GPU Ray Tracing of Triangular Grid Primitives

Max von Buelow

Technical University of Darmstadt, Germany

Abstract

Triangular grid primitives are a technique used to handle memory-intensive meshes more efficiently. They are also referred to as micro meshes in recent proprietary hardware implementations. This representation can reduce the memory footprint during ray tracing of subdivision surfaces or displacement maps that may result from mesh simplification. This paper presents a novel approach to accelerate GPU software ray tracing using a two-level bounding volume hierarchy (BVH) to store vertices in a non-redundant manner. The primary goal is to make the technology more accessible by focusing on standard GPU devices. The bottom-level BVH strictly follows the subdivision recursion, allowing for the side effect of rendering intermediate recursion depths. Our approach enables us to encode geometry and BVH using approximately 6.3 bytes per triangle, reducing standard representations by a factor of 4.5. Additionally, the construction time of the BVH is reduced. Our data structure achieves a peak performance impact of 16 % for a three-level subdivision.

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing;*** *Graphics processors; Mesh geometry models;* • ***Theory of computation*** → ***Data compression;***

1. Introduction

Ray tracing is an important rendering technique that transforms arbitrary scene descriptions, such as triangle meshes, into an image. It has practical applications in computer-aided design, virtual reality, games, and the movie industry. Ray tracing has become increasingly popular in the real-time domain, replacing classical rasterization due to its superior capabilities in physically correct global illumination approximation [PJH17]. Additionally, specialized GPU hardware has embedded ray tracing in recent years [PBD*10]. Although high quality scenes can currently be rendered in real time, memory reduction advancements can be utilized to increase the density of a mesh and compensate for more detail in the output image.

Structured primitives, such as triangular grid primitives, can be used in areas where static geometry can be assumed within a coarse domain, such as subdivision surface representations or displacement maps converted and discretized to the vertex domain. The latter can be derived either from measured data or from previous simplification steps, as in micro mesh rendering. Although simplifying complex structures is an active research topic, our focus is on rendering them using ray tracing. Therefore, we will use traditional subdivision surfaces or displacement maps as input data. Converting from micro meshes will be straightforward. While recent works implement these primitives [BP23; MMT23; KSW21], none of them focus on a pure software implementation that uses the GPU as a ray tracing device. One downside of these hardware implementations is that they require newer hardware, which may limit accessibility for older or less specialized devices.

In this paper we present a data structure optimized for ray tracing triangular grid primitives on the GPU. The main idea is to reduce the memory footprint of connectivity data by exploiting the internal structure of the grid [vBKF23]. This is achieved by distinguishing between interior and boundary vertices of a grid primitive and referencing them using a two-level static indexed triangle list, ensuring that no vertex needs to be stored twice. This basic structure is accelerated by a secondary level BVH using bounding spheres built around the geometry after subdivision recursion. The resulting BVH is stored semi-implicitly in memory, so that only little additional memory is required and its construction is computationally negligible. Our data structure is very easy to implement because it exploits the structure of the subdivision recursion tree.

In summary, our contributions are:

- A joint geometry and BVH data structure for triangular grid primitives that targets software GPU ray tracers that achieves a small memory footprint of approximately 6.3 bytes per triangle for four-level subdivisions.
- Non-redundant storage of vertices shared between grid primitives using an advanced, but computationally efficient to compute two-level indexing scheme.
- Increased availability due to a simple design that is easy to implement on arbitrary GPU architectures.

Related Work GPU-accelerated ray tracing has been a well-studied topic in computer graphics for quite some time. Initial approaches by APPEL [App68] and WHITTED [Whi79] have been successively improved to allow more efficient and realistic rendering of different types of scenes.

Basic memory footprint reduction of ray tracers has been explored in recent years, usually trying to improve runtime performance [VSM*18; AL09] by compressing the BVH or using BVH recursion for compression. This was done in a lossless manner with respect to the output image, either by implicitly accessing the acceleration structure [EBGM12], compressing the leaves [BWWÁ18], or quantizing the BVH in a hierarchical fashion [SE10; WMZ22]. BENTHIN, BOULOS, LACEWELL, and WALD [BBLW07] presents on-the-fly surface traversal of subdivision on GPUs, which can be seen as a compression technique for meshes to be subdivided. The latter has the disadvantage that it is not feasible for the GPU, since on-the-fly operations are computationally expensive and geometry caches would lead to further divergence on the GPU.

Structured mesh privitives [SB87], however, target the topology of the mesh data structure itself by assuming a fixed connectivity in the base geometry. The basic idea of using coarse mesh representations that are refined at render time is very old and is called displacement mapping [Coo84]. For structured grid primitives, quad meshes have traditionally been used as the base geometry, and BENTHIN, VAIDYANATHAN, and WOOP [BVW21] present a lossy CPU ray tracer capable of processing such a representation. They measure a 15 % increase in rendering time. The rasterization engine *Nanite* [KSW21] enables rendering of triangular grid primitive meshes and includes a prior mesh simplification step to retreive this representation from a high-resolution mesh. This process is commonly referred to in the industry as *micro mesh* ray tracing. BENTHIN and PETERS [BP23] port this idea to the GPU, making heavy use of NVIDIA's specialized ray tracing hardware by building second-level BVHs on-the-fly in the format expected by the hardware. Recently, NVIDIA implemented the whole idea completely in hardware [NVI; MMT23].

