CHAPTER 9

Multi-Hit Ray Tracing in DXR

Christiaan Gribble
SURVICE Engineering

ABSTRACT

Multi-hit ray traversal is a class of ray traversal algorithm that finds one or more, and possibly all, primitives intersected by a ray, ordered by point of intersection. Multi-hit traversal generalizes traditional first-hit ray traversal and is useful in computer graphics and physics-based simulation. We present several possible multi-hit implementations using Microsoft DirectX Raytracing and explore the performance of these implementations in an example GPU ray tracer.

9.1 INTRODUCTION

Ray casting has been used to solve the visibility problem in computer graphics since its introduction to the field over 50 years ago. *First-hit traversal* returns information regarding the nearest primitive intersected by a ray, as shown on the left in Figure 9-1. When applied recursively, first-hit traversal can also be used to incorporate visual effects such as reflection, refraction, and other forms of indirect illumination. As a result, most ray tracing APIs are heavily optimized for first-hit performance.

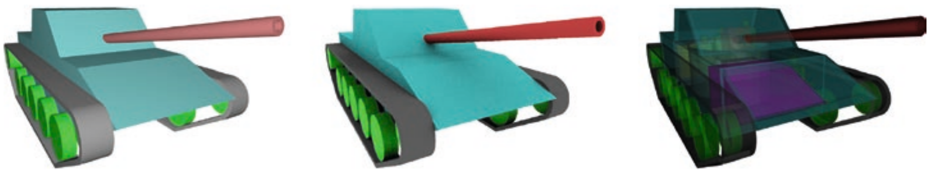

Figure 9-1. *Three categories of ray traversal. First-hit traversal and any-hit traversal are well-known and often-used ray traversal algorithms in computer graphics applications for effects like visibility (left) and ambient occlusion (center). We explore multi-hit ray traversal, the third major category of ray traversal that returns the N closest primitives ordered by point of intersection (for N ≥ 1). Multi-hit ray traversal is useful in a number of computer graphics and physics-based simulation applications, including optical transparency (right).*

© NVIDIA 2019

A second class of ray traversal, *any-hit traversal*, also finds application in computer graphics. With any-hit traversal, the intersection query is not constrained to return the nearest primitive but simply whether or not a ray intersects any primitive within a specified interval. Any-hit traversal is particularly useful for effects such as shadows and ambient occlusion, as shown in the center in Figure 9-1.

In the third class of traversal, *multi-hit ray traversal* [5], an intersection query returns information concerning the N closest primitives intersected by a ray. Multi-hit traversal generalizes both first-hit traversal (where $N = 1$) and all-hit traversal, a scheme in which ray queries return information concerning every intersected primitive (where $N = \infty$), while accommodating arbitrary values of N between these extremes.

Multi-hit traversal is useful in a number of computer graphics applications, for example, fast and accurate rendering of transparent objects. Raster-based solutions impose expensive fragment sorting on the GPU and must be extended to render coplanar objects correctly.[1] In contrast, multi-hit traversal offers a straightforward means to implement high-performance transparent rendering while handling overlapping coplanar objects correctly.

Importantly, multi-hit traversal can also be used in a wide variety of physics-based simulations, or so-called *non-optical rendering*, as shown on the right in Figure 9-1. In domains such as ballistic penetration, radio frequency propagation, and thermal radiative transport, among others, the relevant phenomena are governed by equations similar to the Beer-Lambert Law and so require ray/primitive intervals, not just intersection points. These simulations are similar to rendering scenes in which all objects behave as participating media.

A correct multi-hit ray traversal algorithm is a necessary, but insufficient, condition for modern applications; performance is also critical for both interactivity and fidelity in many scenarios. Modern ray tracing engines address performance concerns by hiding complicated, highly optimized ray tracing kernels behind clean, well-designed APIs. To accelerate ray queries, these engines use numerous bounding volume hierarchy (BVH) variants based on application characteristics provided to the engine by the user. These engines provide fast first-hit and any-hit ray traversal operations for use in applications across optical and non-optical domains, but they do not typically support multi-hit ray traversal as a fundamental operation.

[1]The problem of coplanar objects, both in transparent rendering and in physics-based simulation, is discussed more thoroughly by, for example, Gribble et al. [5]; interested readers are referred to the literature for additional details.

Early work on multi-hit ray traversal [5] assumes an acceleration structure based on spatial subdivision, in which leaf nodes of the structure do not overlap. With such structures, ordered traversal—and therefore generating ordered hit points—is straightforward: sorting is required only within, not across, leaf nodes. However, ordered traversal in a structure based on object partitioning, such as a BVH, is not achieved so easily. While an implementation based on a traversal priority queue (rather than a traversal stack) enables front-to-back traversal of a BVH [7], most publicly available, widely used production ray tracing APIs do not provide ordered BVH traversal variants.

