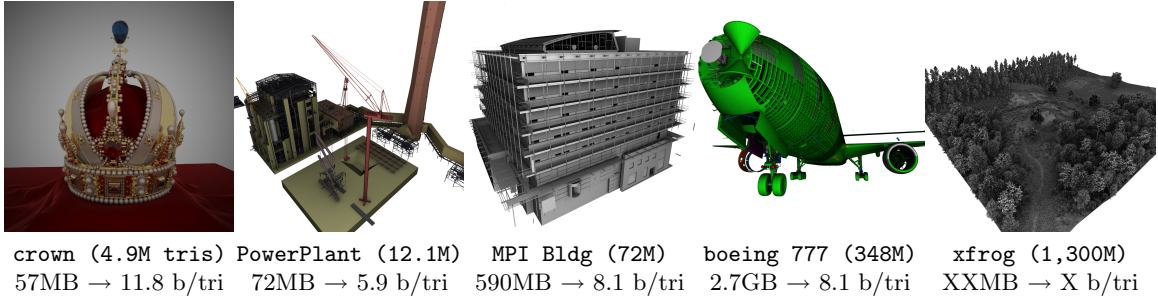


Figure 2: non-scanned models...

Though having to buy lots of memory sounds wasteful even on CPU-based platforms, being limited by the amount of memory becomes particularly problematic in the light of recently available accelerator boards such as GPUs, Intel’s Xeon Phi, or even dedicated ray tracing hardware support (possibly even for mobile devices): accelerators boards and special purpose hardware bear significant potential for faster ray tracing in terms of traversal performance—but as long as they are limited to significantly smaller scenes than what professional users use on CPUs their practical potential is limited.

In this paper, we describe a scheme that hierarchically encodes triangle meshes *and* their associated ray tracing acceleration structure in a way that most models require only 6-8 bytes per triangle without additional vertex data like normals and texture coordinates, and still at 8-12 bytes per triangle if such additional data is present (an overall savings of roughly an order of magnitude compared to uncompressed schemes), without loss of visual quality, and without needing any elaborate paging or caching schemes. Our approach builds on previous work by Mahovski [MW06], Kim et al. [KMKY10], and Segovia et al. [SE10], and combines those systems’ relative strengths. In particular, like Mahovski and Segovia we use a hierarchical encoding of a bounding volume hierarchy that lets us get away with only 32 bits per (Quad-)BVH node, and, like Segovia and Kim also encode the Mesh data. Like Segovia we perform some initial discretization/quantization of the model (with user-controlled accuracy of up to quasi-lossless quantization) and hierarchically encode the resulting data with (much!) fewer bits; but unlike their work retain the ability to maintain vertex sharing, which gives further compression gains in particular for models with per-vertex data. Like Kim we subdivide the model into independently compressed (indexed face set based) meshlets, but do that in a way that allows for decoding individual nodes, primitives, and vertices on the fly, without the need for an elaborate caching scheme

to cache already-decoded meshlets (which would be rather tricky to implement efficiently on accelerator boards with limited memory and hundreds to thousands of parallel threads).

Throughout the paper, we will demonstrate the impact of our individual design decisions and optimizations on a range of different models that include pretty much all publicly available “big” models, including, in particular, both the usually nicely-behaved “scanned” models (see Figure 1 to to the 2 billion triangle variant of *david(v3)*), as well as more engineering style models that often also contain Material information and additional vertex data (Figure 2).

2. Related Work

embree and optix - assume full memory

3. Quantized-Compressed Tri-/Quad-Meshes

The basic premise of our approach is that we can encode both acceleration structure and geometry more efficiently by hierarchically encoding everything relative to some parent BVH node’s bounding box. We logically employ a two-level hierarchy where the model is subdivided into a set of self-contained *meshlets* in which both vertices, primitives and BVH nodes are encoded hierarchically:

Vertex coordinates are encoded as bit strings with just as many bits as necessary to encode the vertex relative to the mesh’s bounding box. Normals and texture coordinates are stored at reduced precision.

BVH nodes are encoded relative to the parent node (and relative to the mesh bounds for the root node) with a total of 32 bits per node (including child pointer etc)

Primitives are encoded as 32-bit integer values whose individual bits can be decoded to yield material ID and up to four vertex IDs.