Unfortunately, none of these works targeting structured triangular grid primitives can be used on standard GPUs without proprietary ray tracing hardware, making them expensive to use and dependent on a single vendor. Our work takes the basic idea of triangular grid primitive ray tracing and presents a simple data structure that can be implemented on arbitrary standard computing devices, but is optimized for standard GPU architectures.

2. Ray Tracing of Triangular Grids

As the name suggests, triangular grids are seen as primitives in our ray tracer, similar to standard triangles in a ray tracing pipeline. Consequently, we use a two-level BVH, where the top-level BVH is built on top of the list of primitives and the bottom-level BVH resides inside a primitive to speed up traversal of the subdivision geometry. The geometric structure of our grid primitive data structure is largely based on the recursive idea of subdivision surfaces and can be seen in fig. 1, while the vertices define the geometry and the connectivity is implicitly given by the static structure of a grid. For the lowest level BVH, we construct bounding spheres around each of the four triangles in each subdivision level. In the following, section 2.1 describes the data structure itself and its construction in detail, and section 2.2 focuses on its traversal.

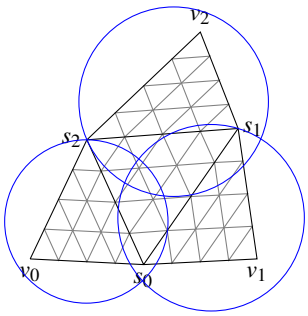
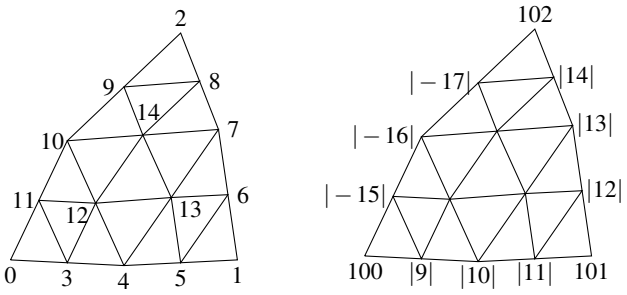


Figure 1: *Two-dimensional sketch of the geometric structure for three subdivisions. The blue circles indicate the bounding spheres for the first subdivision. Gray triangles indicate further subdivisions without spheres. The bounding sphere of the center triangle is omitted for a less cluttered visualization.*



(a) *Preliminary vertex indexing* **(b)** *Translated boundary indices*

Figure 2: *Preliminary vertex indexing and an example of boundary and corner index translation for a two-level subdivision. Final indices are computed given the grid primitive tupel* $(100, 101, 102, 6, 6, -26)$ *and eq.* (3).

2.1. Data Structure

BVH The main idea of our bottom-level hierarchy is to use semi-implicitly derived spheres from triangles as enclosing volumes and to align the BVH with the recursive subdivision aspect. We use the following formulas to compute the sphere center p_t and radius r_t, where (a_t, b_t, c_t) are the triangle vertices and V_t are all vertices of a tringle t and its recursively traversed children in the bottom-level BVH as seen in fig. 1.

$$p_t = 1/3\,(a_t + b_t + c_t) \tag{1}$$

$$r_t = \max_{\hat{v}_i \in V_t} ||\hat{v}_i - p_t||_2 \tag{2}$$

While p_t can be trivially computed on-the-fly while traversing the BVH, we only need to explicitly store r_t. A node of our BVH thus consists of a single floating-point value representing the sphere

radius (lst. 1). While definitions of spherical bounding volumes with narrower radii exist, initial experiments showed us that their center would be undesirably expensive to compute during traversal on the GPU. We implement the BVH construction in an iterative bottom-up manner, successively refining radii from leaves to the inner BVH root, which is overall much more efficient than standard SAH in deep BVH levels.

All radii are stored in memory in a breadth-first search (BFS) traversal order, which ensures that equal levels of detail are within the same region of memory. In addition, since all subtrees have the same subdivision depth, the BVH is a complete tree, making implicit indexing trivial (see section 2.2).

Looking ahead to our evaluation in section 3.4, we have evaluated that it is optimal to exclude the last two layers containing 16 triangles from the BVH in order to take advantage of the trade-off between intersecting unnecessary triangles and the additional sphere intersection overhead [MB90]. For a subdivision depth of three, fig. 1 visualizes that only one BVH level remains, which strictly speaking corresponds to a set of bounding volumes.

