Introduction

One of the core operations in a ray tracer is finding the (closest) surface along a ray. Since this operation is computationally expensive, efficient acceleration structures are crucial for high-performance rendering. During the history of 3D graphics, enormous amount of research has gone into improving the algorithms to construct and traverse them, both on a CPU and a GPU. There are several types of data structures, categorized into spatial and object hierarchies, such as uniform grids, octrees, k-d trees, and bounding volume hierarchies (BVHs). Over the last decade, BVHs have attracted increasing attention due to its combination of lower build times, predictable memory footprint, efficient incremental rebuilding and refitting techniques, and high traversal performance. To the moment, it is considered as an optimal choice for many ray tracing applications.

The BVH may be organized as a binary, quad, or k-ary tree. Originally, multi-branch BVHs were proposed in order to use (wide) SIMD hardware for efficient tracing of a single ray. In comparison with traditional packed tracing, it allows to greatly improve performance for incoherent rays. The BVH usually has a branching factor equal to the SIMD width. Naturally, there were proposed implementations for 4-wide SIMD CPUs using the SSE instruction set, such as QBVH (or Quad-BVH) [1] and MBVH [2–4]. More recently, these structures were adapted for AVX instruction set [5] and Intel MIC architecture [6]. In general, QBVH/MBVH is a lot faster on a CPU (up to 2 times in comparison with 2-ary BVH) and on old AMD HDSoxx GPUs (based on VLIW architecture). Nevertheless, on modern GPUs these structures do not provide any benefit because of the larger stack size and the extra registers used. Therefore, it is widely assumed that this kind of structures is mostly useful for CPUs and Xeon Phi.

At the same time, multi-branch BVHs have several advantages that are particularly important for GPU ray tracing. Firstly, it consumes much less memory than a regular BVH (~1.5x smaller footprint), and thus allows to reference any number of child nodes using only two 32-bit integers: ID of the first child and the number of sibling nodes (from 2 to 4). Besides better compression, this layout also improves cache locality, since child nodes are fetched together following during traversal. As for the rest, we use classical SoA data layout to access the BVH data on a GPU.

Construction and Memory Layout

For top-down construction of the BVH, we use SAH binned builder [7] producing regular BVH that is successively collapsed into 4-ary BVH [1]. Our traversal procedure exploits the special memory layout of the tree. During collapsing, all sibling nodes are placed sequentially. This allows to reference any number of child nodes using only two 32-bit integers: ID of the first child and the number of sibling nodes (from 2 to 4). Besides better compression, this layout also improves cache locality, since child nodes are fetched together following during traversal. As for the rest, we use classical SoA data layout to access the BVH data on a GPU.

GPU-Optimized Traversal with Minimal Stack Size

Our QBVH traversal implementation uses a stack to store the indices of child nodes that are intersected by a ray and are located farther. Each stacked node is visited later, when all other near child nodes have been processed. However, we place all sibling nodes that should be visited in single 32-bit stack entity. Thus, our traversal procedure requires stack of half size compared to regular BVH. Because all sibling nodes are placed sequentially, it is sufficient to store the ID of the first child and up to three offsets (from 0 to 3) of those sibling nodes that should be also visited. In our implementation, we use the lower 26 bits to encode the ID of the first child, and the remaining 2 × 3 = 6 bits contain offsets of siblings to visit. Each time when the node is fetched from the stack, we extract the next offset and shift the entire offset block 2 bits right.

Results and Discussion

Our rendering solution is integrated into OpenCASCADE technology, an open-source platform for developing CAD/CAM/CAE applications [8]. In this case study, all results have been measured using NVIDIA GeForce GTX 770 in a 1280 × 720 rendering window. QBVH provides a speedup, compared to regular BVH, of at least 30% in many tested cases for both primary and incoherent rays. Also, it requires stack of half size and ~1.3 times smaller memory footprint. However, our traversal procedure uses 26 bits for indexing BVH nodes, which results in ~22 millions of nodes and ~84 millions of triangles (5 triangles per leaf). While it is still sufficient for most of in-core scenes, the larger models require increased size of stack entity.

References