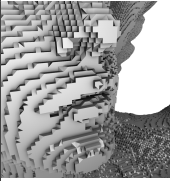
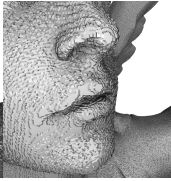

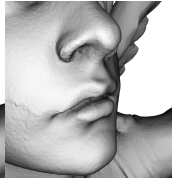
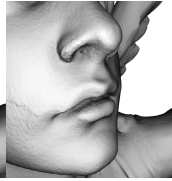
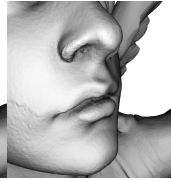
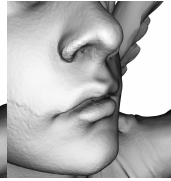
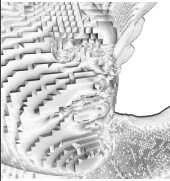






Zoom on lucys face, with ambient occlusion shader, in different quantizations						
#bits=10	12	14	16	18	20	22
						
difference image						
						
PSNR, RMS error, final data size, and loss of triangles due to quantization						
tris=2.7M (-90%)	20.6M (-27%)	26.2M (-6.6%)	27.6M (-1.6%)	27.9M (-0.5%)	28.03M (-0%)	28.05M (-0%)
size=14MB	99MB	136MB	157MB	173MB	189MB	204MB
PSNR=12.3	16.4	25.4	34.9	38.1	45.0	46
RMSE=16k	9.9k	3.6k	1.2k	940	587	142

Figure 3: Quantization artifacts relative to number of bits used for the initial quantization. Artifacts are obvious to severe for anything less than 16 bits; at 16 bits the model looks “OK” without a reference, but a slight un-evenness of the surface is still apparent when presented with a reference image. At 18 and more, differences are hard to see even when flipping between image and reference image. Though the exact threshold where quantization becomes noticeable depends on the model, for all our models except *xfrog* 20 bits have proven to work fine.

The efficiency of this approach depends heavily upon the actual encoding of the meshlets and, in particular, on how exactly the model is split into meshlets.

3.1. Meshlet Encoding Format

Given a (not compressed) candidate mesh to be encoded we want to encode this meshlet as efficiently as possible, yet in a way that we can easily decode each individual BVH node, primitive or vertex. Note that we assume this mesh to *already* have a proper merging of neighboring triangles into quads (where applicable). We also assume that the meshlet generation step already computed the proper vertex sharing information, and that duplicate vertices have already been removed.

3.1.1. BVH Nodes

For each BVH node, we have to encode the following information: the bounding box (relative to the parent), whether it is a leaf or a inner node, a “pointer” to the child nodes or item list, respectively, and the number of children in the node. In addition, since BVH nodes are accessed very often, we want a “native” data format such as 16, 32, or 64 bit that the machine can natively read and process. Of these three choices, 16

bits seemed too few: even assuming 2 bits per relative coordinate to be sufficient, at six coordinates this would leave only four bits for the leaf pointers.

At 32 bits per node, however, spending 3 bits (ie, 8 possible values) per relative BVH coordinate leaves 14 bits to encode all the other information. We chose to spend these 14 bits via 11 bits **offset**, plus 3 bits **count**: For inner nodes, **count** is set to zero, and **offset** gives the child nodes’ offset relative to the current node; for leaves, **count** specifies the number of primitive in the leaf, and **offset** the offset the ID of the first primitive in this leaf. To avoid having to store a separate item list we do the usual trick of storing primitives in the order they are referenced by the tree. Three bits allow only up to 8 primitives per leaf; it is up to the construction code to ensure that this requirement is met.

Rather than spending three bits on the **count** field we also experimented with a single **isLeaf** bit together with a special “end of list” bit in the primitives. This however usually produced slightly lower compression rates since taking a bit from the primitive encoding allowed encoding a smaller range of vertex IDs.

3.1.2. Primitives

For the primitives we have to encode material and up to 4 vertices. Again, ideally we would like to use a native data format (16, 32, or 64 bytes) for fast access to individual primitives. Of these, 16 bits seems far too few to ever encode 4 vertex IDs; while 64 would allow up to 65k vertices per meshlet even without doing anything clever—and would already require four bytes per primitive just for the vertex IDs. At 32 bits, we’d need half that, but have to be a bit more clever in how to encode the total of five IDs (material and 4 vertices). Even without the material information using an equal number of bits per vertex would leave us with at most 256 vertices we could ever use in a meshlet. If, however, there was some amount of correlation between the vertex IDs (ie, all low, or all high—we will discuss later on how we generate those IDs) we could save bits by expressing all but one of these IDs relative to a base ID.

