Multi-Hit Ray Traversal

Christiaan Gribble
Applied Technology Operation
SURVICE Engineering

Alexis Naveros
Applied Technology Operation
SURVICE Engineering

Ethan Kerzner
SCI Institute
University of Utah

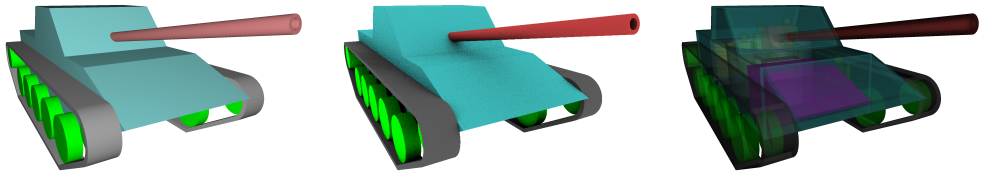


Figure 1. Three categories of ray traversal. *First-hit* traversal and *any-hit* traversal are well-known and often-used ray traversal algorithms in computer graphics applications for effects like visibility (left) and ambient occlusion (center). We introduce *multi-hit* traversal as the third major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

Abstract

Multi-hit ray traversal is a class of ray traversal algorithms 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 introduce an efficient algorithm for ordered multi-hit ray traversal, investigate its performance in a GPU ray tracer, and demonstrate two problems easily solved with our algorithm.

1. Introduction

Ray casting has been used to solve the visibility problem in computer graphics since its introduction to the field some 45 years ago. Such *first-hit* traversal returns in-

 ISSN 2331-7418

formation regarding the nearest primitive intersected by a ray, as shown in the left panel of Figure 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 engines are heavily optimized for first-hit performance.

A second class of ray traversal, *any-hit* traversal, has also received some attention. 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 panel of Figure 1.

A third class of traversal, *multi-hit* ray traversal, has received far less attention. In this case, the intersection query returns information concerning the N closest primitives intersected by a ray. We observe that 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 is an open problem in computer graphics. Current raster-based solutions impose expensive fragment sorting on the GPU [Maule et al. 2011]. Furthermore, these techniques must be extended to render coplanar objects correctly [Vasilakis and Fudos 2012]. Ray-traced transparency has also received some attention. For example, Stephens et al. [2006] use transparent rendering to enhance spatial context when inspecting large engineering CAD models in a ray-based visualization system. Similarly, Ize and Hansen [2011] consider attenuated occlusion, in which case a valid ray/primitive intersection does not necessarily terminate traversal. While these works feature ray-traced transparency, they do not provide either algorithm details or performance metrics, nor do they address the problem of coplanar objects. Our multi-hit traversal algorithm provides 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 in the right panel of Figure 1. In fact, domains such as ballistic penetration, radio frequency propagation, and thermal radiative transport, among others, motivate this work. In these domains, the interesting 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.

An enticing solution with first-hit traversal is to simply "continue" tracing with a new ray using the recently-generated hit point, adjusted by a small ε term, as its origin.

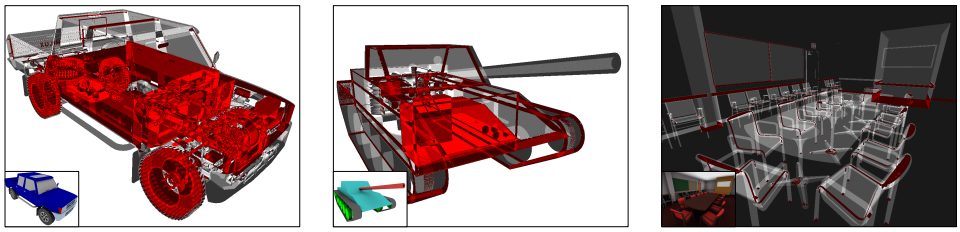


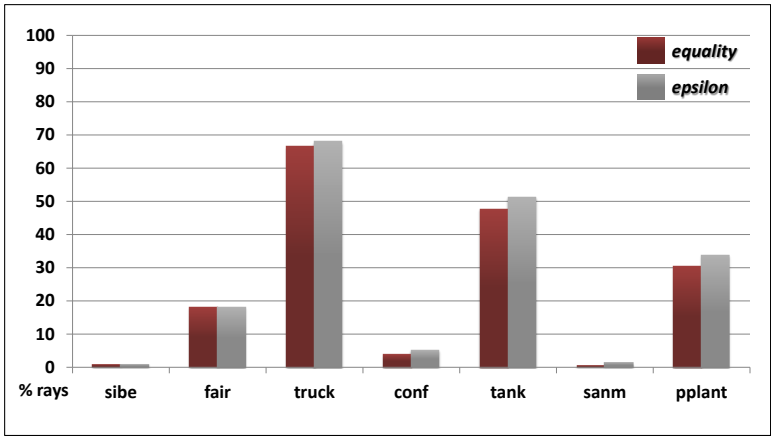
Figure 2. The problem of overlapping coplanar triangles. These images depict rays along which two or more overlapping coplanar triangles are encountered when rendering three models: *truck* (left) and *tank* (center), which are engineering CAD models used in the physics-based simulations motivating our work; and *conference* (right), a scene commonly used in the computer graphics literature. Here, intensity of the red channel is determined by the number of coplanar intersections encountered: brighter red indicates more such intersections. (CAD models courtesy of L. Butler, US Army Research Laboratory.)