However, these APIs, including Microsoft DirectX Raytracing (DXR), expose features enabling implementation of multi-hit ray tracing entirely with user-level code, thereby leveraging their existing—and heavily optimized—BVH construction and traversal routines. In the remainder of this chapter, we present several possible multi-hit implementations using DXR and explore their performance in an example GPU ray tracing application. Source and binary distributions of this application are available [4], permitting readers to explore, modify, or enhance these DXR multi-hit implementations.

9.2 IMPLEMENTATION

As noted in Section 9.1 and discussed in detail by Amstutz et al. [1], the problem of multi-hit ray tracing with unordered BVH traversal variants is compounded by overlapping nodes. Correctness requires either naive multi-hit traversal [5], which is potentially slow, or modification of BVH construction or traversal routines, which not only imposes potentially significant development and maintenance burdens in production environments, but is simply not possible with implementation-neutral ray tracing APIs.

To address these issues, we present two DXR implementations each of two multi-hit traversal algorithms: naive multi-hit traversal and node-culling multi-hit BVH traversal [3]. Our first implementation of each algorithm leverages DXR *any-hit shaders* to satisfy multi-hit intersection queries along each ray. DXR any-hit shaders execute whenever a ray intersects a geometry instance within the current ray interval, $[t_{\text{min}}, t_{\text{max}}]$, regardless of its position along the ray relative to other intersections. These shaders do not follow any defined order of execution for intersections along a ray. If an any-hit shader accepts a potential intersection, its hit distance becomes the new maximum value for the ray interval, t_{max}.

Our second implementation of each algorithm satisfies multi-hit queries using DXR *intersection shaders*, which offer an alternative representation for geometry in a bottom-level acceleration structure. In this case, the procedural primitive is defined by its axis-aligned bounding box, and a user-defined intersection shader evaluates primitive intersections when a ray intersects that box. The intersection shader defines attributes describing intersections, including the current hit distance, that are then passed to subsequent shaders. Generally speaking, DXR intersection shaders are less efficient than the built-in ray/triangle intersection routines, but they offer far more flexibility. We exploit these shaders to implement both naive and node-culling multi-hit ray traversal for triangle primitives as an alternative to the DXR any-hit shader implementations.

In these implementations, each shader assumes buffers for storing multi-hit results: a two-dimensional (width × height) buffer for per-ray hit counts and a three-dimensional (width × height × (N_{query} + 1)) buffer for hit records, each comprising a hit-point intersection distance (t-value), the diffuse surface color, and the value $N_g \cdot V$ to support simple surface shading operations. The any-hit shader implementations use a user-defined ray payload structure to track the current number of hits and require setting the `D3D12_RAYTRACING_GEOMETRY_FLAG_NO_DUPLICATE_ANYHIT_INVOCATION` geometry flag to disallow multiple any-hit shader invocations. The corresponding ray generation shaders set the `RAY_FLAG_FORCE_NON_OPAQUE` ray flag to treat all ray/primitive intersections as non-opaque. In contrast, the intersection shader implementations require buffers storing triangle vertices, faces, and material data, properties typically managed by DXR when using the built-in triangle primitives.

All shaders rely on utility functions for shader-side buffer management, color mapping for visualization, and so forth. Likewise, each shader assumes values controlling the final rendered results, including N_{query}, background color, and various color-mapping parameters affecting the visualization modes supported by our example application. Other DXR shader states and parameters—for example, the two-dimensional output buffer storing rendered results—are ultimately managed by Falcor [2], the real-time rendering framework underlying our application. For clarity and focus of presentation, these elements are omitted from the implementation highlights that follow.

Our example ray tracing application leverages Chris Wyman's `dxrTutors.Code` project [8], which itself builds on Falcor, to manage DXR states. The project `dxrTutors.Code` provides a highly abstracted CPU-side C++ DXR API, designed both to aid programmers in getting DXR applications running quickly and to enable easy experimentation. While these dependencies are required to build our multi-hit ray tracing application from source, the multi-hit DXR shaders themselves can be adapted to other frameworks that provide similar DXR abstractions in a straightforward manner. We highlight these implementations in the remainder of this section, and we explore the resulting performance in Section 9.3.

