An Evaluation of Multi-Hit Ray Traversal in a BVH Using Existing First-Hit/Any-Hit Kernels: Algorithm Listings and Performance Visualizations

This document provides algorithm listings for multi-hit ray traversal using both progressive insertion sort and post-traversal selection sort, as well as performance visualizations of divergence and swap counts for all eight scenes used in our evaluation.

1. Algorithm Listings

As noted in the main text, hit points must be sorted to meet the ordering constraint of multi-hit ray traversal; we explore two sorting methods: progressive insertion sort during traversal and post-traversal selection sort.

**Progressive insertion sort.** In the first approach, intersections are sorted as they are inserted into the local buffer. As outlined in Algorithm 1, the traversal stack and hit list are initialized (lines 2–3), the ray traversed (lines 4–21), and hit points inserted into the hit list (line 13).

Importantly, this algorithm follows the general form of standard BVH traversal. However, when a valid intersection is found (line 11), it is inserted into the hit list in ray-order (lines 12–13). This algorithm flows naturally from a direct application of naive multi-hit traversal, under the constraints imposed by a BVH.

This technique strives to maximize cache locality—hit points are likely already in cache during traversal. Specifically, if a hit point needs to be swapped, it will most often do so with its closest neighboring hit points. Thus, when a valid intersection is found, adjacent hit points will likely be valid in cache and can thus be swapped without the penalty of global memory latency.

**Post-traversal selection sort.** In the second approach, intersections are gathered into the local buffer without regard for proper front-to-back ordering. As seen in Algorithm 2, the traversal process (lines 2–21) is nearly identical to that of Algorithm 1. However, a valid hit point is simply appended to the local buffer (line 13), and the entire collection is sorted after traversal is complete but before returning to the client (line 22).

This approach strives to keep both traversal and sorting operations amenable to SIMD processing. During traversal, rays within a SIMD vector may stall because of
Algorithm 1 Multi-hit ray traversal with progressive sort.

1: function TRAVERSE(root, ray)
2:     INITIALIZE(travStack, hitList)
3:     PUSH(travStack, root)
4:     while !EMPTY(travStack) do
5:         node ← POP(travStack)
6:         if !INTERSECT(node, ray) then
7:             continue
8:         end if
9:         if ISLEAF(node) then
10:            for triangle in node do
11:                if INTERSECT(triangle, ray) then
12:                    hitData ← (t, u, v, tID, ...)
13:                    INSERT(hitList, hitData)
14:                end if
15:            end for
16:            continue
17:        end if
18:        far ← FARCHILD(node)
19:        PUSH(travStack, far)
20:        node ← NEARCHILD(node)
21:     end while
22:     return hitList
23: end function

neighboring rays that must find and store additional hit points. The stall period is directly proportional to the time required to insert hit-point data into the local buffer. If sorting is postponed until after traversal, the potential stall period is reduced. Furthermore, divergence among rays during traversal negatively impacts sorting coherence. As shown in Section 3 of the main text, the actual sorting process is significantly more coherent when the set of intersection points to be sorted is well-known and bounded (after ray traversal) than when this information is unknown (during ray traversal).

2. Performance Visualizations

As noted in the main text, we observe that SIMD utilization is improved by deferring sort until after traversal. Divergence of neighboring rays is an issue even with first-hit ray traversal; sorting intersections during traversal only increases the probability that rays and the corresponding sorting operations will diverge, because the number of intersections along each ray may differ. Post-traversal sort alleviates this issue and,
Algorithm 2 Multi-hit ray traversal with post-traversal sort.

as noted in Section 2 of the main text, exploits a priori knowledge of the number of hit points to sort. Visualizations of divergence are shown for all eight scenes in Figures 1 and 2.

We also see that swap counts are reduced when using post-traversal selection sort. Dense scenes, in particular, benefit the most, as more hit points are likely to be out-of-order. While swap counts do not significantly impact overall performance, we nevertheless observe a reduction in the amount of work imposed by sorting. Visualizations of swap counts are shown for all eight scenes in Figures 1 and 2.
Figure 1. Performance visualizations (1 of 2). Heatmap visualizations of swap count and divergence for our filter-based multi-hit ray traversal techniques using the color scale in Figure 1 of the main text. We observe that post-traversal selection sort reduces both swap count and divergence in each of the eight scenes used for performance evaluation.
Figure 2. Performance visualizations (2 of 2).