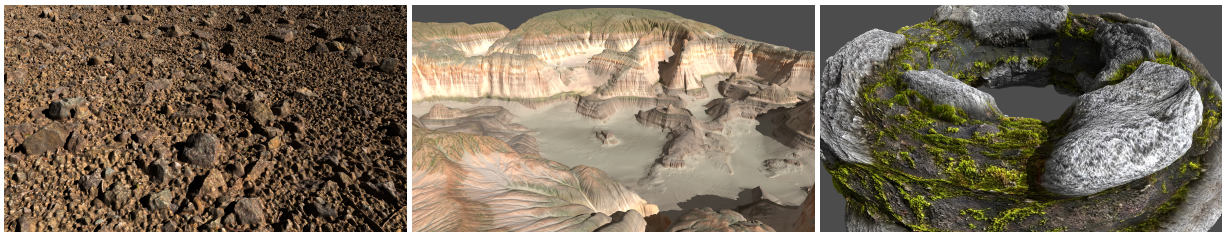
Ray Tracing Lossy Compressed Grid Primitives

Carsten Benthin Karthik Vaidyanathan Sven Woop

Intel Corporation



Three example scenes (Ground 42.4M triangles, Canyon 33.5M triangles, Torus 75M triangles) generated with highly detailed 8K displacement maps and interactively rendered with a diffuse path tracer. Our lossy compressed grid approach requires just $204-380$ *MBytes of memory for ray tracing these complex scenes, including geometry and the bounding volume hierarchy.*

Abstract

We propose a new watertight representation of geometry for ray tracing highly complex scenes in a memory efficient manner. Polygon meshes in the scene are first converted into compressed grid primitives, which are represented by a base bilinear patch with quantized displacement vectors. Ray-scene intersections are then computed by efficiently decompressing these grids on-the-fly and intersecting the implicit triangles. Our representation requires just $5.4-6.6$ *bytes per triangle for the combined geometry and acceleration structure, resulting in a* $5-7\times$ *reduction in memory footprint compared to indexed triangle meshes. This is achieved with less than 15% increase in rendering time.*

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing; Visibility; Massively parallel algorithms;***

1. Introduction

Ray tracing highly complex geometry requires lots of memory when on-the-fly generation of geometry (e.g. subdivision surfaces) is not applicable. Given that indexed triangle meshes typically require ≈ 30 bytes per triangle, with additional bounding volume hierarchy data (BVH) on top of that, a more efficient geometric representation is required to reduce the memory footprint of large static meshes.

In this paper, we propose an efficient lossy compression scheme for triangle meshes which can be represented by a set of vertex grids. Our scheme follows a hybrid approach of applying lossy compression to the majority of the vertex data while keeping a small subset uncompressed. At the same time, all triangle connectivity data is implicitly expressed. This reduces the memory consumption compared to index triangle meshes and existing grid-based representations by multiple factors while at the same time only slightly affecting the rendering performance.

2. Previous Work

Most previous work has been focused on reducing the memory consumption of BVH data by applying reduced precision techniques [MW04, Kee14, VAMS16, YKL17] to encode BVH nodes. An approach which reduces memory consumption of both BVH and geometry data has been proposed by Segovia et. al. [SE10] who applied hierarchical mesh quantization to combine BVH and triangle data into a single unified data structure. All vertices within a BVH leaf are quantized with respect to the leaf bounding box. Gaps due to different quantization reference points are avoided by snapping the vertices and leaf bounding boxes to a global grid over the scene.

Another combined approach are random-accessible compressed BVHs [KMKY10] which decompose adjacent nodes into clusters to support random access on compressed BVHs and quantized bounding boxes. The nodes' connectivity in each cluster and the indices of the triangle are expressed with lower precision data types or delta coding. Clusters and meshes are further compressed using a dictionary-based compressor.

In wide BVHs, leaf nodes make up most nodes, but they are less often intersected than the inner nodes. Benthin et al. [BWWA18] introduced dedicated compressed multi-leaf nodes and reduced memory consumption within leaf nodes by exploiting data sharing while also minimizing performance degradation by compressing only those.