9.2.1 NAIVE MULTI-HIT TRAVERSAL

Any multi-hit traversal implementation returns information concerning the $N \leq N_{\text{query}}$ closest ray/primitive intersections, in ray order, for values of N_{query} in $[1, \infty]$. A first approach to satisfying such queries, naive multi-hit ray traversal, simply collects all valid intersections along the ray and returns at most N_{query} of these to the user. A DXR any-hit shader implementation of this algorithm is shown in the following listing.

```
[shader ("anyhit")]
void mhAnyHitNaive(inout mhRayPayload rayPayload,
                   BuiltinIntersectionAttribs attribs)
{
  // Process candidate intersection.
  uint2 pixelIdx  = DispatchRaysIndex();
  uint2 pixelDims = DispatchRaysDimensions();
  uint  hitStride = pixelDims.x*pixelDims.y;
  float tval      = RayTCurrent();

  // Find index at which to store candidate intersection.
  uint hi = getHitBufferIndex(min(rayPayload.nhits, gNquery),
                              pixelIdx, pixelDims);
  uint lo = hi - hitStride;
  while (hi > 0 && tval < gHitT[lo])
  {
    // Move data to the right ...
    gHitT       [hi] = gHitT       [lo];
    gHitDiffuse [hi] = gHitDiffuse [lo];
    gHitNdotV   [hi] = gHitNdotV   [lo];

    //... and try next position.
    hi -= hitStride;
    lo -= hitStride;
  }

  // Get diffuse color and face normal at current hit point.
  uint primIdx    = PrimitiveIndex();
  float4 diffuse = getDiffuseSurfaceColor(primIdx);
  float3 Ng      = getGeometricFaceNormal(primIdx);

  // Store hit data, possibly beyond index of the N <= Nquery closest
  // intersections (i.e., at hitPos == Nquery).
  gHitT       [hi] = tval;
  gHitDiffuse [hi] = diffuse;
  gHitNdotV   [hi] =
      abs(dot(normalize(Ng), normalize(WorldRayDirection())));

  ++rayPayload.nhits;

  // Reject the intersection and continue traversal with the incoming
  // ray interval.
  IgnoreHit();
}
```

For each candidate intersection, the shader determines the index at which to store the corresponding data, actually stores that data, and updates the number of intersections collected so far. Here, intersection data is collected into buffers with exactly $N_{\text{query}} + 1$ entries per ray. This approach allows us to always write (even potentially ignored) intersection data following the insertion sort loop—no conditional branching is required. Finally, the candidate intersection is rejected by invoking the DXR `IgnoreHit` intrinsic in order to continue traversal with the incoming ray interval, $[t_{\text{min}}, t_{\text{max}}]$.

The intersection shader implementation, outlined in the listing that follows, behaves similarly. After actually intersecting the primitive (in our case, a triangle), the shader again determines the index at which to store the corresponding data, actually stores that data, and updates the number of intersections collected so far. Here, `intersectTriangle` returns the number of hits encountered so far to indicate a valid ray/triangle intersection, or zero when the ray misses the triangle.

```
[shader("intersection")]
void mhIntersectNaive()
{
  HitAttribs hitAttrib;
  uint nhits = intersectTriangle(PrimitiveIndex(), hitAttrib);
  if (nhits > 0)
  {
    // Process candidate intersection.
    uint2 pixelIdx  = DispatchRaysIndex();
    uint2 pixelDims = DispatchRaysDimensions();
    uint hitStride  = pixelDims.x*pixelDims.y;
    float tval      = hitAttrib.tval;

    // Find index at which to store candidate intersection.
    uint hi = getHitBufferIndex(min(nhits, gNquery),
                                pixelIdx, pixelDims);
    // OMITTED: Equivalent to lines 13-35 of previous listing.

    uint hcIdx = getHitBufferIndex(0, pixelIdx, pixelDims);
    ++gHitCount[hcIdx];
  }
}
```

Aside from the need to compute ray/triangle intersections, some important differences between the any-hit shader and the intersection shader implementations exist. For example, per-ray payloads are not accessible from within DXR intersection shaders, so we must instead manipulate the corresponding entry in the global two-dimensional hit counter buffer, `gHitCount`. In addition, the multi-hit intersection shader omits any calls to the DXR `ReportHit` intrinsic, which effectively rejects every candidate intersection and continues traversal with the incoming ray interval, $[t_{\text{min}}, t_{\text{max}}]$, as is required.

Naive multi-hit traversal is simple and effective. It imposes few implementation constraints and allows users to process as many intersections as desired. However, this algorithm is potentially slow. It effectively implements the all-hit traversal scheme, as the ray traverses the entire BVH structure to find (even if not store) all intersections and ensure that the $N \le N_{\text{query}}$ closest of these are returned to the user.