However, the physical objects modeled for these simulations tend to have surfaces in perfect contact with each other: surfaces between objects are modeled explicitly and are coplanar. The problem of overlapping coplanar triangles is particularly acute in the engineering CAD models used in simulations motivating this work. For example, the left and center panels of Figure 2 depict rays along which two or more overlapping coplanar triangles are encountered when rendering two such models.

However, as shown in the right panel of Figure 2, the problem is not unique to geometry from engineering CAD: scenes common to the computer graphics literature exhibit such overlap as well. In fact, the data in Figure 3 show that all models used in this work—most of which come from the computer graphics literature—exhibit at least some overlapping coplanar triangles.

Most optical ray tracers do not handle intersections at the shared interface correctly, as illustrated in Figure 4. If a new ray is traced using a negative ε-offset, intersections may be incorrectly repeated. Similarly, if a new ray is traced using a positive ε-offset, intersections may be incorrectly missed. In general, there is no mechanism to reliably and correctly compute multiple intersections for models with overlapping coplanar facets using traditional first-hit traversal. An accurate and efficient solution to multi-hit traversal is thus necessary to resolve overlapping coplanar triangles, both in computer graphics applications and in physics-based simulations.

Moreover, the overhead imposed by the necessary workarounds makes performance of a first-hit solution unacceptable for real-time applications. For example, the impact of re-traversing the acceleration structure—an operation that already dominates many ray tracing applications—simply exacerbates the situation. When combined with the overhead of launching new rays to find additional intersections—as in GPU-based ray tracing engines—any gains provided by using massively parallel architectures are quickly and significantly reduced.



scene	# hits	equality		epsilon	
		# coplanar hits	% total	# coplanar hits	% total
sibe	786432	6237	0.79%	6336	0.81%
fair	576370	104082	18.06%	104316	18.10%
truck	406507	269658	66.34%	275636	67.81%
conf	786432	29715	3.78%	39907	5.07%
tank	317953	151087	47.52%	162558	51.13%
sanm	780664	4376	0.56%	9901	1.27%
pplant	333828	101016	30.26%	112571	33.72%

Figure 3. Overlapping coplanar triangles are a common problem. Though more prevalent in engineering CAD models, overlapping coplanar triangles are not unique to these models. Here, the graph depicts the percentage of rays intersecting geometry that also encounter overlapping triangles for all scenes depicted in this work. In the *equality* case (red), a coplanar hit is counted if the floating-point test for equality (for example, `operator==` in C++) between two or more t-values along the ray is true. In the *epsilon* case (gray), a coplanar hit is counted if the difference between two or more t-values along the ray is less than a scene-dependent ε-value.

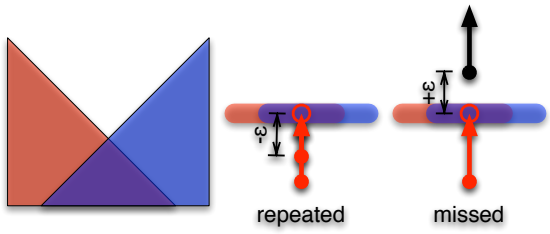


Figure 4. The problem of "continuing" *first-hit* ray traversal. Approximating *multi-hit* traversal with ε-offsets in a *first-hit* engine is insufficient in the case of fully or partially overlapping coplanar triangles (left). Intersections may be erroneously repeated (center) or missed entirely (right), leading to incorrect results.

In contrast, a system that supports multi-hit traversal as a fundamental operation alleviates these issues: it avoids inaccurate, incorrect approximations using ε-offsets; it properly resolves intersections at overlapping coplanar facets; and it generates additional intersections without re-traversing the acceleration structure or explicitly launching new rays.

We introduce an efficient multi-hit ray traversal algorithm that returns, in order, multiple primitives intersected by a ray. We also investigate its performance in a GPU ray tracer, and demonstrate two problems easily solved with our algorithm.