Geometry The geometry is largely based on an indexed triangle list with some modifications to improve runtime performance on GPUs. First, due to the structure of each triangular grid primitive, we can naturally assume that each grid has the same connectivity. This allows us to store vertex indices for each triangle in the GPU's constant memory, making them almost as fast to access as registers. We extend this simple scheme to prevent vertices of grid boundaries and corners from being encoded multiple times by introducing a secondary indexing layer that translates from *preliminary indices* (as visualized in fig. 2a) to *final indices*. Preliminary indices are referenced in the constant memory indexed triangle list and translated into final indices in the corresponding vertex buffer at runtime. First, the vertices of each grid corner are defined to have preliminary indices $C \in \{0,1,2\}$. Given the number of vertices on a boundary (excluding corners) $n_b = 2^l - 1$, the following preliminary indices $B_0 \in [3, 3+n_b)$, $B_1 \in [3+n_b, 3+2\cdot n_b)$, and $B_2 \in [3+2\cdot n_b, 3+3\cdot n_b)$ represent boundary vertices for each of the three boundaries. Given this semantic, we can easily map these preliminary indices to separate storage locations shared by grids residing in a global address space. Therefore, we explicitly store the base pointer c to each corner and b for continuously stored vertices of a boundary as the following tupel $(c_0, c_1, c_2, b_0, b_1, b_2)$. Listing 1 shows how a tupel explicitly represents a grid in memory. Preliminary indices of inner vertices follow boundary indices and can be translated completely implicitly, because each grid has the same number of inner vertices, which are used exclusively by one grid and stored continuously in a separate memory region. Because vertices of adjacent boundaries are encoded in reverse, stored boundary pointers can be negative in such a way that the absolute value computes the correct *final index* a of the desired vertex as follows and is visualized in fig. 2b:

$$a = |b_j + B_j| \tag{3}$$

The idea of encoding adjacent offsets as negative offsets using the absolute value function avoids further special cases and additional explicitly stored indicators for these cases.

Listing 1: *Our data structure used to encode a BVH node represented only by the sphere center and a grid primitive.*

```
typedef float BottomLevelBVHNode;
struct GridPrimitive {
        uint32_t c0, c1, c2; // corners
        int32_t b0, b1, b2; // boundaries
};
```

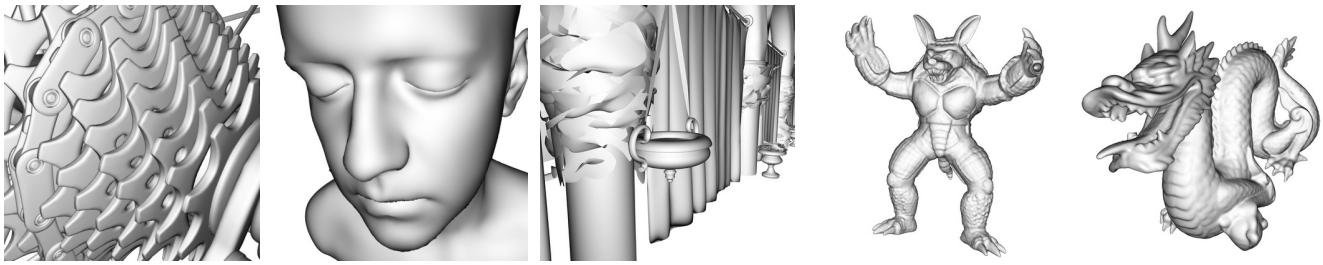
Top-Level BVH We construct a binary BVH on the set of grid primitives using the *surface area heuristic* (SAH) [MB90] and store it in a manner similar to the work of WALD, SLUSALLEK, BENTHIN, and WAGNER [WSBW01], analogous to our reference implementation further described in section 3.1. Its node structure in memory is visualized in fig. 4.

2.2. Traversal

The traversal of our structure is based on the *while-while* approach [AL09], which we implement slightly differently for the two-level BVH. The first inner while loop traverses the top-level hierarchy and the second traverses the bottom-level hierarchy and its leaves, rather than separating between inner and leaf nodes. Other configurations, such as three separate while loops for bottom-level, top-level, and leaves, proved less efficient and resulted in less device occupancy. The top-level hierarchies are traversed as usual.

Bottom-level hierarchy traversal begins with loading the tupel of the grid, which contains the corner and boundary pointers, from memory. Additionally, the six preliminary vertex indices for the first subdivision level consisting of four triangles are fetched from constant memory and translated as described in section 2.1. From these triangles, the traversal implementation computes the bounding spheres to intersect in order to decide whether to continue traversing a child. Each intersection of a bounding sphere results in an intersection interval $[t_0, t_1]$. Our tracer then pushes a t_0-sorted list of intersected spheres onto the traversal stack, making sure that the near geometry is tested first, and continues traversing until the maximum traversal depth is reached. In this case, we perform standard triangle intersection tests [MT97] instead of sphere intersections. The traversal state needs only one variable besides the stack and the current interval to keep track of the current node. Calculation of child node offsets can be done implicitly because of the complete BFS layout. The first of the four children of the current node n is at position $c_0 = 4\cdot n + 1$, followed by $c_1 = c_0 + 1$, $c_2 = c_0 + 2$, and $c_3 = c_0 + 3$. Note, that the tree is indexed in BFS layout, while it is traversed in depth first search (DFS) order as predefined by the stack data structure.