9.2.2 NODE-CULLING MULTI-HIT BVH TRAVERSAL

Node-culling multi-hit BVH traversal adapts an optimization common for first-hit BVH traversal to the multi-hit context. In particular, first-hit BVH traversal variants typically consider the current ray interval, $[t_{\text{min}}, t_{\text{max}}]$, to cull nodes based on t_{max}, the distance to the nearest valid intersection found so far. If during traversal a ray enters a node at $t_{\text{enter}} > t_{\text{max}}$, the node is skipped, since traversing the node cannot possibly produce a valid intersection closer to the ray origin than the one already identified.

The node-culling multi-hit BVH traversal algorithm incorporates this optimization by culling nodes encountered along a ray at a distance beyond the farthest valid intersection among the $N \ge N_{\text{query}}$ collected so far. In this way, subtrees or ray/primitive intersection tests that cannot produce valid intersections are skipped once it is appropriate to do so.

Our node-culling DXR any-hit shader implementation is highlighted in the listing that follows. The corresponding naive multi-hit implementation differs from this implementation only in the way that valid intersections are handled by the shader. In the former, intersections are always rejected to leave the incoming ray interval $[t_{\text{min}}, t_{\text{max}}]$ unchanged and, ultimately, traverse the entire BVH. In the latter, however, we induce node culling once the appropriate conditions are satisfied, i.e., only after $N \ge N_{\text{query}}$ intersections have been collected.

```
[shader("anyhit")]
void mhAnyHitNodeC(inout mhRayPayload rayPayload,
     BuiltinIntersectionAttribs attribs)
{
  // Process candidate intersection.
  // OMITTED: Equivalent to lines 5-37 of first listing.

  // If we store the candidate intersection at any index other than
  // the last valid hit position, reject the intersection.
  uint hitPos = hi / hitStride;
  if (hitPos != gNquery - 1)
    IgnoreHit();

  // Otherwise, induce node culling by (implicitly) returning and
  // accepting RayTCurrent() as the new ray interval endpoint.
}
```

We also note that the DXR any-hit shader implementation imposes an additional constraint on ray interval updates: With any-hit shaders, we cannot accept using any intersection distance other than the one returned by the DXR `RayTCurrent` intrinsic. As a result, the implicit *return-and-accept* behavior of the shader is valid only when the candidate intersection is the last valid intersection among those collected so far (i.e., when it is written to index `gNquery-1`). Writes to all other entries, including those within the collection of valid hits, must necessarily invoke the `IgnoreHit` intrinsic. This DXR-imposed constraint stands in contrast to node-culling multi-hit traversal implementations in at least some other ray tracing APIs (see, for example, the implementation presented by Gribble et al. [6]), and it represents a lost opportunity to cull nodes as a result of stale t_{max} values.

However, the node-culling DXR intersection shader implementation, shown in the following listing, does not fall prey to this potential loss of culling opportunities. In this implementation, we control the intersection distance reported by the intersection shader and can thus return the value of the last valid hit among the $N \geq N_{query}$ collected so far. This is done simply by invoking the DXR `ReportHit` intrinsic with that value any time the actual intersection point is within the N_{query} closest hits.

```
[shader("intersection")]
void mhIntersectNodeC()
{
  HitAttribs hitAttrib;
  uint nhits = intersectTriangle(PrimitiveIndex(), hitAttrib);
  if (nhits > 0)
  {
    // Process candidate intersection.
    // OMITTED: Equivalent to lines 9-20 of second listing.

    // Potentially update ray interval endpoint to gHitT[lastIdx] if we
    // wrote new hit data within the range of valid hits [0, Nquery-1].
    uint hitPos = hi / hitStride;
    if (hitPos < gNquery)
    {
      uint lastIdx =
          getHitBufferIndex(gNquery - 1, pixelIdx, pixelDims);
      ReportHit(gHitT[lastIdx], 0, hitAttrib);
    }
  }
}
```

Node-culling multi-hit BVH traversal exploits opportunities for early-exit despite unordered BVH traversal. Early-exit is a key feature of first-hit BVH traversal and of buffered multi-hit traversal in acceleration structures based on spatial subdivision, so we thus hope for improved multi-hit performance with the node-culling variants when users request fewer-than-all hits.

9.3 RESULTS

Section 9.2 presents several implementation alternatives for multi-hit ray tracing in DXR. Here, we explore their performance in an example GPU ray tracing application. Source and binary distributions of this application are available [4], permitting readers to explore, modify, or enhance these multi-hit implementations.