2. Multi-Hit Ray Traversal

A multi-hit ray traversal algorithm is one that returns information concerning one or more, and possibly all, primitives intersected by a ray. Multi-hit traversal generalizes both first-hit and all-hit traversal schemes, though distinguishing among them may be beneficial for performance (see Section 3). While not strictly required, we assume multi-hit traversal reports intersections in ray-order, as most applications utilizing multi-hit results will require such ordering.

Naive multi-hit ray traversal. Algorithm 1 provides pseudocode for a naive multi-hit traversal algorithm. The algorithm maintains a per-ray data structure to record information about each intersection and proceeds in two phases. First, the ray iteratively traverses the acceleration structure, recording information about each intersection in sorted order (lines 4–10). Second, a per-hit user-level callback is invoked to process each intersection point once traversal is complete (lines 11–13). The return value indicates whether or not additional intersections should be processed (lines 12–13). Any user-level processing required to finalize the trace operation can be performed after traversal and intersection processing are complete.

This algorithm is simple and effective: it imposes very few constraints on an actual implementation, it does not assume a particular acceleration structure, and it allows the user to process as many intersections as desired.

However, this algorithm is potentially very slow: it effectively implements the all-hit traversal scheme as opportunities for early-exit occur during intersection processing, only after all intersections have been found. All-hit is a particular specialization of multi-hit traversal and should not be imposed in cases for which users require only a subset of the intersections. Instead, we seek an efficient multi-hit traversal algorithm that permits opportunities for early-exit during ray traversal, so that unnecessary operations are avoided when possible.

Buffered multi-hit ray traversal with early-exit. Algorithm 2 provides pseudocode for just such an algorithm. As before, this version maintains a per-ray data structure to record information about each intersection. However, rather than allow the list to grow without bound, an ordered buffer of a fixed size is used.

```
1:  function TRAVERSE(root, ray)
2:      INITIALIZE(hitList)
3:      node ← root
4:      while VALID(node) do
5:          if !EMPTY(node) then
6:              for triangle in node do
7:                  if INTERSECT(triangle, ray) then
8:                      hitData ← (t, u, v, tID, ...)
9:                      INSERT(hitList, hitData)
10:         node ← NEXT(node)
11:     for hitData in hitList do
12:         if !USERHIT(ray, hitData) then
13:             return
```

Algorithm 1. Naive *multi-hit* ray traversal.

```
1:  function TRAVERSE(root, ray)
2:      INITIALIZE(hitList)
3:      node ← root
4:      while VALID(node) do
5:          if !EMPTY(node) then
6:              INITIALIZE(hitMask)
7:              repeat
8:                  repeatNode ← FALSE
9:                  for triangle in node do
10:                     if !CONTAINS(hitMask, tID) then
11:                         if INTERSECT(triangle, ray) then
12:                             hitData ← (t, u, v, tID, ...)
13:                             if FULL(hitList) then
14:                                 repeatNode ← TRUE
15:                             INSERT(hitList, hitData)
16:                 for hitData in hitList do
17:                     if !USERHIT(ray, hitData) then
18:                         return
19:                     if repeatNode then
20:                         ADD(hitMask, hitData.tID)
21:             until !repeatNode
22:         node ← NEXT(node)
```

Algorithm 2. Buffered *multi-hit* ray traversal.

In the common case—where the number of primitives, and thus possible intersections, in a given node is less than the number of buffer entries—the algorithm collects intersections and dispatches the per-hit callback for each intersection on a per-node basis (lines 7–15 and 16–20).

Multiple passes handle nodes with more primitives than buffer entries (lines 7–21). To account for this possibility, the algorithm also maintains a per-primitive flag indicating whether or not each primitive was intersected in a previous pass (lines 6 and 19–20).

In each pass, all unflagged primitives are tested for intersection, and that information is added to the buffer in sorted order as necessary (lines 11–15). If the buffer is full, each nearer intersection pushes the last entry off the end of the buffer and a flag is set to indicate another pass is required (lines 13–15). After all primitives have been tested, the per-hit callback is invoked on the buffered entries (lines 17–18), and the algorithm either prepares for the next pass (lines 19–21) or continues to the next node (line 22). Finally, once all nodes have been traversed or all required intersections have been found, any user-level processing required to finalize the trace operation can then be performed, as in the naive algorithm.

This algorithm provides an opportunity for early-exit: as before, the per-hit callback indicates whether or not additional intersections are desired. For cases in which they are not, ray traversal—not just intersection processing—ends (line 18). Otherwise, the primitive is marked as complete (lines 19–20), and processing continues with the next entry.