Level of Detail Our proposed data structure makes it possible to treat all inner nodes as leaf nodes by performing triangle intersection tests directly on the intermediate subdivision. Together with the BFS indexing, a very basic level of detail functionality can be easily implemented by simply reducing the maximum hierarchy depth during traversal. Although this functionality exists, it has two limitations, which we discuss further in section 4.



(a) *Bike* **(b)** *Head* **(c)** *Sponza* **(d)** *Armadillo* **(e)** *Dragon*

Figure 3: *Meshes used in our evaluation rendered at the used viewport.*

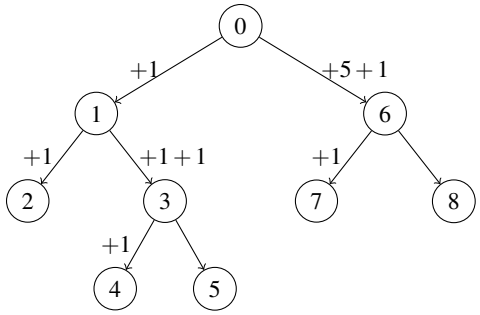


Figure 4: *The traversal order of the BVH in memory. Nodes contain the address of a node, and edges are labeled with the address offset from the parent. Left childs are addressed completely implicitly by adding one to the parent address. Right childs require the recursive size of the left child, which must be stored explicitly (5 for node 0 and 1 for node 1).*

3. Results

3.1. Reference Implementation

We give a detailed explanation of our reference here to make our results more transparent and reproducible. Our reference implementation is designed to be a strict subset of our specialized ray tracing implementation for triangular meshes, which mostly shares the implementation for the top-level BVH. This allows us to effectively measure differences in runtime performance and memory consumption introduced by our specialized techniques.

BVH We use a binary BVH constructed recursively using SAH [MB90] with 256 bins and stop object partitioning when leaves contain no more than 16 triangles, which we found to be optimal on our test device. These 16 triangles must be tested sequentially, but the overhead of intersecting more bounding boxes by increasing the BVH depth would be higher [MB90]. Each leaf is encoded as potentially padded 16 indexed triangles so that our implementation can address them implicitly. We have evaluated that an additional indexing layer to prevent padding results in a less than optimal memory footprint. Nodes are encoded with a 32 bit signed integer value in memory that represents the local offset o_i to its right child. The sign distinguishes between leaf and inner nodes in the BVH. This offset can be thought of as the size of the left subtree that may need to be skipped during traversal. Left children are encoded directly after their parent node, which strictly follows a DFS traversal order [WSBW01] and allows implicit decent into the left children. The advantage of this representation is that it requires little memory and is fast to traverse [WSWG13]. We have visualized this compact structure in fig. 4. The traversal implements the *while-while* approach [AL09]. It must keep track of the index to the current node n_i and the next leaf l_i. Descent to the left node is done by incrementing $n_l = n_i + 1$, to the right by calculating $n_r = n_i + 1 + o_i$. The index of the next left leaf does not change during node decent, and the index of the next right leaf is computed by the number of leaves in the left sub-tree $l_r = l_i + 1/2(o_i + 1)$. As mentioned before, leaf nodes are denoted by negative o_i values and obviously do not contain any further subtrees, so a local offset is unnecessary. We use $n_i = -o_i - 1 \leq 16$ to encode how many triangles are actually encoded in a leaf to avoid unnecessary intersections on dummy triangles of padded memory regions.

Geometry Our geometry representation is similar as in traditional graphics pipelines. Vertex positions and other attributes like normals are encoded in two different buffers to increase cache hit rates, since only vertex positions are needed during BVH traversal. A triangle is defined by three indices to vertices and encoded as described above. We have implemented the MÖLLER and TRUMBORE triangle intersection test [MT97].

3.2. Meshes

We evaluate the implementation of our triangle mesh data structure on several individual meshes, whose ray tracing viewport is visualized in fig. 3. Figure 3a shows a cassette of a hand-modeled *Bike* containing about 1.7 million triangles in the basic representation and 429.4 million triangles for the four-layer LOOP subdivision. Memory requirements can be calculated with fig. 6. The triangulated *Head* mesh (fig. 3b) results from a static displacement map subdivision. The mesh is hand modeled and the displacement map is derived from a 3D reconstruction of a real human head. It consists of 17 684 triangles for the root and 4.5 millions for the corresponding four-layer subdivision. The *Sponza* mesh visualized in fig. 3c is also hand-modeled and contains 262 231 root triangles and approximately 67.1 million level-four triangles. Figure 3d shows the 3D reconstructed *Armadillo* consisting of 345 944 root triangles and 88 million level-four triangles. Finally, the reconstructed *Dragon*