9.3.1 PERFORMANCE MEASUREMENTS

We report performance of our DXR multi-hit ray tracing implementations using eight scenes of varying geometric and depth complexity rendered from the viewpoints depicted in Figure 9-2. For each test, we render a series of 50 warmup frames followed by 500 benchmark frames at 1280 × 960 pixel resolution using visibility rays from a pinhole camera and a single sample per pixel. Reported results are averaged over the 500 benchmark frames. Measurements are obtained on a Windows 10 RS4 desktop PC equipped with a single NVIDIA GeForce RTX 2080 Ti GPU (driver version 416.81). Our application compiles with Microsoft Visual Studio 2017 Version 15.8.9 and links against Windows 10 SDK 10.0.16299.0 and DirectX Raytracing Binaries Release V1.3.

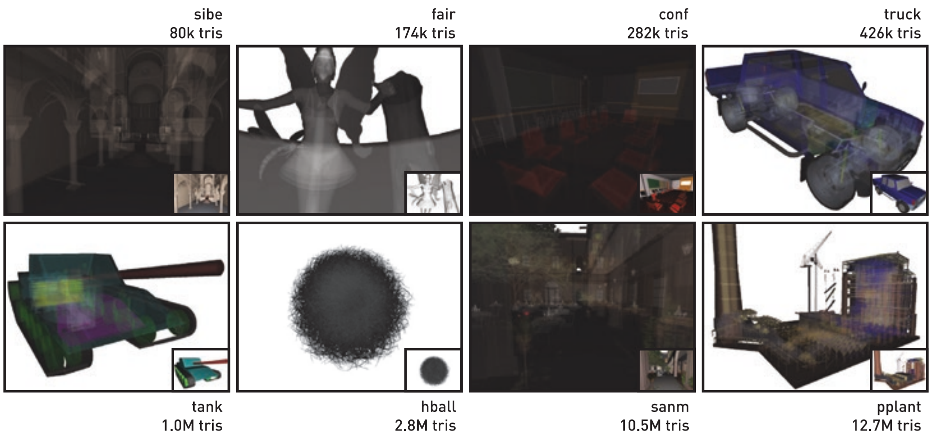

Figure 9-2. *Scenes used for performance evaluation. Eight scenes of varying geometric and depth complexity are used to evaluate the performance of our multi-hit implementations in DXR. First-hit visible surfaces hide significant internal complexity in many of these scenes, making them particularly useful in tests of multi-hit traversal performance.*

In the figures referenced throughout the remainder of this section, we use the following abbreviations to denote particular traversal implementation variants:

> *fhit:* A straightforward implementation of standard first-hit ray traversal.

> *ahit-n:* The any-hit shader implementation of naive multi-hit ray traversal.

> *ahit-c:* The any-hit shader implementation of node-culling multi-hit ray traversal.

> *isec-n:* The intersection shader implementation of naive multi-hit ray traversal.

> *isec-c:* The intersection shader implementation of node-culling multi-hit ray traversal.

Please refer to these definitions when interpreting results.

9.3.1.1 FIND FIRST INTERSECTION

First, we measure performance when specializing multi-hit ray traversal to first-hit traversal. Figure 9-3 compares performance in millions of hits per second (Mhps) when finding the nearest intersection using standard first-hit traversal against finding the nearest intersection using multi-hit traversal (i.e., $N_{\text{query}} = 1$). The advantage of node culling is clearly evident in this case. Performance with any-hit shader node-culling multi-hit BVH traversal approaches that of standard first-hit traversal (to within about 94% on average). However, the intersection shader node-culling variant performs worst overall (by more than a factor of 4×, on average), and performance with the naive multi-hit traversal variants is more than a factor of 2× to 4× worse (on average) than that with first-hit traversal for our test scenes.