To enable early-exit while guaranteeing correctness, we assume an acceleration structure based on space-partitioning: nodes do not overlap and can therefore be traversed in strict front-to-back order.[1] Correctly intersecting a primitive that spans multiple nodes remains an issue, however. As noted by Ize and Hansen [2011], an approach similar to mailboxing can be used to mitigate this issue, and primitive splitting offers another straightforward solution. We avoid multiple intersection as a consequence of our implementation: rays are effectively clipped to the bounds of each leaf node during traversal, creating a line segment. Such a ray cannot intersect a primitive across multiple nodes, except on boundary conditions that acceleration structure construction prevents; so, if a ray intersects a primitive, it will do so only within the bounds of one node.

Thus, in a space-partitioning structure, the buffered algorithm potentially improves performance relative to the naive algorithm:

- by simplifying the data structure used to record intersection information, as it need not grow without bound; and,

[1] Acceleration structures based on object-partitioning will work with a modified version of our buffered multi-hit algorithm; however, these structures impose additional complexity to exploit the early-exit case, as all overlapping nodes must be resolved to ensure required intersections are computed and ordered correctly before allowing early-exit.

- by reducing in-memory storage requirements, as there is no need to retain all intersections along a ray.

Ultimately, the buffered algorithm leads to better overall performance—even in the all-hit case, as shown in Section 3.

3. Results

To understand the impact of multi-hit ray traversal as a fundamental operation, we investigate performance in a GPU ray tracer and demonstrate two problems easily solved with multi-hit traversal.

3.1. Performance

To evaluate multi-hit performance, we render the five test scenes shown in Figure 5 under several different scenarios using an NVIDIA GTX 690 GPU. For all scenes except *tank*, we use the geometry and viewpoints provided by Aila et al. [2009; 2012].

Implementation details. Traversal algorithms are implemented in the open source GPU ray-tracing engine, *Rayforce*.[2] The engine uses a graph-based spatial indexing structure to accelerate ray/primitive intersection operations [Gribble and Naveros 2013] and achieves first-hit performance in the range of 200–800 million rays per second (Mrps) for the test scenes on the test hardware. Performance is thus commensurate with other state-of-the-art, GPU-optimized first-hit ray-tracing systems (for example, Aila et al. [2009; 2012] and OptiX [Parker et al. 2010]).

The INSERT function used by both the naive (Algorithm 1, line 9) and the buffered (Algorithm 2, line 15) multi-hit implementations uses insertion sort to order intersection points along each ray. Though more sophisticated sorting algorithms could be used, maintaining points in sorted order has very little impact on overall traversal performance—we opted for simplicity in this case.

Figure 5. Scenes used for performance evaluation. Five scenes of varying geometric and depth complexity are used to evaluate the performance of multi-hit ray traversal. Observe that in the *tank* scene, the first-hit visible surfaces hide significant internal complexity. This model is representative of real datasets used in ballistic penetration simulations and is, therefore, particularly useful as a test of multi-hit traversal.

[2] *Rayforce*: Exceptional performance through non-traditional means. Source code is available via http://rayforce.survice.com.

For naive multi-hit traversal, the USERHIT function (Algorithm 1, line 12) simply logs intersection information required for shading. To finalize the trace operation, alpha-blending accumulates per-ray samples and terminates intersection processing when appropriate; pixel values in the framebuffer are then set accordingly.

In the case of our buffered implementation, USERHIT (Algorithm 2, line 17) accumulates per-ray samples for alpha-blending and indicates whether or not ray traversal should continue. After traversal, the trace operation is finalized simply by setting pixel values to the incrementally computed results. Finally, in our buffered implementation, three entries are used: this size was empirically determined to provide the highest performance on the test hardware.

Experimental setup. We render a series of 1000 frames at 1024×768 pixels on an NVIDIA GeForce GTX 690 and measure the resulting performance. We report results in terms of millions of intersections per second (Mips). Although millions of rays per second (Mrps) is a more common metric for ray-traversal performance, in this context it is less meaningful in an absolute sense: with multi-hit traversal, the number of intersections per ray is generally greater than one. While scaling Mrps by the average number of intersections per ray or performing similar manipulations may give a reasonable indicator of performance, we choose Mips as a direct indicator of multi-hit performance, even if raw traversal performance is less obvious when compared to traditional first-hit engines. We note, however, that Mips is equivalent to Mrps in cases for which only the nearest intersection is required. Thus, multi-hit results relative to first-hit performance, as in the find-first-intersection tests discussed below, can be compared directly.

As noted above, shaders use only a simple alpha-blending operation; nevertheless, all per-frame overhead—GPU kernel launch, ray generation, host/device synchronization, and so forth—is included. For each test, a separate counting pass is used to accurately determine the total number of intersections computed during rendering.

Find-first-intersection. We first measure the impact of maintaining the internal state necessitated by our buffered multi-hit algorithm when computing just the nearest intersection. Figure 6 compares a standard first-hit traversal algorithm to our multi-hit scheme in this case. Multi-hit clearly imposes some overhead—about 30% in these tests—but it generally performs quite well, even in this specialized find-first-intersection case.