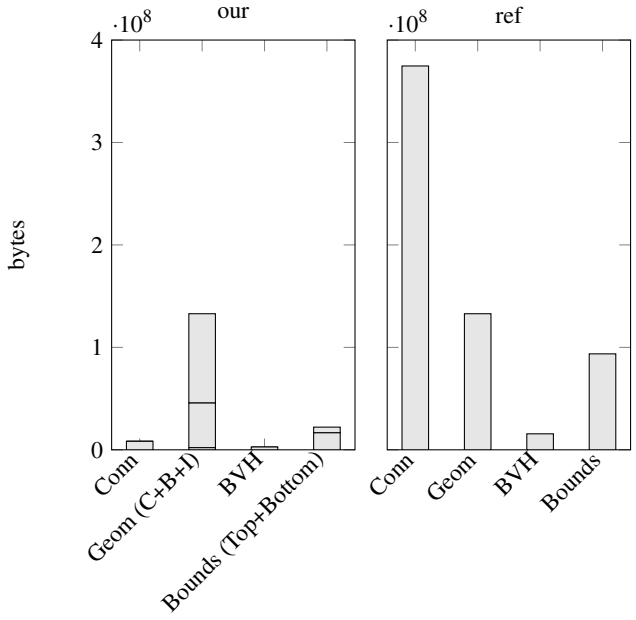


Figure 5: *Detailed memory footprint of our data structure and reference to the representative Armadillo mesh for three levels of subdivision. Some values in this graph are stacked to further distinguish between different categories. In this context, C stands for vertices at corners, B for vertices at boundaries and I for inner vertices, stacked in that order. Note that the amount of corner geometry is very small and almost invisible in the diagram.*

mesh consists of 871 414 root triangles and 223 million level-four triangles.

3.3. Memory Footprint

Figure 5 shows a detailed overview of the memory footprint produced by the implementation of our data structure compared to the footprint of the reference implementation on the Armadillo mesh. Other meshes produce plots with the same relative values. We have divided the memory footprint into four groups: the connectivity consisting of vertex indices and our grid data structure, the geometry consisting of vertices, the BVH consisting of the tree data structure (section 3.1), and the bounding boxes and bounding spheres. It is clearly observed that our data structure drastically reduces the storage overhead of the connectivity. This fact confirms the idea of grid data structures to remove unnecessary connectivity information by assuming static connectivity within a grid. The plot of our triangular grid primitive implementation also shows that boundary vertices take up about one-third of the space of the entire geometry, confirming the importance of not storing them redundantly. However, the memory usage of the geometry remains the same, confirming that our implementation has not duplicated any vertices at these mesh boundaries. The reduced memory consumption for the BVH, including its bounding volumes, results from the implicit representation of the bottom-level BVH and its drastically reduced geometric information.

The middle row of fig. 6 plots the memory requirements for each level of subdivision. Because the numbers are normalized to the total number of faces, the values of individual meshes have insignificant differences that are not apparent from the graph. We see the obvious fact that increasing the BVH level requires additional memory to store them. In all cases, our representation is more efficient in terms of memory, except for the one-level subdivision with full BVH depth. This is because our primitive tuple of section 2.1 has a large memory footprint compared to a standard indexed triangle for every four corresponding triangles resulting from the subdivision.

3.4. Run-Time Performance

We compare our triangular grid data structure to our reference implementation of section 3.1 on a *NVIDIA RTX 2080 Ti* GPU. Both approaches are implemented using the *CUDA* runtime API. The top row of Figure 6 shows the runtime performance of the ray tracers. The plots compare individual meshes, subdivision depths, and BVH depths. Unfortunately, the *Bike* has no data point for the reference implementation at four subdivision levels, due to the fact that the mesh is simply too large for the memory of our GPU device. Its memory consumption for geometry and BVH is about 11.15 GiB without the normals we need to visualize the result.

Looking at the runtime performance of each level of subdivision, the one-level subdivision has a speedup between 0 % to 28 %. For two-level subdivision, we see speedups of 10 %, ranging to impacts of 39 %. The trend continues for three levels with 16 % to 68 % performance impact and for four levels with 55 % to 85 %. This behavior is explained by the fact that our semi-implicitly constructed bottom-level BVH has increased endpoint overlap [AKL13], resulting in more BVH nodes being traversed unnecessarily during rendering.

When comparing individual meshes, our results show that meshes with a higher number of triangles visible in the viewport tend to produce more computational overhead, which was expected since more BVH nodes need to be intersected. We also see that the Head and Armadillo meshes have the highest potential of our bottom-level BVH, which can be explained by the lower number of self-intersecting bounding volumes resulting from lower mesh complexity of them.

Given these observations and the memory footprint of section 3.3, we recommend our triangular grid data structure for subdivision depths no greater than three, as the non-significantly different compression rates do not justify the greatly increased computational overhead. For larger subdivision depths, coarse subdivision levels could be triangulated and handed over to the domain of the top-level BVH during preprocessing, combining the compression rates of our approach with the increased runtime performance of the reference implementation. Regarding the BVH depth, it can be seen that deeper BVH levels do not necessarily increase the runtime performance. This is a typical behavior, as bounding volume intersections add computational overhead that must be balanced against the overhead of intersecting triangles far from the ray [MB90]. Our results suggest excluding the bottom two BVH layers from the bottom-level BVH, which also has a positive impact on memory consumption. This behavior is consistent with our reference implementation (section 3.1), where we evaluated 16 triangles

per leaf as optimal, which corresponds to two levels of subdivision consisting of 4^2 triangles.

