LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-208375

Ray tracing through a hexahedral mesh in HADES

G. L. Henderson, M. B. Aufderheide

December 3, 2004

NECDC 2004
Livermore, CA, United States
October 4, 2004 through October 7, 2004

Proceedings from the NECDC 2004

Ray tracing through a hexahedral mesh in HADES (U)

G.L. Henderson, * M.B. Aufderheide*

*Lawrence Livermore National Laboratory, Livermore, California, 94550

In this paper we describe a new ray tracing method targeted for inclusion in HADES. The algorithm tracks rays through three-dimensional tetrakis hexahedral mesh objects, like those used by the ARES code to model inertial confinement experiments. (U)

Problem Statement

It is often useful to simulate radiographic images in industrial procedures that utilize both radiography and computer modeling. Comparing a simulated image with a radiographic image produced during an experiment provides a means to verify correctness of the physics models and input data. Careful comparison between real and simulated images can aid in the interpretation of experimental results.

HADES is a software program that simulates radiography using ray tracing techniques. The program was originally developed to simulate X-Ray transmission radiography, for nondestructive evaluation applications. Over time however, HADES has grown to simulate neutron radiography over a wide range of neutron energies, proton radiography in the 1 MeV to 100 GeV range, and recently phase contrast radiography using X-Rays in the keV energy range. HADES can simulate parallel-ray or cone-beam radiography through a variety of mesh types, as well as through collections of geometric objects. HADES can be run on a variety of computer architectures including: SGI, Sun and HP/Compaq workstations, Cray and IBM computers, and Macintosh personal computers.

ARES is a multiprocessing program that models physical phenomena in three dimensions. The code can be run on a variety of computer architectures. An ARES mesh composition can include millions of hexahedral cells.

In order to render a detailed radiographic image, a HADES simulation depicts a detector as a square array of pixels. For a 1024 x 1024 image, more than one million transmission rays must be tracked through a multi-million cell ARES mesh; a daunting computation.

Currently, HADES uses an algorithm that is fast and simple to ray trace this type of mesh. For every voxel in the mesh, HADES computes a “mesh shadow” onto the detector

pixels. The path length due to each voxel is added at the detector plane. This approach is very fast because it does not require tracking through the mesh. The disadvantage of this approach is that the source-detector rays are not actually tracked through the mesh. For future applications of HADES, the radiography algorithm will need complete tracks returned by the ray tracking procedure. The problem of tracking through a tetrakis hexahedral mesh has not been solved in general. Currently, Monte Carlo codes which need to track through such meshes rezone into a Cartesian mesh.

Algorithm Overview

The new algorithm computes ray path lengths through cells that comprise a three dimensional mesh object. This method considers that each ray may intersect with every cell, but imposes a set of hierarchal filters to quickly discard consideration of ray/surface pairs that have no spatial extent overlap.

The intersection function requires pointers to structures that describe hexahedral cell coordinates and the physical properties of all cells for one domain[1]. A single function invocation can track multiple rays through one domain. The function returns a set of linked lists, one list per input ray. Each list contains all the intersection segments between a particular ray and an entire mesh domain of the 3D hexahedral mesh object.

The hierarchical ray filters are applied at three "mesh object" levels prior to computing intersections. Each filter is the same spatial extent test, comparing the Cartesian extent of ray end points with the extents of the intersection object (an entire domain, a single cell, or one cell plane facet). The filter system works as follows: spatial overlap is measured along the "independent" component of a ray (i.e. the [x, y, or z] component that spans the greatest extent). If the ray and object overlap in this component, the two "dependent" ray components are "trimmed" to the overlap extent. Ray intersections are possible only if the ray extent along each trimmed, dependent component overlaps the respective object extents.

The intersection function populates a linked list for each ray by looping over the cells of a domain. Every cell is decomposed into 24 triangular facets (six faces per cell, 4 facets per face). Each triangular facet is checked for intersection with the ray. After all facets of a cell are processed, the ray/cell intersection set is ordered by distance along the ray, enabling intersection path length(s) to be computed.

The intersection algorithm itself is not the subject of this paper. Intersection methods are widely published. In fact, much of our low-level intersection procedure is implemented directly from "Computational Geometry in C" (*O'Rourke, 1998*).