DOI: 10.2312/egs.20211009

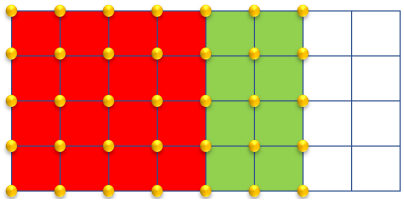


Figure 1: *A 7×5 grid of vertices (yellow points) is subdivided into two 5×5 sub-grids. All quads of the first 5×5 sub-grid are completely valid (red) while for the second sub-grid only a sub-region is valid (green). A sub-grid with just valid quads corresponds to 16 quads or 32 triangles.*

3. Conversion to Vertex Grids

In most cases, scene input geometry is given as a set of indexed triangle or quad meshes. Converting those meshes into set of vertex grids is done in a pre-processing step. The resulting vertex grids have a varying resolution, depending on how many triangles/quads could be merged into a single grid. A grid itself just stores the resolution and a pointer to the vertices. All triangle connectivity information is implicitly expressed. The vertices themselves are rearranged to fit the grid's horizontal and vertical layout.

The conversion step uses a locally greedy algorithm to merge adjacent triangles/quads into the grid, trying to maximize its size. Once converted, each grid is subdivided into smaller fixed size sub-grids, over which a BVH is finally built. In our implementation we use a 5×5 vertex grid resolution for the sub-grids (see Figure 1). Therefore, each sub-grid implicitly corresponds to $4 \times 4 = 16$ quads or 32 triangles. Invalid triangles of partially filled sub-grids are ignored during ray-triangle intersection.

4. Lossy Vertex Sub-Grid Compression

The conversion step from Section 3 produces a set of 5×5 sub-grids referencing uncompressed vertex data. The next phase converts this data into a self-contained, lossy, compressed version. The basic idea of our lossy compression scheme is to represent the uncompressed vertices of a sub-grid as a set of quantized displacement vectors starting off a simple base surface (see Figure 2), which is closely aligned to the shape of the sub-grid. In our implementation, we use a bilinear patch as the base surface which is completely defined by four vertices and already allows for handling a certain amount of curvature. As bilinear control vertices, we use the sub-grid's four corner vertices. This means that at these points the base primitive matches the vertex grid exactly.

The bilinear patch is now evaluated (bilinear interpolation) on a uniform 5×5 grid in *u,v* space, resulting in a 5×5 grid of points on the patch. For each point on the bilinear patch a displacement vector to the corresponding point of the uncompressed vertex grid is computed. This displacement vector is then compressed using a lossy encoding scheme (see Section 4.1), which reduces its size from uncompressed 12 bytes (3 floats) to 3 bytes. As the four corners of the uncompressed vertex grid are used as bilinear patch control vertices, the corresponding displacements will always be zero and do not have to be explicitly stored. This reduces the number of displacement vectors from 25 to 21.

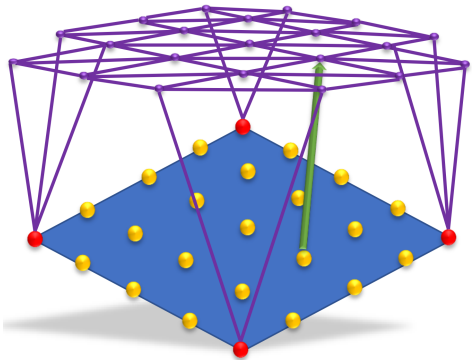


Figure 2: *Encoding of an uncompressed 5×5 vertex sub-grid (purple) into our lossy compressed format. First, a bilinear patch (blue) is created based on the four corner vertices of the vertex grid (red). Next, 5×5 points on the bilinear patch (red and yellow) are evaluated using an uniform grid in u,v space. For each of the 5×5 points on the bilinear patch a displacement vector (green) is computed, taking the difference between the point on the bilinear patch and the corresponding point on the vertex grid. The displacement vector is then quantized with reduced precision.*

In general, our lossy compression scheme is based on the assumption that the displacement vectors which represent the difference between the base primitive and the original grid vertex won't be large relative to world space and can therefore be represented with sufficient accuracy in compressed form.