Find-all-intersections. We next measure the impact of our buffered multi-hit traversal algorithm when computing all intersections. Figure 7 compares the naive and buffered algorithms for this find-all-intersections case. As can be seen, our buffered algorithm outperforms the naive algorithm by about 10% in these tests—the speedup, though modest, is measurable. This result is very encouraging: we expect to do at least this well in the find-some-intersections case.

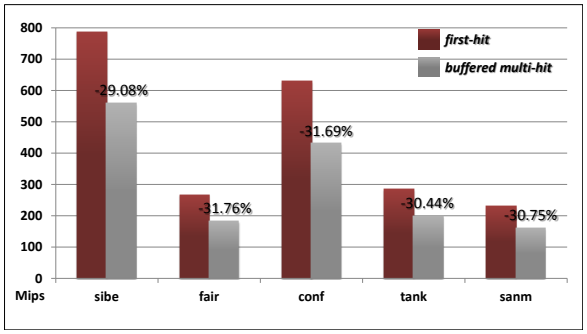


Figure 6. *Multi-hit* traversal overhead. Here, the graph compares performance (in Mips) between first-hit traversal and buffered multi-hit traversal when the user requests only the nearest intersection point. Though multi-hit traversal imposes some overhead to maintain internal state, performance degrades by only about 30% for our test scenes.

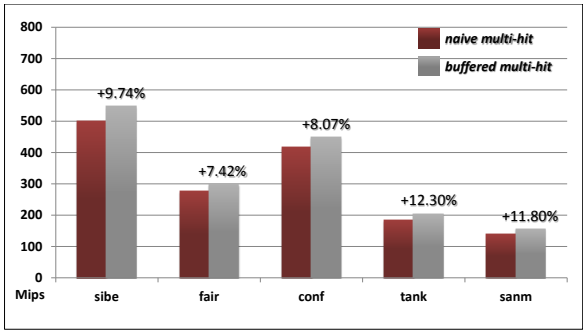


Figure 7. Naive v. buffered: find-all-intersections. Here, the graph compares performance (in Mips) of naive multi-hit and buffered multi-hit traversal when the user requests all intersections. Our buffered algorithm outperforms the naive algorithm by about 10% in these tests.

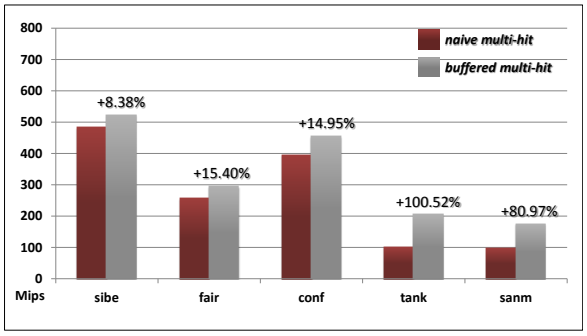


Figure 8. Naive v. buffered: find-some-intersections. Here, the graph compares performance (in Mips) of naive multi-hit and buffered multi-hit traversal when the user requests only the first five intersections along a ray. Our buffered algorithm outperforms the naive algorithm by about 44% for the scenes tested.

Find-some-intersections. Finally, we measure the impact of our buffered multi-hit traversal algorithm when computing only the first five intersections. Figure 8 shows that buffered multi-hit outperforms the naive algorithm by about 44%, with a more dramatic impact for models of high geometric complexity.

3.2. Examples

We demonstrate the utility of multi-hit traversal by applying our buffered algorithm to two important problems: optical transparency in computer graphics and ballistic penetration simulation in non-optical rendering.

Optical transparency. Fast and accurate transparency is an open problem in computer graphics. To deal with issues that are difficult to solve in object-space, current raster-based solutions utilize fragment sorting on the GPU. Unfortunately, this operation can be more expensive than geometry sorting, and it fails in cases of coplanar objects. However, efficient multi-hit traversal in a ray-based rendering system provides an attractive alternative to raster-based transparency. Figure 9 shows the results of using our buffered multi-hit implementation to render a model with coplanar geometry using alpha-blending. This image can be generated at interactive rates using our implementation on an NVIDIA GTX 690.