After experimenting with several different values we ended up with the following encoding, which seems to produce consistently good results

3.1.3. Vertices

Vertices are encoded as bit strings relative to the meshlet’s bounding box (in fact, in integer coordinates we simply have to add the resulting bit strings to the coordinates meshlet bounding box’s lower-left corner). For each meshlet, we first determine how many bits are required to encode vertices in x, y, and z direction, and store these three values B_x , B_y , and B_z (plus their sum B_w - i.e., the total bits required per vertex) in 4 8-bit values per mesh.

All the vertices’ bits are stored successively in a given “bit array” per mesh. Having a arbitrary number of bits per vertex means that decoding a given vertex involves reading B_w bits from a bit offset inside the vertex bit array that usually is not aligned to native word size, and that a given vertex’ bits may not even lie within the same native data word. This requires some significant amount of address computations and bit shifting/masking that will not come for free. On the other hand, choosing the optimal number of bits results in a significant memory savings in particular for meshlets that are either thin and/or long (in which case one or two of the bounding box dimensions need hardly any bits at all!). Even in the general case the number of bits spent on vertices is significant: Vertex memory is “the” biggest individual use of memory, so every bit is important.

Note that each mesh will have different bit values, but all vertices inside the *same* meshlet will be encoded with the same number of bits. In particular,

this means that we can decode an arbitrary vertex by computing this vertex’ bit offset (as $vertexID * B_w$, reading B_w bits from this location, extracting the respective 3 sub-bitfields, and adding them to the mesh bounding box’s coordinate.

In addition to the vertex position, our system also supports vertex normals and texture coordinates where available. Vertex normals are encoded using 16 bits: we precompute a set of 2^{16} unit vectors, and find, for each vertex normal, the one that is closest to the given normal vector (this obviously assumes unit normal vectors). We also achieved good results with as few as 8 bits, but to be on the safe side opted for 16. Textures coordinates are stored as two 16-bit half-float values. This may be too few for some models, but could easily be changed to store full floats (only one of our models has texture coordinates at all). Vertex and Texture coordinates are stored only for meshlets that have any; and it is perfectly possible to mix meshlets with and meshlets without those attributes.

3.2. Finding Suitable Meshlets

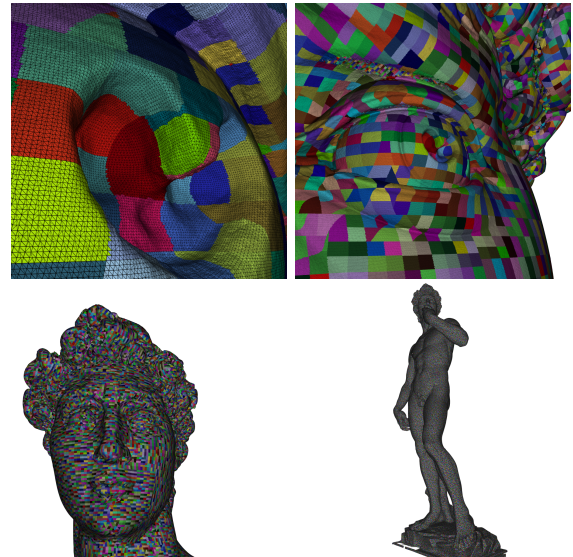


Figure 4: Result of our meshlet decomposition as visualized on the 1 billion triangle *david(v2)*: The wireframe shows individual input triangles; and triangles with the same color are encoded in the same meshlet. Note how our meshlet decomposition creates nice, compact meshlets, and how neighboring meshlets fit seamlessly. a)-d) start with a zoom on the david’s left iris (the only view where individual triangles can be differentiated), and gradually zoom out to the entire model (where entire meshlets project to single pixels).

One of the key variables controlling the compression

rates of our approach is choosing the “right” subdivision of the model into meshlets: On one hand, bigger meshlets means more potential for vertex sharing: Whereas a single quad needs 4 vertices, a fully connected mesh needs only $1+\epsilon$ vertices per quad—ie, a 4x reduction in memory used for vertices. Also, bigger meshlets mean that all other “per mesh” cost can be amortized over more vertices, that we need newer nodes in the top-level tree, etc.

On the other hand, given the low number of bits to encode each primitive there are some meshlets that simply cannot be encoded: in particular the restricted number of bits to encode the leaf item list, material ID and vertex IDs puts a strict limit on how many primitives a mesh may have, and on what quads’ quadruple of vertex IDs can be encoded. Even without this restriction (after all, we might fix this by spending more bits on the quads and/or BVH nodes!) there is a limit as to how big we would want the meshes to become: Assuming the size of the primitives to be fix, putting more primitives into the same mesh means this mesh’s bounding box will expand, requiring us to then spend more bit on encoding vertices. This would be particularly the case when inserting primitives that are far away from the original mesh, and that would therefore result in a over-proportional increase in mesh bounds.