3.5. Construction Time

The bottom row of fig. 6 plots the construction time on an *Intel Core i9-9900K* CPU with the same parameters as in the previous sections. One clear observation is that the number of BVH levels has little effect on the runtime performance of the construction procedures. This was expected, since our low-level BVH is computationally more efficient than object partitioning by an order of magnitude. The full SAH construction of the reference implementation becomes a bigger bottleneck the larger the dataset is. For a three-level sudivision on the *Bike*, our algorithm is 42 times faster than full SAH, and 383 times faster for a four-level subdivision. A final observation is that a more efficient construction and memory footprint does not directly correlate to the runtime performance of the traversal.

4. Conclusion

Efficiently structured triangular grid primitives are a promising technique for reducing the memory footprint of ray tracing implementations that already depend on similar structures. Practical applications could range from virtual environments to gaming or computer aided design. While recent work has already presented implementations, typically called *micro mesh* renderers, all of them either focus on CPU architectures or make use of heavily specialized ray tracing hardware that is not available on many devices. In this paper, we presented a novel data structure that exploits the structure of triangular grid primitives and compresses their connectivity information to a minimum compared to standard indexed triangle lists. Since our data structure implicitly preserves the subdivision recursion information, it is possible to limit to intermediate subdivision depths without regenerating the data structure, which could be useful in computationally constrained environments. Our results show that our approach achieves 6.3 bytes per triangle, reducing standard representations by a factor of 4.5 for geometry and BVH, while we are able to speed up the construction time by a factor of 42 compared to SAH in the lowest levels for a subdivision depth of three. Our data structure has a performance impact between 16 % to 68 % for the same three-level subdivision depending on the rendered mesh, but the tradeoff of losing runtime performance for more efficient storage is a common behavior.

Limitations and Future Work In the future, we would like to add lossy compression to the triangular grid primitives, as their local grouping would potentially allow further redundancy reduction through quantization. The general idea of wavelet subdivision [VP04; LQS04] stores detail vectors that refine coarse trivial-to-compute subdivisions, which are usually more efficient to quantize. While this idea cannot be used directly with our approach, as it would add unfeasible overhead to the traversal stack that needs to store decompressed intermediate recursion levels, a solution that quantizes relative to the root triangle instead of the direct parent would have more potential. While we have tried several things in this direction, current GPU architectures do not have enough registers to implement this with high device occupancy, and we leave this further extension to related work.

In terms of level of detail, our system has two limitations. The first concerns the correctness of vertices at early traversal breaks. In the context of subdivision surfaces, our approach would lead to incorrect visualizations for maximum depths lower than generated, because the LOOP subdivision scheme [Loo87] refines *all* vertices per iteration, and our rendering does not upload vertices from previous iterations to the GPU. While the visual effects appear to be small, it is generally possible to upload additional vertices from corresponding levels of subdivision. The second limitation is that our approach cannot be used adaptively without creating small cracks in the surface where the level of detail changes. Especially solving the latter with respect to our data structure would be an interesting topic for future work.

It would also be interesting to combine our implementation of triangular grid primitives with other already existing promising compression techniques like short stacks [VWB22], quantization of the BVH [SE10] or compression of BVH leaves [BWWÁ18].

Acknowledgements

The meshes used are licensed as follows. Bike is licensed under CC-BY 4.0: Yasutoshi Mori, Head is licensed under CC-BY 3.0: I-R Entertainment Ltd., Sponza is licensed under CC-BY 3.0: Frank Meinl, Crytek, and Armadillo and Dragon are Stanford Scans from Stanford University.

References

[AKL13] AILA, TIMO, KARRAS, TERO, and LAINE, SAMULI. "On Quality Metrics of Bounding Volume Hierarchies". *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. New York, NY, USA: Association for Computing Machinery, 2013, 101–107. DOI: 10.1145/2492045.2492056 5.

[AL09] AILA, TIMO and LAINE, SAMULI. "Understanding the efficiency of ray traversal on GPUs". *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*. HPG '09. ACM Press, 2009. DOI: 10.1145/1572769.1572792 2–4.

[App68] APPEL, ARTHUR. "Some Techniques for Shading Machine Renderings of Solids". *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968, 37–45. DOI: 10.1145/1468075.1468082 2.

[BBLW07] BENTHIN, CARSTEN, BOULOS, SOLOMON, LACEWELL, DYLAN, and WALD, INGO. *Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces*. 2007 2.

[BP23] BENTHIN, CARSTEN and PETERS, CHRISTOPH. "Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail". *Computer Graphics Forum* (2023). DOI: 10.1111/cgf.14868 1, 2.

[BVW21] BENTHIN, CARSTEN, VAIDYANATHAN, KARTHIK, and WOOP, SVEN. "Ray Tracing Lossy Compressed Grid Primitives". *Eurographics 2021 - Short Papers*. The Eurographics Association, 2021. DOI: 10.2312/egs.20211009 2.