4.1. Lossy Compressed Displacements

For compressing 3D displacement vectors from an uncompressed 12 bytes to 3 bytes, we evaluated two different approaches, but many others are possible [CDE*14]. The first approach represents the displacement vector as the triple: *octantID*, *unit vector* and *length*. The *octantID* is encoded in 3 bits while the unit vector is mapped to a closest representative inside a set of predefined unit vectors in the first octant. The predefined unit vectors are pre-stored inside a small lookup table and 10 bits (1024 entries) are used to index the table. With 8 octants 8192 different unit vectors can be expressed for the entire unit sphere. The *length* component is compressed to a reduced precision floating point format with a 6 bit exponent and a 5 bit mantissa. For conversion, we simply clamp the original 8 bit exponent of the length and round the original mantissa bits to 5 bits of accuracy.

The second approach uses a shared 6 bit exponent with 3×5 bit mantissas (one per dimension) and a 3 bit *octantID*. The shared exponent is the maximum of the three original exponents and the mantissas are fixed-point representations with respect to the shared exponent floating point value. Memory requirements of both approaches sum up to 24 bits, or 3 bytes per displacement vector. The second approach offers faster decompression but the first offers better accuracy which is why we picked it for our implementation. In general, the chosen distribution between exponent and mantissa bits can be adjusted for use case scenarios with different precision requirements.

Listing 1: *Layout of our 128 bytes 5x5 lossy compressed vertex sub-grid, and the 3 bytes compressed displacement vector.*

```
struct LossyCompressed5x5VertexSubGrid {
 Vec3f bilinearPatchVtx[4];  // 4*3*4=48 bytes
 Displacement displ[21];     // 21*3=63 bytes
 uchar resX : 4;             // 1 byte
 uchar resY : 4;
 uint objectID, primitiveID; // 2*4=8 bytes
 ushort gridX, gridY;        // 2*2=4 bytes
 ushort gridResX, gridResY;  // 2*2=4 bytes
 }
struct Displacement {        // 3 bytes
 uchar3 octantID     :  3;
 uchar3 unitVectorID : 10;
 uchar3 exponent     :  6;
 uchar3 mantissa     :  5;
}
```

4.2. Data Layout

The 5×5 sub-grid resolution allows for fitting 32 triangles into 128 bytes (see Listing 1), lowering memory requirements to just 4 bytes per triangle (assuming all triangles inside the sub-grid are valid). Also, a size of 128 bytes naturally aligns with the 64 bytes cache-line sizes found in most common CPUs. The uncompressed control vertices of the bilinear patch require 48 bytes, while the 21 displacement vectors take up 63 bytes. The rest is taken up by the auxiliary data, e.g. object and primitive ID. The position of the sub-grid in the original grid (gridX,gridY,gridResX,gridResY) are needed to compute per grid *u,v* coordinates (see Section 5). The internal resolution of the sub-grid (resX,resY) is required for expressing partially filled sub-grids, which typically occur at the right and lower border of the original grid.

4.3. Lossy Sub-Grid Decompression

Decompressing the lossy compressed sub-grid into an uncompressed vertex grid is straight forward. First, the uniform 5×5 grid in u,v space is used to evaluate the bilinear patch at 5×5 positions. Next, the 21 displacements are decompressed, extended to 25 displacements by filling in zeros at the corners and added to the patch positions, resulting in a 5×5 grid of uncompressed vertex positions. Decompressing a displacement only requires loading the unit vector based on the *unitVectorID*, setting its sign bits based on the *octantID* and multiplying the result by a floating point value constructed out of the 6 bit *exponent* and 5 bit *mantissa* values.

As neighboring sub-grids share the same two bilinear control points and the same displacement vectors along the shared border, the decompressed vertices along the border will be exactly the same. This property is important for achieving watertight ray sub-grid intersection (see Section 5).

5. Ray Traversal and Intersection

Once all lossy sub-grids are generated, a bounding volume hierarchy (BVH) is built using the bounding box of each sub-grid. As the main focus is on saving memory, a compressed 8-wide BVH is chosen [YKL17]. Each BVH leaf references a single sub-grid. If the ray intersects a BVH leaf, the lossy compressed sub-grid is loaded and temporarily decompressed into a 5×5 grid of uncompressed vertices. These 32 triangles are now intersected with a single ray. We chose an approach which intersects all eight triangles (four quads) of a single grid row at once using 8-wide vector instructions. For a 5×5 sub-grid four rows have to be intersected at most. Note that for wider vector instructions, e.g. 16-wide, just two iterations would be required.