Given these complex and counteracting factors influencing the “best” meshlet size we decided that a good a-priori threshold for determining the meshlet size would be too hard to define, and instead opted for a algorithmic approach in which we determine the meshlet size automatically: We first build a simple binary BVH over the entire model (using a cheap spatial median BVH; operating out of core, see Section ??), down to a small number of primitives per leaf, and make a temporary meshlet from these few primitives. Then, we successively try to merge neighboring leaf nodes into successively larger meshlets until either the merged meshlet can no longer be encoded, or until the merged meshlet turns out to require more memory than the two individual meshlets. If the merge was successful we remove the two smaller meshlets and replace them with the merged one; otherwise we tag the two input meshes as “final” meshlets, and emit them to disk. Essentially, this algorithm computes a “cut” through the initial BVH that provides a good (though not necessarily globally optimal) decomposition of the scene into meshlet.

Since we originally started with a spatial median BVH neighboring meshlets in the tree are usually next to each other spatially, which results in good vertex sharing and nice, compact meshlets (see Figure 4). In all other respects, the initial BVH does not influence either the top-level BVH nor the quad-BVHs inside each

meshlet: “merging” a mesh means that a new mesh is completely re-built from the constituent primitives, which will automatically compute the proper vertex sharing from vertices found in both meshes, and which will also compute a fresh quad-BVH for these vertices before encoding. For the top-level BVH, we *can* re-use the initial partitioning, but can just as well build a completely new SAH BVH over the meshlet bounds.

3.3. Meshlet Encoding Process

The process of encoding a given meshlet is actually quite an interesting problem all by itself: Given that we spend varying numbers of bits for each of a quads’ different vertex IDs the order of the vertices in the mesh—and just as much, the order of the vertex IDs in the quad—strongly influences whether a given mesh can be encoded, or not. For example, using the bits specified above the quad (0,1,2,20) can not be encoded (due to the difference in w and z coordinates to require more than 4 bits), but either swapping vertices 5 and 20, or “rotating” the quad’s vertex IDs to (2,20,0,1)—in both cases, yielding logically exactly the same mesh—*does* allow encoding this same mesh.

Finding the *optimal* order of vertices and vertex IDs is quite a challenging task, and probably NP-complete (though we will not attempt a proof here). As such, we resorted to a simpler greedy approach that incrementally adds vertices and quads on the fly, and which seems to produce good results: First, we re-order the vertices along a Hilbert curve to increase spatial coherence of the vertices (and thus, hopefully increase similarity of the vertex IDs). Then, we iterate over all primitives and try to *add* each primitive to the meshlet: for each each of the quads’ vertices we first check if the respective vertex has already been added, and return its vertex ID if this is the case; if not, we encode that vertex and add it to a bit-stream containing all the mesh’s encoded vertices. We then try to encode the resulting four vertex IDs into the 32-bit compressed primitive and proceed to the next primitive if successful.

3.3.1. Index rotations

If the quad could *not* be encoded we try all combinations of rotating and/or reversing the order of the vertex IDs, and try again for each such combination (no new vertices get added this way; we only change the order of the vertex IDs). This is also the reason for the **flip** bit in the primitive encoding: rotating IDs changes which pair of vertices connect the shared edge, and since our quads are not guaranteed to be either planar or convex we need this information for properly decoding the two triangles after rotating (without this we could have used only a subset of all possible

scene	naive		rotations		restart	
	PpM	b/t	PpM	b/t	PpM	b/t
crown						
pplant						
lucy						

Table 1: Impact of the vertex ID rotations and restart optimizations (see Section 3.3) on meshlet encoding efficiency as measured in average number of prims per mesh (PpM) and on the average number of bytes per input triangle (b/t).

combinations, which experiments show produces lower compression rates even considering that not allowing flips frees one more bit for the one of the offsets).

3.3.2. Re-start with vertex replication

If a quad *still* could not be encoded in any of its rotations, the reason is typically that some of the vertices have very low IDs, while others have high IDs, with too high a difference between these IDs to encode in the low-bit offsets. To catch this case we take the lowest vertex ID and replicate its associated vertex at the current end of the vertex bit stream, thus getting a higher vertex ID for this vertex. We then “re-start” the encoding of this (modified) quad, again in all combinations, etc, and go on to replicating the then-smallest vertex ID, etc, until a combination can be found. If no proper encoding could be found even after all four vertices have been replaced we throw an exception that tells the meshlet merging stage that this mesh cannot be encoded.