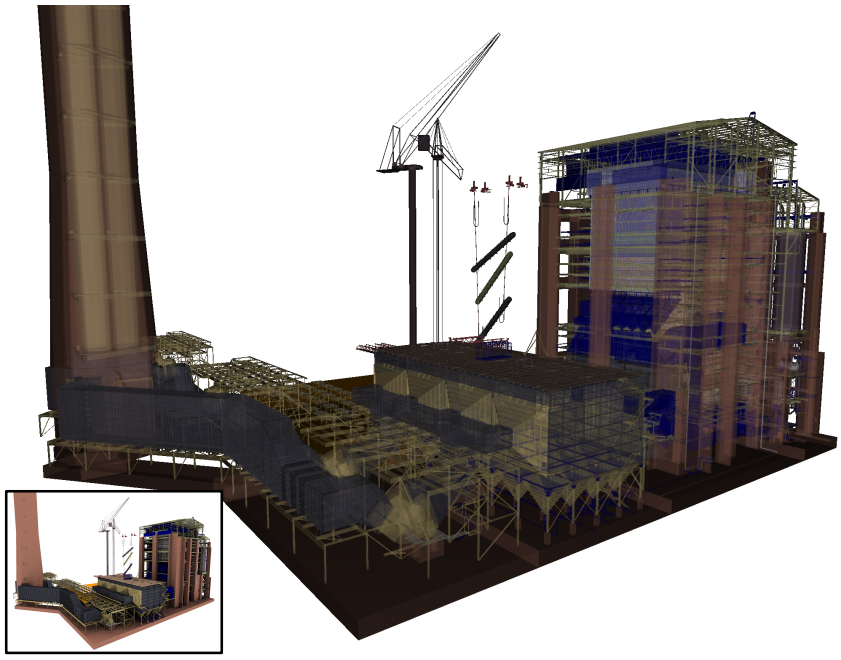


Figure 9. Optical transparency. Direct support for multi-hit traversal permits fast and accurate transparent rendering, even in cases of coplanar objects. Here, multi-hit traversal is used to render the *power plant* model, in which about one-third of rays intersecting geometry also encounter overlapping coplanar facets. (Model courtesy of M. McGuire, Williams College and NVIDIA Research.)

Ballistic penetration simulation. Vulnerability/lethality (V/L) analysis employs ballistic penetration simulation to evaluate threat-target interactions and plan live-fire testing events. In these non-optical rendering scenarios, ray tracing is used to determine the path and interaction of projectiles and debris with other objects. In a manner similar to optical transparency, ballistic simulation computes the energy absorbed as projectiles (rather than photons) pass through objects. Ballistic penetration follows a type of exponential decay similar to the Beer-Lambert Law; however, the equations are derived empirically and are typically functions of both material properties and threat parameters [Butler and Stephens 2007].

Figure 10 shows one simulation generated using our buffered multi-hit implementation. Here, pixel color is computed by an absorbance function based on object material and distance traveled. Performance is sufficient to execute the simulation and produce the corresponding visualization at interactive rates using our implementation on an NVIDIA GTX 690.

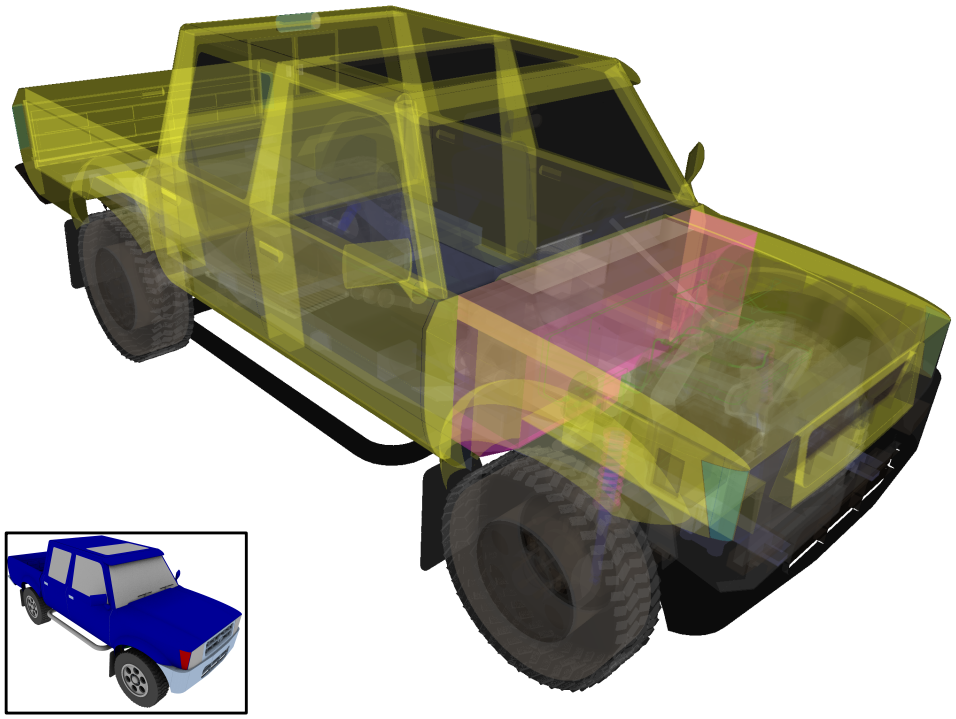


Figure 10. Ballistic penetration simulation. Direct support for multi-hit traversal permits fast and accurate ballistic penetration simulation for V/L analysis. Here, multi-hit traversal is used to simulate the impact of a threat against a vehicle target.

