One approach to traversing QBVH on a GPU

Denis Bogolepov, Danila Ulyanov

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 HD5xxx 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 increase the maximum complexity of 3D models that can be processed in-core. Secondly, it reduces branch divergence and optimizes memory bandwidth, resulting in higher utilization of GPU's execution units, and likely better performance. Thirdly, QBVH has half the height of the binary BVH, and thus in theory can be traversed with the stack of half size, that can be crucial for complex scenes. To date, however, we have not seen such an implementation, although it would be quite interesting.

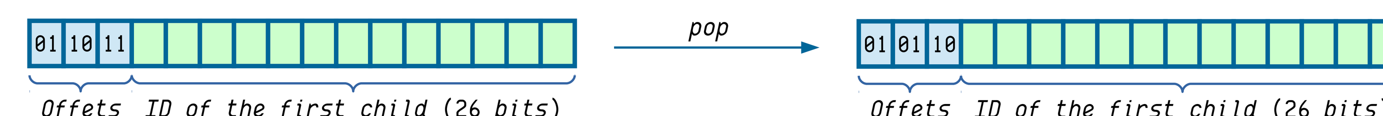
The goal of this case study is to re-evaluate the feasibility of using the QBVH/MBVH acceleration structures for GPU ray tracing by developing efficient traversal procedure requiring stack size equal to depth of 4-ary tree. Below we present the details of our QBVH implementation.

Construction and Memory Layout

For top-down construction of the QBVH, we use SAH binned builder [7] producing regular BVH that is successively collapsing 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 sidling 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 $2 \times 3 = 6$ 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 left:

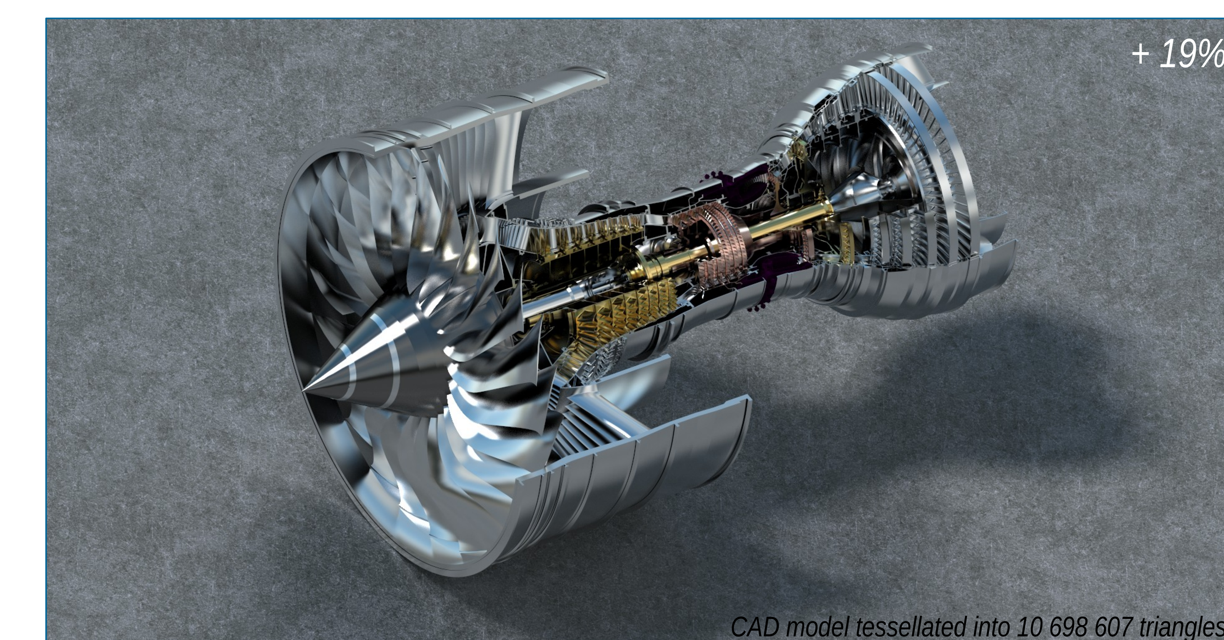


In this way, we should be able to distinguish the case when all siblings were processed and the entire 32-bit record should be removed from the stack. Since the encoding of offsets requires all possible combinations of 2 bits, it is impossible to use some kind of terminal element. Instead, we can use an observation that the offsets cannot be repeated. Thus, the stopping condition is reached when the next offset to be fetched is equal to the current one. GLSL-based pseudocode of traversal procedure:

```
int Pop (inout int head) {
  int data = Stack[head];

  int mask = data >> 26;
  int node = mask & 0x3;

  mask >>= 2;

  if ((mask & 0x3) == node) { // stopping condition
    --head;
  }
  else { // shift offset block 2 bits left
    Stack[head] = (data & 0x03FFFFFF) | ((mask | (mask << 2) & 0x30) << 26);
  }

  return (data & 0x03FFFFFF) + node;
}

void Traverse() {
  for (int node = root; /* true */;) {
    if (IsInner (node)) {
      vec4 hitTimes; // contains intersection time with each child node or MAXFLOAT
                     // constant if the child was not intersected or does not exist

      ivec4 children = ivec4 (0, 1, 2, 3);  // sort sub-nodes by intersection times

      children.xy = hitTimes.y < hitTimes.x ? children.yx : children.xy;
      hitTimes.xy = hitTimes.y < hitTimes.x ? hitTimes.yx : hitTimes.xy;
      children.zw = hitTimes.w < hitTimes.z ? children.wz : children.zw;
      hitTimes.zw = hitTimes.w < hitTimes.z ? hitTimes.wz : hitTimes.zw;
      children.xz = hitTimes.z < hitTimes.x ? children.zx : children.xz;
      hitTimes.xz = hitTimes.z < hitTimes.x ? hitTimes.zx : hitTimes.xz;
      children.yw = hitTimes.w < hitTimes.y ? children.wy : children.yw;
      hitTimes.yw = hitTimes.w < hitTimes.y ? hitTimes.wy : hitTimes.yw;
      children.yz = hitTimes.z < hitTimes.y ? children.zy : children.yz;
      hitTimes.yz = hitTimes.z < hitTimes.y ? hitTimes.zy : hitTimes.yz;

      if (hitTimes.x != MAXFLOAT) { // if at least one intersected child

        int hitMask = (hitTimes.w != MAXFLOAT ? children.w : children.z) << 2
                    | (hitTimes.z != MAXFLOAT ? children.z : children.y);

        if (hitTimes.y != MAXFLOAT) // if more than one intersected child
          Stack[++head] = FirstChild(node) | (hitMask << 2 | children.y) << 26;

        node = FirstChild(node) + children.x;  // go to the closest child
      }
      else {
        if (head < 0)
          break;

        node = Pop (head);
      }
    }
    else {
      /* process triangles, store intersection, and go to the next node */
    }
  }
}
```

BVH	QBVH
FPS (1280x720)	
17.6	24.5
Stack size	
34	15
Memory footprint	
8.04	6.50

BVH	QBVH
FPS (1280x720)	
10.4	11.9
Stack size	
25	11
Memory footprint	
5.99	4.46



BVH	QBVH
FPS (1280x720)	
1.47	1.75
Stack size	
54	25
Memory footprint	
225.95	169.30

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 15% 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 (with 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

[1] H. Dammertz, J. Hanika, A. Keller. *Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In Proceedings of the Nineteenth Eurographics conference on Rendering (EGSR '08).*

[2] I. Wald, C. Benthin, S. Boulos. *Getting rid of packets – efficient SIMD single-ray traversal using multibranching BVHs. In Proceedings of the Eurographics Symposium on Interactive Ray Tracing, 2008.*

[3] M. Ernst, G. Greiner. *Multi bounding volume hierarchies. In Proceedings of the Eurographics Symposium on Interactive Ray Tracing, 2008.*

[4] M. Ernst. *Embree: Photo-realistic ray tracing kernels. SIGGRAPH 2011 Talk (2011).*

[5] Attila T. Áfra. *Improving BVH ray tracing speed using the AVX instruction set. In Eurographics 2011 – Posters, 2011.*

[6] C. Benthin, I. Wald, S. Woop, M. Ernst, W. R. Mark. *Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. IEEE TVCG 18, 9 (2012).*

[7] I. Wald. *On fast Construction of SAH-based Bounding Volume Hierarchies. In Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing (RT '07).*

[8] OpenCASCADE Technology. *Official site: http://dev.opencascade.org.*