An alternative intersection approach is to first extract a set of bounding boxes from the 16 quads and intersect them with the ray before performing single ray-triangle intersection tests. Whether this approach is faster depends on the underlying architecture and efficiency of the vector instruction set. For simplicity, our implementation skips the bounding box pre-test.

Once all ray-triangle intersection tests are performed, the closest valid triangle intersection is reported as hit, and its local per-triangle *u,v* coordinates are transformed into per grid *u,v* coordinates values, e.g. the final *grid_u* is computed by $grid\_u = (triangle_u + gridX)/(gridResX - 1)$. As the shared border between adjacent sub-grids evaluate to exactly the same vertices (see Section 4.1), using a watertight ray triangle intersection test [WBW13] makes the ray sub-grid intersection test watertight as well.

Scene	Ground	Canyon	Torus
Triangles (M)	42.4	33.5	75
Vertices (M)	21.2	16.7	37.7
	Embree Triangle Meshes		
Geometry (MBytes)	1109.8	877.6	2092.4
BVH (MBytes)	333.5	286.6	631.4
Total (MBytes)	1443.4	1164.2	2723.8
Per Triangle (Bytes)	35.6	36.4	37.8
	Embree Grids		
SubGrids (M)	5.3	4.2	9.4
Geometry (MBytes)	422.0	331.9	722.2
BVH (MBytes)	101.8	81.2	137.8
Total (MBytes)	523.8	413.1	860.0
Per Triangle (Bytes)	16.3	12.9	11.9
	Lossy Grids		
SubGrids (M)	1.3	1.0	2.4
Geometry (MBytes)	169.9	128	288
BVH (MBytes)	42.2	37.7	90.6
Total (MBytes)	204.1	165.7	378.6
Per Triangle (Bytes)	6.6	5.4	5.2
Reduction vs. Meshes	**5.4**×	**6.7**×	**7.3**×
Reduction vs. Grids	**2.4**×	**2.4**×	**2.3**×

Table 1: *Memory consumption during rendering for the three example scenes. Including BVH data our lossy grid approach requires just* $5.2-6.6$ *bytes per triangle, while Embree's uncompressed grid and indexed triangle mesh approach require* $2.3-2.4\times$ *and* $5.4-7.3\times$ *more memory (factors even slightly higher when excluding BVH data).*

6. Texture Coordinates and Vertex Attributes

Interpolating general per vertex attributes, like texture coordinates, requires vertex data to be stored in a similar grid layout as the vertex positions. As the ray/sub-grid intersection (see Section 5) returns per grid u,v coordinates, these coordinates can be used to compute the grid cell along with the four surrounding grid indices. These indices are then used to look up per vertex attributes in the separate vertex attribute grid. The loaded attributes are finally interpolated using local per grid cell u,v coordinates.

Scene	Ground	Canyon	Torus
Embree Tri-Meshes	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)
Embree Grids	0.93 (0.65)	1.08 (0.76)	1.03 (0.83)
Lossy Grids	0.87 (0.16)	1.07 (0.19)	0.98 (0.12)

Table 2: *Relative rendering performance and last-level cache misses. Embree's uncompressed grids achieve* $0.93-1.08\times$ *of Embree's triangle mesh performance. The decompression overhead and the cost of more ray-triangle intersection tests are offset by a large reduction in cache misses* $(0.12-0.16\times)$ *due to a smaller memory footprint. Our lossy grid approach achieves* $0.87-1.07\times$ *the rendering performance of Embree's triangle mesh and* $0.94-0.99\times$ *of the Embree's grid performance.*

7. Results

We have integrated our lossy grids approach into the Embree ray tracing kernel library [WWB*14] and compared against Embree's standard indexed triangle meshes and uncompressed grid primitives. Both of these approaches rely on uncompressed vertex data. Embree's grid approach sub-divides each grid into 3×3 sub-grids, while each sub-grid just stores it's position inside the grid and a reference to it (12 bytes). As mentioned in Section 5 an 8-wide compressed BVH [YKL17] is used in all three cases, either built over triangles or sub-grids. All measurements were performed on a dual-socket Intel® Xeon® E5-2699 v3 workstation (56 cores total) with 96 GB of memory using the Clang 11.0 compiler.