4. Discussion

We have introduced an efficient algorithm for ordered multi-hit ray traversal. As multi-hit traversal generalizes the first-hit scheme, one could imagine implementing

only multi-hit in a ray-tracing engine; however, as shown in Section 3, a separate first-hit traversal mechanism provides a clear performance advantage for instances in which only the nearest intersection is required. Similarly, all-hit specializes multi-hit ray traversal, but evaluation shows the performance impact of our buffered algorithm outweighs the additional logic it requires, even in the find-all-intersections case. Therefore, supporting a separate, naive version of the algorithm specific to all-hit traversal is (probably) not justified.

We have also demonstrated two potential applications of multi-hit ray traversal: it offers an interesting alternative for optical transparency in computer graphics, and it could radically transform legacy V/L applications by providing real-time simulation and visualization capabilities. Other possible applications of multi-hit traversal include:

- *Alpha textures.* The use of alpha textures implies that, even in first-hit scenarios, more than one intersection point may be required. In instances when traversal kernels do not have access to texture data, multi-hit may be of value: ray traversal can easily return several intersection points—for example, the first two or three such points encountered in the nearest node of an acceleration structure—for use during shading. We observe that even a first-hit traversal algorithm must test all primitives in a node to determine the closest intersection; therefore, leveraging multi-hit traversal to make several intersection points available during shading may reduce the overhead imposed by locating the nearest intersection on an opaque object in the presence of alpha textures.

- *Thin fibers.* A similar approach may also improve performance when rendering thin fibers—for example, hair or fur. Alpha-blending is commonly used in hair rendering, as light-colored hair is semi-transparent and hair strands are generally much thinner than pixels are wide. Some recent methods utilize advanced features of GPUs to determine (possibly approximate) primitive order when rendering semi-transparent phenomena [Sintorn and Assarsson 2008; Sintorn and Assarsson 2009; Yu et al. 2012], while others rely on random, sub-pixel stipple patterns to correctly alpha-blend geometry on average [Enderton et al. 2010]. In contrast, multi-hit traversal computes any number of intersection points along a ray in sorted order and naturally permits correct alpha-blending of many fibers. As such, multi-hit may improve performance relative to other methods for optical transparency when rendering semi-transparent surfaces and can be easily incorporated into rendering pipelines already utilizing ray tracing for other visual effects.

- *Constructive solid geometry.* Constructive solid geometry (CSG) methods model complex objects by performing set operations over collections of simpler ob-

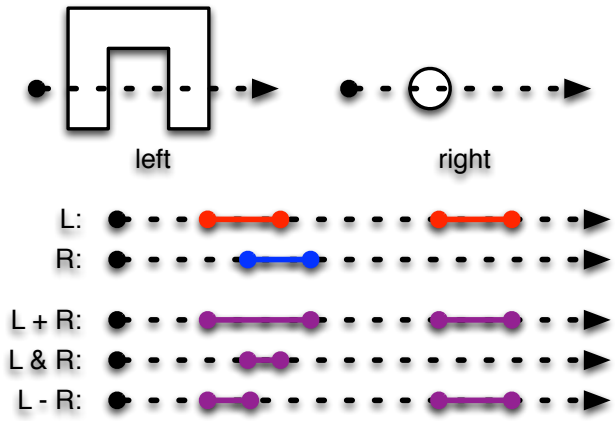


Figure 11. Multi-hit ray traversal for CSG modeling. Multi-hit traversal provides a straightforward mechanism with which to implement Roth-style CSG, as evaluating set operations such as union (+), intersection (&), and difference (−), reduces to classification of intersection points along a ray with respect to solid objects. (Figure after Roth [1982].)

jects. Roth [1982] introduces ray casting as a means to implement CSG modeling. As illustrated in Figure 11, evaluating set operations reduces to classification of intersection points along a ray with respect to solid objects. Multi-hit ray traversal is a natural fit in this context, as multiple intersection points along a ray are required to evaluate these operations correctly.

We are excited to see how these and other problems in rendering can be solved by applying multi-hit ray traversal.

5. Future Work

We have focused exclusively on multi-hit traversal in which rays do not change direction at points of intersection. Such an algorithm would be particularly useful for handling dielectric objects in a Whitted-style ray tracer, for example. We plan to implement such an algorithm in the GPU ray-tracing engine used for the performance evaluation in this work.