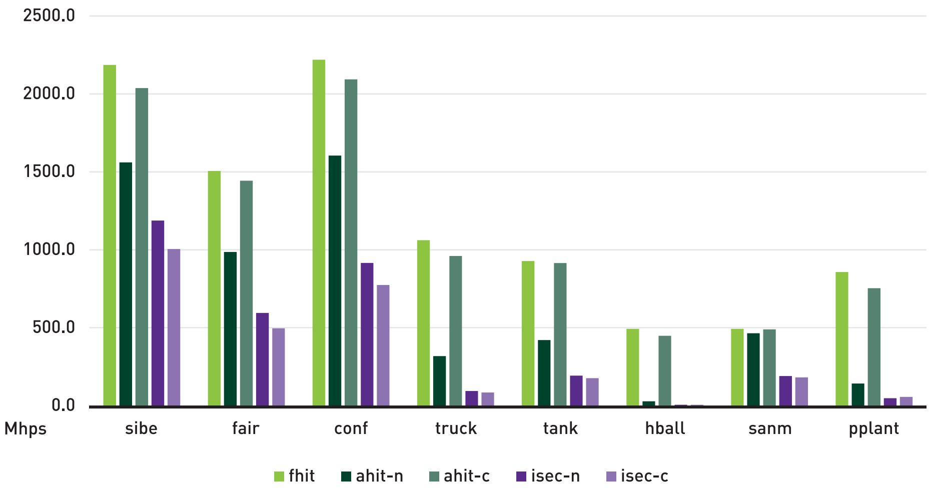

Figure 9-3. *Performance of standard first-hit and multi-hit variants for finding first intersection. The graph compares performance in millions of hits per second (Mhps) among standard first-hit traversal and our multi-hit implementations when $N_{query} = 1$.*

9.3.1.2 FIND ALL INTERSECTIONS

Next, we measure performance when specializing multi-hit ray traversal to all-hit traversal ($N_{\text{query}} = \infty$). Figure 9-4 compares performance in Mhps when using each multi-hit variant to gather all hit points along a ray. Not surprisingly, naive and node-culling variants across the respective shader implementations perform similarly, and differences are generally within the expected variability among trials.

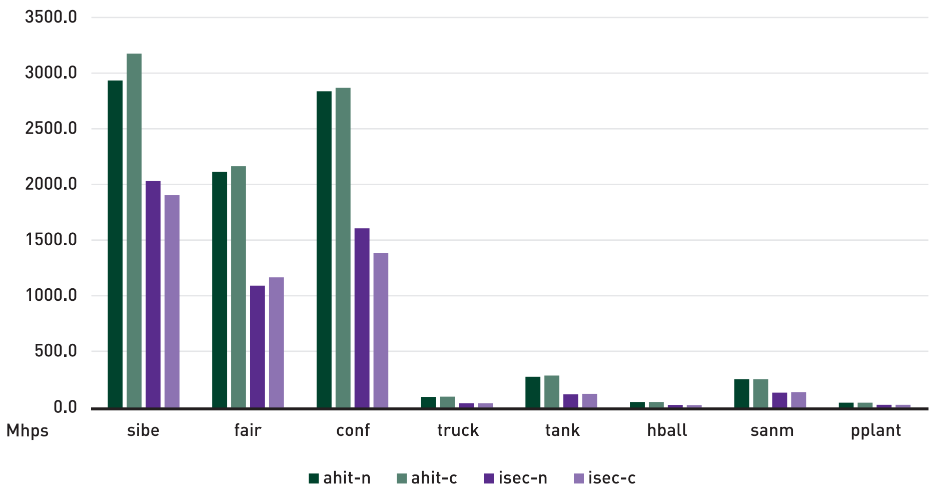

Figure 9-4. *Performance of multi-hit variants for finding all intersections. The graph compares performance in Mhps among our naive and node-culling variants when* $N_{query} = \infty$.

9.3.1.3 FIND SOME INTERSECTIONS

Finally, we measure multi-hit performance using the values of N_{query} considered by Gribble [3], which, aside from the extremes $N_{\text{query}} = 1$ and $N_{\text{query}} = \infty$, comprise 10%, 30%, and 70% of the maximum number of intersections encountered along any one ray for each scene. The find-some-intersections case is perhaps the most interesting, given that multi-hit traversal cannot be specialized to either first-hit or all-hit algorithms in this case. For brevity, we examine only results for the *truck* scene; however, the general trends present in these results are observed in those obtained with the other scenes as well.

Figure 9-5 shows performance in the *truck* scene as $N_{\text{query}} \to \infty$. Generally speaking, the impact of node culling is somewhat less pronounced than in other multi-hit implementations. See, for example, the results reported by Gribble [3] and Gribble et al. [6]. With the any-hit shader implementations, the positive impact of node culling on performance relative to naive multi-hit decreases from more than a factor of 2× when $N_{\text{query}} = 1$ to effectively zero when $N_{\text{query}} = \infty$. Nevertheless, the any-hit shader node-culling implementation performs best overall, often performing significantly better (or at least not worse) than the corresponding naive implementation. In contrast, the intersection shader implementations perform similarly across all values of N_{query}, and both variants perform significantly worse overall compared to the any-hit variants.