Replicating vertices obviously results in a certain increase in memory required for the vertices; however, this is possibly offset by a higher degree of vertex sharing in a larger mesh, and the meshlet merging stage (Section 3.2 will automatically reject the merged mesh if that merging did not produce an overall savings. On average we see roughly 20% of vertices to be replicated, which indicates that a globally more optimal ordering scheme might save an additional 20% of vertex memory. However, as shown in Table 1, index rotations and re-start alone can significantly increase the average number of primitives that can be encoded with the few bits we have, and typically achieve pretty good results, so for now we are perfectly happy with this solution. The total outcome of our scheme for producing meshlets can also be seen in Figure 4.

3.4. Out of Core QTQ Construction

After the basic ideas and data formats are now already described we will use this section to describe how we actually build those data structures.

3.4.1. Conversion to streaming format

Our input models come in a large variety of input formats—PLY, OBJ, Embree, VRML—with different mesh and material formats. We first convert the input model into a unified “streaming” format in which the model is a stream of self-contained polygons, where each polygon is stored as a material ID plus a list of vertices, and where each vertex can store position, normal, vertex color, and texture coordinate (vertex attributes that aren’t present are explicitly marked as such). Materials themselves are stored separately in an auxiliary XML file that encodes whatever material properties, textures, etc the respective input format had specified (this is a variant of the Embree XML format). The stream file itself is automatically and transparently streamed through `g(un)zip` to reduce disk space.

3.4.2. Initial Quantization

The next stage is to convert this stream file into a linear array of fixed-size, self-contained primitives where each primitive has a fixed number of four vertices (triangles are encoded as quads with the last two vertices being the same). Having fixed-size, self-contained primitives later facilitates memory-mapping the resulting file, and processing it in an out-of-core fashion. Consequently we call this the *memory file*. We generate this memory file by streaming through the streaming geometry file polygon by polygon. Each vertex is first snapped to the global quantization grid; any degeneracies produced through this snapping (zero-length edges, zero-area triangles, etc) get resolved immediately before ever being passed to the next stage.

The resolution of this initial quantization grid can be specified by the user, either through a fixed quant size (such as “10 microns”), or through a certain number of bits relative to the widest dimension of the model’s bounding box. Obviously, any form of quantization always leads to *some* loss of information, and in particular when specifying a coarse quantization grid one can easily start to see otherwise smooth surfaces to become “jerky” (by vertices snapping to the nearest integer coordinates), or even by losing triangles after some of their vertices snap to the same discrete positions.

In theory, specifying a grid resolution of 23 bits should result in “quasi” loss-less encoding: assuming an IEEE floating point representation for the input model each vertex has only 23 bits of mantissa for each vertex, anyway, so 23 bits inside the same bounding box should result in “comparable” accuracy. However, unlike our grid floating point values are not equally dense, anyway, so for the really complex models such as the `boeing` we typically see *some* loss of triangles

scene	#tris	bits	num	#per meshlet			memory spent on							bytes	
	input		prims	node	prim	vtx	toplevel	meshlet	node	prim	vtx	nor	tex	sum	/tri
Non-scanned Models															
e87	4.3M	20	2.3M	194	437	911	334K	334K	4.1M	9.1M	21.5M	9.5M	—	44.9M	10.5
crown	4.9M	20	2.5M	197	455	733	344K	344K	4.3M	9.8M	18.9M	7.8M	15.8M	57.3M	11.8
PPlant	12.1M	20	6.1M	264	654	909	594.5K	594.6K	9.8M	24.3M	36.9M	—	—	72.2M	5.9
e89	27.2M	20	14.4M	99	228	432	4.0M	4.0M	25.1M	57.6M	112.2M	54.3M	—	257.2M	9.5
MPI	73.0M	20	37.2M	276	635	960	3.7M	3.7M	64.6M	148.7M	256.8M	112.2M	—	589.6M	8.1
boeing	347.6M	20	190.7M	62	143	274	85.3M	85.3M	329.8M	762.8M	1.5G	—	—	2.7G	7.9
xfrog	1.6G	20	802.6M	282	646	1K	79.5M	79.5M	1.4G	3.2G	7.8G	3.2G	6.3G	22.1G	13.9
Scanned Models															
lucy	28.0M	20	15.3M	465	1K	1K	921.7K	921.7K	26.8M	61.2M	98.9M	—	—	188.7M	6.7
david(V1)	56.1M	20	29.3M	459	1K	1K	1.8M	1.8M	51.1M	117.3M	181.0M	—	—	352.9M	6.3
stmatthew	371.6M	20	195.5M	467	1K	1K	11.7M	11.7M	340.7M	782.2M	1.1G	—	—	2.2G	6.0
atlas	505.7M	20	268.6M	491	1K	1K	15.3M	15.3M	469.8M	1.1G	1.5G	—	—	3.0G	6.0
david(V2)	921.0M	20	485.4M	469	1K	1K	28.9M	28.9M	845.7M	1.9G	2.5G	—	—	5.3G	5.8
david(V3)	2.0G	20	1.0G	254	571	1K	114.7M	114.7M	1.8G	4.1G	9.1G	—	—	15.3G	7.5