Similarly, we have implemented our buffered multi-hit traversal using an acceleration structure based on space-partitioning. We also plan to implement the modifications necessary to accommodate structures based on object-partitioning in the future.

Acknowledgments

Discussions with several people helped to shape the ideas presented here, including Lee Butler (US Army Research Laboratory); Jefferson Amstutz, Mark Butkiewicz, and Scott Shaw (SURVICE Engineering); and, Carsten Benthin, Ingo Wald, and Sven Woop (Intel Labs).

Alexis Naveros is chief architect of the *Rayforce* engine, which has been funded in part by research grants from the US Office of Naval Research and the US Army Research Laboratory.

Ethan Kerzner is funded in part by the US Army Research Laboratory "Cooperative Agreement: Applying GPU Computing and Computer Graphics to Engineering Analysis and Military Applications."

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High Performance Graphics, 2009*, ACM, New York, NY, USA, 145–149. http://doi.acm.org/10.1145/1572769.1572792. 8

AILA, T., LAINE, S., AND KARRAS, T. 2012. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. Tech. Rep. NVR-2012-02, NVIDIA Research. https://research.nvidia.com/publication/understanding-efficiency-ray-traversal-gpus-kepler-and-fermi-addendum. 8

BUTLER, L. A., AND STEPHENS, A. 2007. Bullet ray vision. In *2007 IEEE Symposium on Interactive Ray Tracing*, IEEE, Los Alamitos, CA, 167–170. 12

ENDERTON, E., SINTORN, E., SHIRLEY, P., AND LUEBKE, D. 2010. Stochastic transparency. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, I3D '10, 157–164. http://doi.acm.org/10.1145/1730804.1730830. 13

GRIBBLE, C., AND NAVEROS, A., 2013. Ray tracing with a graph. Unpublished manuscript. 8

IZE, T., AND HANSEN, C. 2011. RTSAH traversal order for occlusion rays. *Computer Graphics Forum 30*, 2, 297–305. http://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2011.01861.x/abstract. 2, 7

MAULE, M., COMBA, J. L., TORCHELSEN, R. P., AND BASTOS, R. 2011. A survey of raster-based transparency techniques. *Computers & Graphics 35*, 1023–1034. 2

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, K., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGRUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2010) 29*, 4, 66:1–66:13. http://doi.acm.org/10.1145/1778765.1778803. 8

ROTH, S. D. 1982. Ray casting for modeling solids. *Computer Graphics and Image Processing 18*, 2 (February), 109–144. 14

SINTORN, E., AND ASSARSSON, U. 2008. Real-time approximate sorting for self shadowing and transparency in hair rendering. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, I3D '08, 157–162. http://doi.acm.org/10.1145/1342250.1342275. 13

SINTORN, E., AND ASSARSSON, U. 2009. Hair self shadowing and transparency depth ordering using occupancy maps. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '09, 67–74. http://doi.acm.org/10.1145/1507149.1507160. 13

STEPHENS, A., BOULOS, S., BIGLER, J., WALD, I., AND PARKER, S. G. 2006. An application of scalable massive model interaction using shared memory systems. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, Eurographics Association, Aire-la-Ville, Switzerland, EG PGV'06, 19–27. http://dx.doi.org/10.2312/EGPGV/EGPGV06/019-026. 2

VASILAKIS, A. A., AND FUDOS, I. 2012. Depth-fighting aware methods for multifragment rendering. *IEEE Transactions on Visualization and Computer Graphics 19*, 6, 967–977. http://dx.doi.org/10.1109/TVCG.2012.300. 2

YU, X., YANG, J. C., HENSLEY, J., HARADA, T., AND YU, J. 2012. A framework for rendering complex scattering effects on hair. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, I3D '12, 111–118. http://doi.acm.org/10.1145/2159616.2159635. 13

Author Contact Information

Christiaan Gribble
Applied Technology Operation
SURVICE Engineering Company
6014 Penn Avenue
Pittsburgh, PA 15206
christiaan.gribble@survice.com
http://www.rtvtk.org/~cgribble/

Alexis Naveros
Applied Technology Operation
SURVICE Engineering Company
1362 Brass Mill Road, Suite 5
Belcamp, MD 21017
alexis.naveros@suvice.com

Ethan Kerzner
SCI Institute
University of Utah
72 Central Campus Drive
Salt Lake City, UT 84112
kerzner@sci.utah.edu

Christiaan Gribble, Alexis Naveros, Ethan Kerzner, Multi-Hit Ray Traversal, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 1, 1–17, 2014
http://jcgt.org/published/0001/02/01/

Received:	2013-08-26		
Recommended:	2013-11-20	Corresponding Editor:	Matt Pharr
Published:	2014-02-07	Editor-in-Chief:	Morgan McGuire