[BWWÁ18] BENTHIN, CARSTEN, WALD, INGO, WOOP, SVEN, and ÁFRA, ATTILA T. "Compressed-Leaf Bounding Volume Hierarchies". *Proceedings of the Conference on High-Performance Graphics*. HPG '18. New York, NY, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3231578.3231581 2, 6.

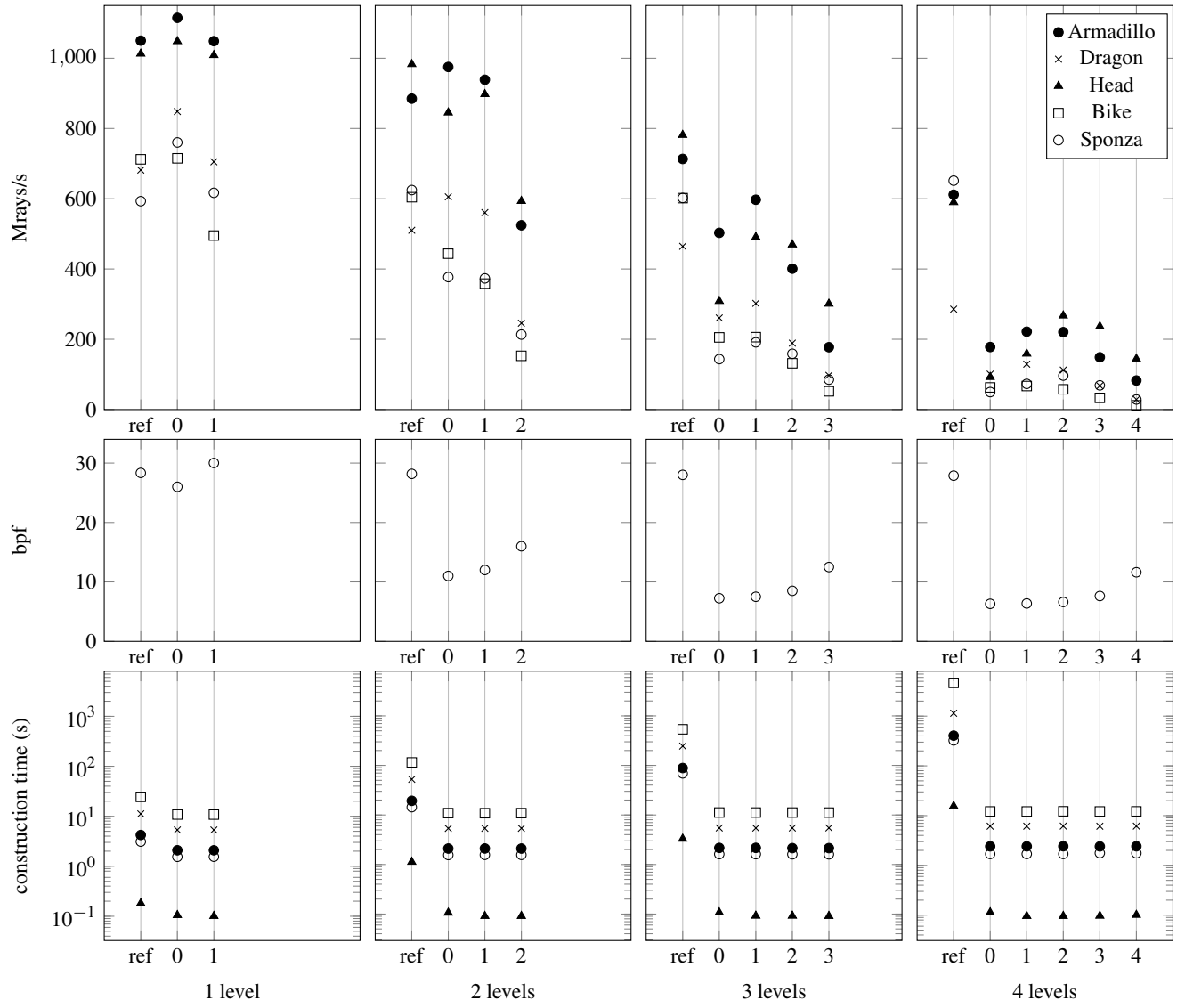


Figure 6: *The plots in the top row show the runtime performance in terms of how many millions of rays can be traced per second (Mrays/s), the plots in the middle row show the memory consumption of our data structure, and the plots in the bottom row show the construction time compared to the reference implementation for all test datasets. Each plot represents a single subdivision depth, and our data structure is further subdivided into multiple BVH depths. Each data point represents a single mesh.*

[Coo84] COOK, ROBERT L. "Shade Trees". *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), 223–231. DOI: 10.1145/964965.808602 2.

[EBGM12] EISEMANN, M., BAUSZAT, P., GUTHE, S., and MAGNOR, M. "Geometry Presorting for Implicit Object Space Partitioning". *Computer Graphics Forum* 31.4 (2012), 1445–1454. DOI: 10.1111/j.1467-8659.2012.03140.x 2.