Table 2: Encoding statistics for our test scenes. Scanned models typically have only a single material, and no vertex attributes, and can be encoded very efficiently at XX - XX bytes / triangle. Since we can only encode up to 4 materials per meshlet non-scanned models typically have smaller meshlets; and in particular the additional memory for vertex normals and texture coordinates is expensive. Nevertheless, even including additional vertex attributes these models all encode in XX to XX bytes/tri.

even when specifying *more* than 23 bits (though the current version only supports up to 23 bits to enable floating point processing, an earlier version did support more).

In practice, we typically see *visible* quantization artifacts only for relatively low resolutions of 16 or less bits; however, in order to not having to fine-tune parameters for each model we typically opt for a rather safe 20 bits per model, which even in a $100m \times 100m$ domain—which would conveniently fit even the entire **boeing**—would produce an accuracy of a tenth of a millimeter, roughly the diameter of a human hair (for the 5.17m tall **david**, it would correspond to 5 microns). The only model where this does not suffice is the **XFrog** model, where we have individual, thin leaves on a $1km \times 1km$ plane; this model usually comes in a two-level format where each plant is represented in its coordinate system—even without any quantization, expanding this two-level representation into a single coordinate system is already a stretch for floating point numbers, so we opted to use all 23 bits for this model.

3.4.3. Generating Quads from Triangles

Since encoding one quad is more efficient than encoding two separate triangles we want to maximize the number of quads in the model, and increase the ratio of quads by merging triangles where applicable (Quads already present in the input are handed through “as is”). In particular, note that neither our encoding nor our ray tracing scheme requires the two triangles form-

ing a quad to be co-planar. I.e., our system explicitly allows to take *any* two triangles sharing an edge and to encode those as a quad.

For most engineering models simply checking successive triangles for a shared edge will be sufficient; for some of the scanned models however triangles come in a more random order where this simple strategy doesn’t work. Thus, we added a explicit *recombination buffer* stage that buffers the last $N = 100$ un-merged triangles. As soon as a new triangle is encountered in the input model we try merging it with another triangle in this buffer, and, if successful, discard the two triangles and emit a quad; if no such partner could be found we insert the triangle into the recombination buffer where it is available for later merging. If that buffer is now larger than the maximum size we remove the oldest unmerged triangle from this buffer, and only then actually emit it as a triangle (ie, a quad with vertices 2 and 3 being the same). This merging significantly reduces the total amount of primitives to be encoded, and works equally well for all models. For the **david(v3)** model, merging reduces the number of final primitives from 2.041 billion triangles to only 1.024 billion “quads” (of which a “tiny” 5.9 million are triangles that could not be matched), a savings of almost exactly 2x. For engineering models the ratios are slightly smaller, but still significant: For the **boeing**, merging triangles reduces the number of primitives from 348 million to 190 million (1.8x).

3.4.4. Temporary BVH Construction

Once all primitives have been quantized, merged, and written to the memory file have to proceed to build the initial BVH that the recursive meshlet generation stage can then work upon. As mentioned above this BVH is only required for the meshlet generation, so we use a cheap builder that recursively computes the bounding box of the respective primitives' centroids, determines the dimension in which this box is widest, and splits along the spatial median of that dimension. The entire process works out of core: The file containing all primitives is memory mapped, and subtrees are recursively partitioned "in place" by successively swapping primitives similar to a quicksort partitioning step.

To keep memory usage low we never explicitly store the resulting tree with inner nodes, bounding volume information, child pointers, etc. Instead, for each primitive we only store one single 64-bit integer whose bits encode the path from the root of the tree to the respective leaf (i.e., the uppermost bit encodes whether it's in the left or right subtree of the root node, etc). The meshlet generation stage can then determine the start and end of each subtree by simply doing a binary search for the first and last primitive whose bits match the respective subtree, etc. This does not obtain the subtrees' bounding box; however, since the meshlet merging proceeds bottom up we have this information during any point in time, anyway: for the leaves it's simple the bounding box of the leaf primitive; for inner nodes it's the union of the two merged meshlets' bounding volume.