Table 1 shows detailed memory consumption for three example scenes during rendering. For triangle meshes, geometry data includes vertex and index data, while for grids, it includes vertex and sub-grid data. Including BVH data, our lossy grid approach requires just $5.2-6.6$ bytes per triangle, while Embree's grid approach requires $2.3-2.4\times$ more, and indexed triangle meshes even $5.4-7.3\times$ more. Excluding BVH data, the memory consumption per triangle goes down to $4.0-4.1$ bytes per triangle. It is worth mentioning that our lossy grid approach uses 5×5 subgrids which results in $4-5\times$ lesser sub-grids than Embree's 3×3 sub-grid approach. The smaller number of sub-grids results in a smaller BVH at the expense of having larger leaves resulting in a higher number of ray sub-grid intersection tests.

Table 2 compares rendering performance using triangle meshes, grids, and our lossy grids approach. Due to decompression overhead, and the BVH over coarser (5×5) sub-grids, the rendering performance of our lossy grid approach reduces slightly to $>0.94\times$ of Embree's uncompressed grids performance. The *Ground* scene has the highest number of triangle intersections per ray and therefore the lowest performance with lossy grids, while the *Canyon* scene has the lowest intersections per ray and the highest performance. Both grid approaches reduce the memory footprint which leads to a reduction of last-level cache misses. This offsets the higher intersection costs and allows for even reaching or surpassing the performance of triangle meshes for the larger scenes.

As our grid compression scheme is mostly lossy, 84% of the vertices are compressed while 16% are kept in uncompressed form, the decompressed geometry will be different than the original uncompressed geometry. However, displacements from the base primitive rarely extend far in world space which means our local compression scheme introduces just a small error (see Table 3).

Scene	Ground	Canyon	Torus
RMS	0.00009	0.0005	0.0003
Diag(%) avg/max	0.1/4.1	0.06/4.4	0.3/6.3

Table 3: RMS *error over all vertices.* Diag *error is computed per sub-grid and shows relative vertex error with respect to the length of sub-grid's bounding box diagonal. The average error per sub-grid is* $<0.3\%$ *and the maximum over all sub-grids is between* $4.1-6.3\%$.

8. Conclusion

We have proposed a lossy compressed representation for grid-based triangle meshes which reduces memory consumption including BVH data to 5.2-6.6 bytes per triangle. Ray-tracing performance is only slightly reduced and, as vertex decompression is consistent along shared borders, watertight ray intersection is guaranteed.

9. Future Work

Memory consumption could be further reduced for the sub-grid data structure (see Listing 1) as the current version is designed to be self-contained with no references to external data. However, the bilinear patch control points are shared by up to 4 adjacent patches and could be therefore indirectly referenced.

Acknowledgement

Canyon scene is courtesy of Bartosz Domiczek.

References

[BWWA18] BENTHIN C., WALD I., WOOP S., ÁFRA A. T.: Compressed-leaf Bounding Volume Hierarchies. In *High-Performance Graphics* (2018), pp. 6:1–6:4. 1

[CDE*14] CIGOLLE Z. H., DONOW S., EVANGELAKOS D., MARA M., MCGUIRE M., MEYER Q.: A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT) 3*, 2 (April 2014), 1–30. 2

[Kee14] KEELY S.: Reduced Precision for Hardware Ray Tracing in GPUs. In *High-Performance Graphics* (2014), p. 29–40. 1

[KMKY10] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (2010), 273–286. 1

[MW04] MAHOVSKY J., WYVILL B.: Memory Conserving Bounding Volume Hierarchies with Coherent Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics (submitted)* (2004). 1

[SE10] SEGOVIA B., ERNST M.: Memory Efficient Ray Tracing with Hierarchical Mesh Quantization. In *Graphics Interface* (2010), pp. 153–160. 1

[VAMS16] VAIDYANATHAN K., AKENINE MÖLLER T., SALVI M.: Watertight Ray Traversal with Reduced Precision. In *High-Performance Graphics* (2016), pp. 33–40. 1

[WBW13] WOOP S., BENTHIN C., WALD I.: Watertight Ray/Triangle Intersection. *Journal of Computer Graphics Techniques 2*, 1 (2013), 65–82. 3

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics 33* (2014). 4

[YKL17] YLITIE H., KARRAS T., LAINE S.: Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *High-Performance Graphics* (2017), pp. 4:1–4:13. 1, 3, 4