[KSW21] KARIS, BRIAN, STUBBE, RUNE, and WIHLIDAL, GRAHAM. "A Deep Dive into Nanite Virtualized Geometry". *Advances in Real-Time Rendering in Games: Part I (proc. SIGGRAPH courses)*. 2021. URL: https://advances.realtimerendering.com/s2021/%20Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf 1, 2.

[Loo87] LOOP, CHARLES. "Smooth Subdivision Surfaces Based on Triangles". MA thesis. Jan. 1987 6.

[LQS04] LI, DENGGAO, QIN, KAIHUAI, and SUN, HANQIU. "Unlifted loop subdivision wavelets". *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.* 2004, 25–33. DOI: 10.1109/PCCGA.2004.1348331 6.

[MB90] MACDONALD, J. DAVID and BOOTH, KELLOGG S. "Heuristics for ray tracing using space subdivision". *The Visual Computer* 6.3 (May 1990), 153–166. DOI: 10.1007/bf01911006 3–5.

[MMT23] MAGGIORDOMO, ANDREA, MORETON, HENRY, and TARINI, MARCO. "Micro-Mesh Construction". *ACM Transactions on Graphics* 42.4 (July 2023), 1–18. DOI: 10.1145/3592440 1, 2.

[MT97] MÖLLER, TOMAS and TRUMBORE, BEN. "Fast, Minimum Storage Ray-Triangle Intersection". *Journal of Graphics Tools* 2.1 (1997), 21–28. DOI: 10.1080/10867651.1997.10487468 3, 4.

[NVI] NVIDIA. *Micro-Mesh Graphics Primitive For Micro Triangles*. URL: https://developer.nvidia.com/rtx/ray-tracing/micro-mesh 2.

[PBD*10] PARKER, STEVEN G., BIGLER, JAMES, DIETRICH, ANDREAS, et al. "OptiX. a general purpose ray tracing engine". *ACM Transactions on Graphics* 29.4 (July 2010), 1–13. DOI: 10.1145/1778765.1778803 1.

[PJH17] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically Based Rendering*. Elsevier, 2017. DOI: 10.1016/c2013-0-15557-2 1.

[SB87] SNYDER, JOHN M. and BARR, ALAN H. "Ray Tracing Complex Models Containing Surface Tessellations". *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, 119–128. DOI: 10.1145/37401.37417 2.

[SE10] SEGOVIA, BENJAMIN and ERNST, MANFRED. "Memory Efficient Ray Tracing with Hierarchical Mesh Quantization". *Proceedings of Graphics Interface 2010*. GI '10. CAN: Canadian Information Processing Society, 2010, 153–160 2, 6.

[vBKF23] Von BUELOW, MAX, KUIJPER, ARJAN, and FELLNER, DIETER W. "A GPU Ray Tracing Implementation for Triangular Grid Primitives". *International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments Posters and Demos*. The Eurographics Association, Dec. 2023. DOI: 10.2312/egve.20231341 1.

[VP04] VALETTE, S. and PROST, P. "Wavelet-based multiresolution analysis of irregular surface meshes". *IEEE Transactions on Visualization and Computer Graphics* 10.2 (2004), 113–122. DOI: 10.1109/TVCG.2004.1260763 6.

[VSM*18] VASIOU, ELENA, SHKURKO, KONSTANTIN, MALLETT, IAN, et al. "A detailed study of ray tracing performance: render time and energy cost". *The Visual Computer* 34.6-8 (Apr. 2018), 875–885. DOI: 10.1007/s00371-018-1532-8 2.

[VWB22] VAIDYANATHAN, K., WOOP, S., and BENTHIN, C. "Wide BVH Traversal with a Short Stack". *Proceedings of the Conference on High-Performance Graphics*. HPG '19. Goslar, DEU: Eurographics Association, 2022, 15–19. DOI: 10.2312/hpg.20191190 6.

[Whi79] WHITTED, TURNER. "An Improved Illumination Model for Shaded Display". *SIGGRAPH Comput. Graph.* 13.2 (Aug. 1979), 14. DOI: 10.1145/965103.807419 2.

[WMZ22] WALD, INGO, MORRICAL, NATE, and ZELLMANN, STEFAN. "A Memory Efficient Encoding for Ray Tracing Large Unstructured Data". *IEEE Transactions on Visualization and Computer Graphics* 28.1 (2022), 583–592. DOI: 10.1109/TVCG.2021.3114869 2.

[WSBW01] WALD, INGO, SLUSALLEK, PHILIPP, BENTHIN, CARSTEN, and WAGNER, MARKUS. "Interactive Rendering with Coherent Ray Tracing". *Computer Graphics Forum* 20.3 (Sept. 2001), 153–165. DOI: 10.1111/1467-8659.00508 3, 4.

[WSWG13] WODNIOK, DOMINIK, SCHULZ, ANDRE, WIDMER, SVEN, and GOESELE, MICHAEL. "Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays". *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013. DOI: 10.2312/EGPGV/EGPGV13/057-064 4.