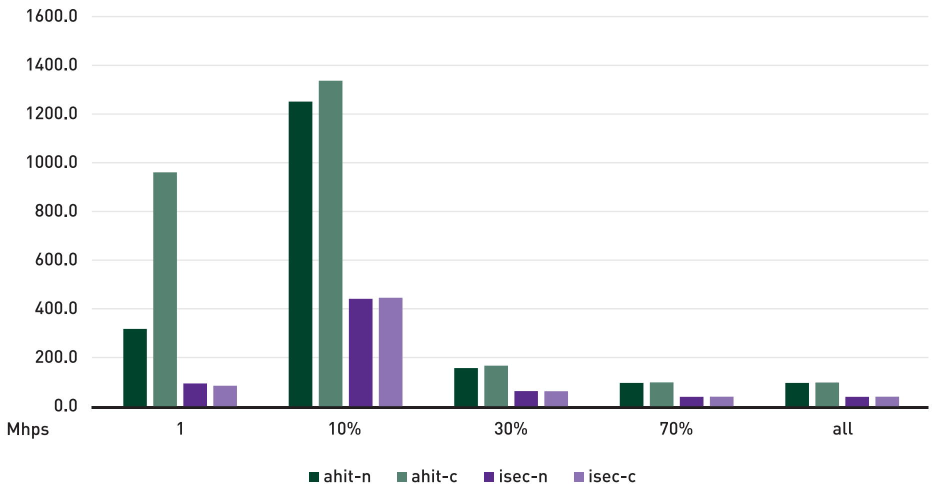

Figure 9-5. *Multi-hit performance in the truck scene. The graph compares multi-hit performance in Mhps among our multi-hit implementations for various values of N_{query}.*

9.3.2 DISCUSSION

To better understand the results above, we report the total number of candidate intersections processed by each multi-hit variant in Figure 9-6. We see that the naive multi-hit implementations process the same number of candidate intersections, regardless of N_{query}, as expected. Likewise, we see that node culling does, in fact, reduce the total number of candidate intersections processed, at least when N_{query} is less than 30%. After that point, however, both node-culling implementations process the same number of candidate intersections as the naive multi-hit implementations. Above this 30% threshold, node culling offers no particular advantage over naive multi-hit traversal for our scenes on the test platform.

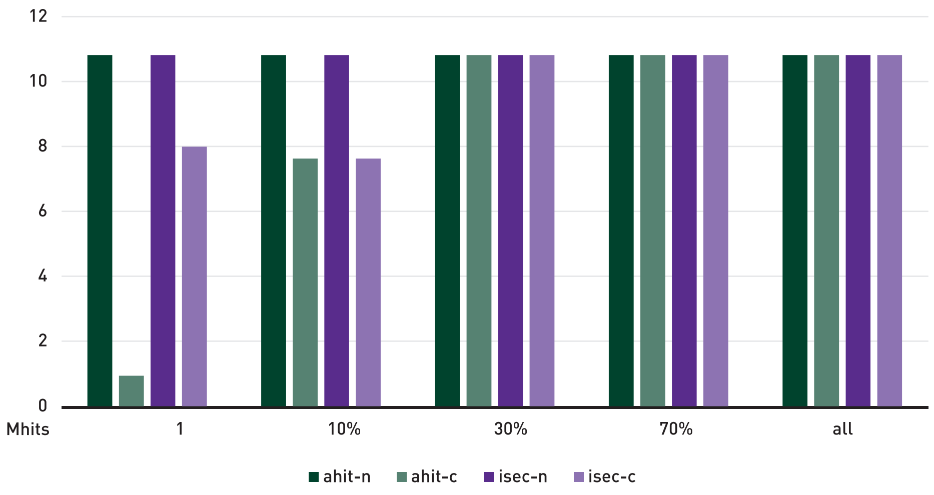

Figure 9-6. *Number of candidate intersections processed in the truck scene. The graph compares the number of candidate intersections (in millions) processed by each multi-hit implementation.*

The data in Figure 9-6 also indicates that the lost opportunity to cull some nodes with the any-hit shader variant (as discussed in Section 9.2) does not affect overall traversal behavior in practice. In fact, when observing performance across all three experiments, we see that the any-hit shader node-culling implementation outperforms the intersection shader implementation by more than a factor of 2× (on average) for all values of N_{query} considered here.

Although inefficiencies arising when implementing (the otherwise built-in) ray/triangle intersection using DXR's mechanisms for user-defined geometry may account for the large gap in performance between the node-culling multi-hit variants, the visualizations in Figure 9-7 offer some additional insight. The top row depicts the number of candidate intersections processed by each multi-hit variant for $N_{\text{query}} = 9$, or 10% of the maximum number of hits along any one ray, while the bottom row depicts the number of interval update operations invoked by each implementation. As expected, the naive multi-hit implementations are equivalent. They process the same total number of candidate intersections and impose no interval updates whatsoever. Similarly, both node-culling variants reduce the number of candidate intersections processed, with the DXR intersection shader implementation processing fewer than the any-hit shader variant (7.6M versus 8.5M). However, this implementation imposes significantly more interval updates than the any-hit shader implementation (1.7M versus 437k). These update operations are the only major source of user-level execution path differences between the two implementations. In DXR, then, the opportunity to cull more frequently in the intersection shader implementation actually imposes more work than the culling itself saves and likely contributes to the overall performance differences observed here.