The same procedure can also be easily modified to guarantee a certain maximum number of materials in each subtree, or even to guarantee a single material per meshlet: rather than performing all tree partitions based on the spatial median we can also force splits to partition based on material ID until a certain threshold of different materials is reached, and only then switch to spatial splits; nothing else in the rest of the system is affected by that change. In theory that would allow to free the primitives' two material bits for other tasks by enforcing a single material per meshlet; however we currently do not use this scheme except for the *xfrog* model for which using only two material bits otherwise results in too small meshes.

3.4.5. Parallel Subtree Processing

To facilitate parallel processing we can also specify that a build process be restricted to a given subtree of the memfile's initial spatial median tree. If specified the build will first determine the start and end of the respective subtree's ID; then build that subtree's respective meshlets, and write them out to a tempo-

rary file. A final *reconstitution* step then merges the respective partial meshlet files, builds the final top-level BVH over the meshlets (using a binned surface area heuristic builder), and generates the final binary output file.

4. Rendering

Once our compressed model has been generated, rendering it is rather straightforward. We have implemented a prototypical—and by no means optimized—ray traverser for our format in the itself prototypical RIVL ray tracer, a close early relative of the ISPC-based *Embree* 2.0 ray tracer. Unlike *Embree* our RIVL codebase uses a slightly different compiler called IVL (though there is no reason why an implementation in ISPC would lead to any different results), and does not use any specially hand-optimized traversal kernels; instead, RIVL is designed for simplicity, and all ray traversal is implemented in the IVL SPMD language, typically using a packet tracing approach operating on a SAH-based binary BVH, and including support for instancing built on top of virtual geometry classes. We implemented our traverser as simply another geometry class with different *trace* and *occluded* functions, all implemented in IVL.

This geometry class gets initialized with a single binary block of data that encodes the model; this is typically a memory-mapped file when run on the host, or a single "buffer" (allocated using IVL's equivalent of CUDA's *cudaMalloc*) when run on a Intel Xeon Phi node. The geometry will allocate some minor utility classes for the meshlets, but will simply point into the binary buffer for all bulky data like BVH nodes, primitives, etc. All BVHs etc are already pre-built in this buffer; and no additional data structures ever get built at all.

4.1. Straight-forward Packet Traversal

When RIVL passes a ray—or to be more precise, an entire *packet* of rays—to our compressed geometry type we first traverse this packet through our "top-level" BVH built over the meshlets, and automatically switches to traversing the meshlets once it reaches them. Inside each meshlet, we switch to a different—and quite straight-forward—traversal based on a quad-BVH: Since the root node is the only node for which our builder cannot guarantee that it is fully occupied we initially iterate over the children of the root node; we decompress each root child node relative to the meshlet's bounding box (available via the bounding volume of the leaf node that referenced this meshlet), test each such node, and it onto the traversal stack if it intersected any of the rays.

We then enter the main traversal loop that—again straightforwardly—proceeds as long as any node remains on the stack: we pop the top-most node off the stack (no ordering of the stack nodes is done yet, at all), decode that node’s children, etc.

So far, the only thing different from a straightforward QBVH-traverser is that in addition to the node itself we also have to push each node’s decoded bounding volume to the stack: Since all nodes are encoded relative to its parent we need this information to properly encode the respective child nodes later on.

As soon as the packet reaches a leaf, we iterate over the primitives, decode the respective vertex IDs, use these IDs to decode the respective vertices from the vertex bit array, and perform the one or two triangle intersections. If either of these triangles found an intersection we squeeze meshlet ID, primitive ID, and a bit indicating whether it was the first or second triangle of this quad into the 32-bit `primID` field that RIVL reserves for specifying a geometry’s “primitive” ID. Based on this information, the shader can afterwards come back to our geometry and ask for additional information like normals, texture coordinates, etc, and the geometry will be able to accurately pinpoint the actual triangle and vertex coordinates for that given hit point. Being thus fully integrated into the RIVL rendering system our compressed geometry traverser is fully compatible with the rest of RIVL, and can be used with any of its renderers and shaders (including a complete Embree-style path tracer, see Figure ??), or in theory, even with other geometry types.

4.2. Optimization Potential

Though this implementation proved perfectly sufficient to interactively render all of our models (we will discuss its performance in the following section) it is *intentionally* simplistic: The goal of this paper was to concentrate on the best model representation and encoding scheme; not on the most efficient way of traversing this data on any particular architecture (in theory, the same scheme should work just as well on GPUs, integrated graphics chipd, or specialized ray tracing hardware such as proposed by XXX et al or XX et al; or in any other programming language such as ISPC, OpenCL, or CUDA).

That said, there are many rather obvious sources of possible optimizations: for example, given the cost of traversing a meshlet or even doing all the decodes in a leaf any sorting for the traversal stack should easily pay off. More importantly, we currently decode nodes, primitives, and vertices in purely scalar code, even though it would be trivial to implement that in manual SIMD code by decoding all four child nodes (with

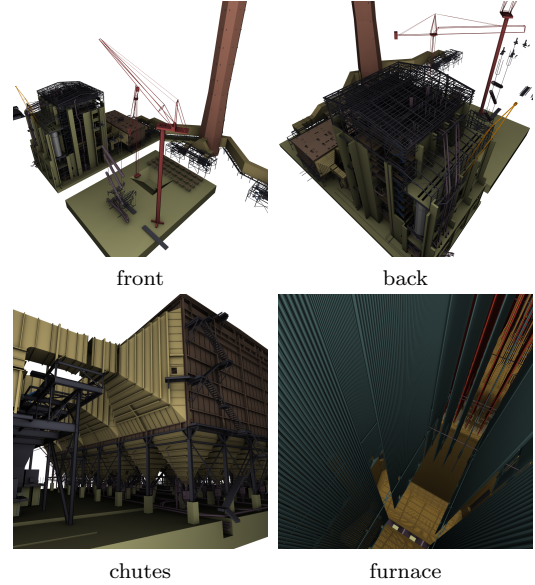


Figure 5: Several views of the powerplant including, in particular, the *furnace* inside view that contains most of the actual, intricate geometry of the model.

2 times 3 coordinates each) in parallel, or doing the same for the four vertices of a primitive. SIMD instructions could also be used to great effect when moving entire boxes on/off the stack, when sorting this stack (not yet done), etc.

In the end, given how tiny even entire meshes become for anything other than very close views (see Figure 4) it also seems like a natural choice to abandon a packet tracing approach altogether, and to instead use a pure single-ray SIMD approach similar to the one presented by Benthin et al. [BWW*]. Though the rest of RIVL would still be packet-based, in particular the transition from the top-level BVH traversal to meshlet traversal seems like a natural choice for switching from packet to single-ray SIMD mode.

5. Results

5.1. Scenes

TODO: embree scene size comparison

5.2. quantization quality

6. Summary and Discussion

Acknowledgements

The `crown` model is courtesy Martin Lubisch, www.loramel.net. The `PowerPlant` model is courtesy of the Gamma group at University of North Carolina

scene	input model size			Embree					QHC (22bit)				
	format	size	bzip'ed	vtx	prim	bvh	total	B/tri	vtx	prim	bvh	total	B/tri
pplnt													
lucy													
david													

Table 4: comparison to embree: memory consumptoin...

2xXeon XX				4xXeon Phi XX		
scene	emb	QHC	ratio	emb	QHC	ratio
eyelight						
pplnt						
lucy						
david						
ambient occlusion						
pplnt						
lucy						
david						

Table 3: comparison to embree: performance (eyelight and simpleao). compare both avx and mic

ceedings of Graphics Interface 2010 (2010), Canadian Information Processing Society, pp. 153–160. 2

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum* 20, 3 (2001), 153–164. (Proceedings of Eurographics 2001).

at Chapel Hill (UNC). The `lucy`, `stmatthew`, `atlas`, and `david` (v1-v3) models are courtesy of the Stanford Scanning Repository; special thanks also to Thiago Ize and the SCI Scientific Computing and Imaging Institute for providing these models. The `MPI Building` is courtesy Vlastimil Havran. The `boeing 777` model is courtesy David Kasik from Boeing corp. The `xfrog` scene is courtesy Oliver Deussen. We would like to also, and in particular, acknowledge Benjamin Segovia and Manfred Ernst, who provided valuable insight in their highly related paper that unfortunately never got published, as well as Carsten Benthin and Sven Woop, both of whom provided valuable counsel and ideas. Wes Shimanek helped tremendously by providing some Intel SSD drives that allowed for reducing out-of-core build time from days to hours.

References

- [BWW*] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W. R.: Combining Single and Packet Ray Tracing for Arbitrary Ray Distributions on the Intel(R) MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics*. (accepted for publication). 9
- [KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: Racbvhs: Random-accessible compressed bounding volume hierarchies. *Visualization and Computer Graphics, IEEE Transactions on* 16, 2 (2010), 273–286. 2
- [MW06] MAHOVSKY J., WYVILL B.: Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum* 25, 2 (June 2006). 2
- [SE10] SEGOVIA B., ERNST M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Pro-*