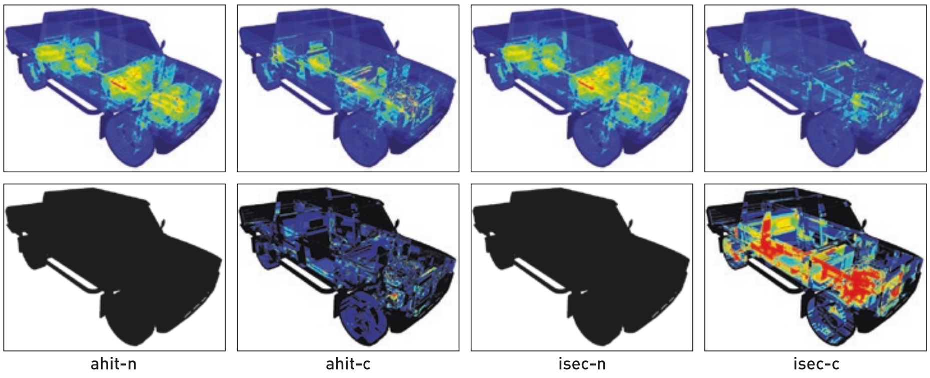

Figure 9-7. *Efficiency visualization. Heatmap visualizations using a rainbow color scale reveal that far less work must be done per ray when using node culling compared to using naive multi-hit traversal for $N_{query} = 9$ (top row). However, when comparing the node-culling variants, the potential savings due to fewer traversal steps and ray/primitive intersection tests with the intersection shader evaporate due to significantly more ray interval updates (bottom row). The costs outweigh the savings in this case.*

9.4 CONCLUSIONS

We present several possible implementations of multi-hit ray tracing using Microsoft DirectX Raytracing and report their performance in an example GPU ray tracing application. Results show that, of the implementations explored here, node-culling multi-hit ray traversal implemented using DXR any-hit shaders performs best overall for our scenes on the test platform. This alternative is also relatively straightforward to implement, requiring only a few more lines of code than the corresponding naive multi-hit traversal implementation. At the same time, the any-hit shader node-culling variant does not require reimplementation of the otherwise built-in ray/triangle intersection operations, further reducing development and maintenance burdens in a production environment relative to other alternatives. Nevertheless, we make available both source and binary distributions of all four DXR multi-hit variants in our example GPU ray tracing application [4], permitting readers to further explore multi-hit ray tracing in DXR.

REFERENCES

[1] Amstutz, J., Gribble, C., Günther, J., and Wald, I. An Evaluation of Multi-Hit Ray Traversal in a BVH Using Existing First-Hit/Any-Hit Kernels. *Journal of Computer Graphics Techniques 4*, 4 (2015), 72–90.

[2] Benty, N., Yao, K.-H., Foley, T., Kaplanyan, A. S., Lavelle, C., Wyman, C., and Vijay, A. The Falcor Rendering Framework. `https://github.com/NVIDIAGameWorks/Falcor`, July 2017.

[3] Gribble, C. Node Culling Multi-Hit BVH Traversal. In *Eurographics Symposium on Rendering* (June 2016), pp. 22–24.

[4] Gribble, C. DXR Multi-Hit Ray Tracing, October 2018. http://www.rtvtk.org/~cgribble/research/DXR-MultiHitRayTracing. Last accessed October 15, 2018.

[5] Gribble, C., Naveros, A., and Kerzner, E. Multi-Hit Ray Traversal. *Journal of Computer Graphics Techniques 3*, 1 (2014), 1–17.

[6] Gribble, C., Wald, I., and Amstutz, J. Implementing Node Culling Multi-Hit BVH Traversal in Embree. *Journal of Computer Graphics Techniques 5*, 4 (2016), 1–7.

[7] Wald, I., Amstutz, J., and Benthin, C. Robust Iterative Find-Next Ray Traversal. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018), pp. 25–32.

[8] Wyman, C. A Gentle Introduction to DirectX Raytracing, August 2018. Original code linked from http://cwyman.org/code/dxrTutors/dxr_tutors.md.html; newer code available via https://github.com/NVIDIAGameWorks/GettingStartedWithRTXRayTracing. Last accessed November 12, 2018.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.