---

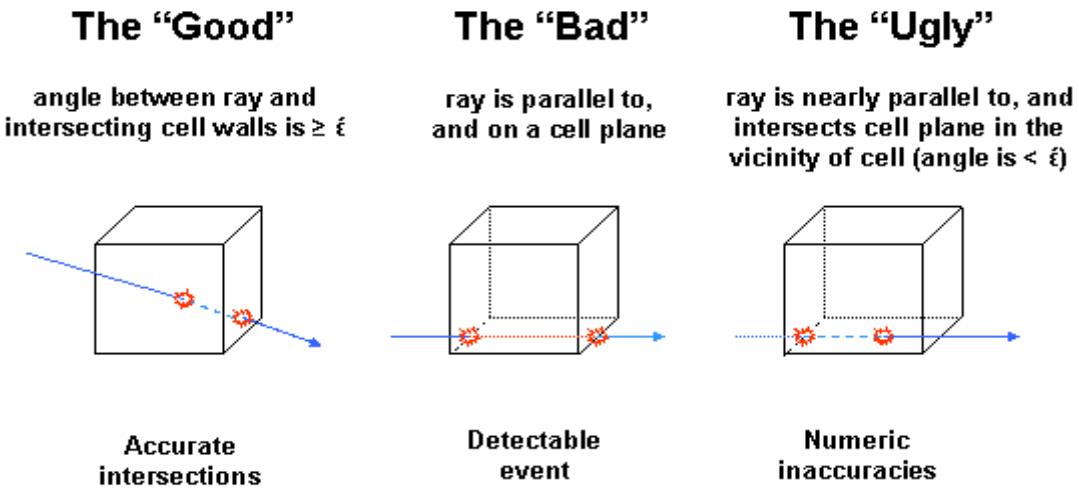
[1] In this context, a "domain" represents a collection of zones, defined by the simulation code, for the purpose of forming parallel (independent) processes, in order to partition work to multiple computer processors.

Instead, we will describe numeric difficulties that arose during the software testing phase, and the solutions we implemented to counter those problems.

Numeric Difficulties

Deficiencies in the intersection method began to appear during a test that involved tracking hundreds of rays. In a small test that tracks 650 rays, four rays had object path lengths computed incorrectly. The incorrect path lengths were as much as five percent short. Investigation revealed that, for every erroneous ray path length, the ray traversed nearly (but not exactly) parallel through a face of one or more cells. Further inspection indicated that, in these situations, an intersection point was computed, but the ray and plane were so nearly parallel that computations produced inaccurate results. In this scenario, the point of intersection typically appears to fall outside the triangular facet in question, and a valid cell intersection was not tallied.

The figure below illustrates three classes of ray/cell intersections. The *Good* class of intersections is computed correctly. Instances of the *Bad* class are detected and also computed correctly by the algorithm. (Duplicate ray segments that are accumulated along contiguous cell interfaces that share the computed intersection segment are removed later). In our test, all four incorrectly computed ray path lengths derived from instances of the *Ugly* ray/cell intersection class.



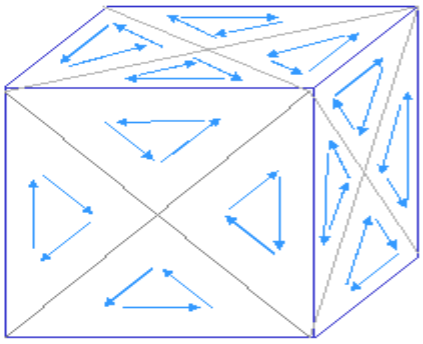
Solutions to Numeric Inaccuracies

(1) Tolerances have been introduced at several critical computations. This helps somewhat, but only in determining whether a ray/plane intersection point falls within or outside of a triangular cell face facet. Even here, adding a numeric tolerance only provides a knob capable of dialing up or down additional intersection segments along a temperamental ray/cell track. A larger tolerance epsilon increases the number of intersection segments tabulated.

(2) A second tactical change produces consistently better results. Note that each "leg" of a cell facet is, in fact shared by two facets within the cell. For every adjacent cell that touches that leg, two more facets share that same leg. It is possible to assure identical (and consistent) results for the common computations that are repeated along those contiguous facets. We accomplish this by considering a specific leg to be a directed vector extending from *point A* to *point B, **for all the facets that share that AB segment***. This can be done by imposing a vertex ordering regimen within the intersection algorithm. These directed line segments are used to generate planar coefficients, as well as signed area cross products, computed to answer the ray-through-facet question. Within a cell, consistent vector directions are attained by numbering vertices of adjacent facets in opposite directions, in a clockwise or counter-clockwise (CW or CCW) manner. Notice that adjacent facets which share a "corner" leg also must order vertices in opposite directions. From an inter-cellular point of view, consistency is achieved by considering adjacent cells to be "even" and "odd", in all three logical mesh dimensions (visualize a three dimensional checkerboard pattern where even cells are white, and odd cells are black). The vertex order for each specific facet of an odd cell is opposite that of the vertex order direction (CW versus CCW) on even cells.

Within a cell, consistent vector directions are attained by numbering vertices of adjacent facets in opposite directions (CW and CCW). Notice that adjacent facets that share a "corner" leg also order vertices in opposite directions.

From an inter-cell view, consistency is achieved by considering adjacent cells to be "even" or "odd", in all three logical mesh dimensions (visualize a 3-D, black and white checkerboard pattern). For a specific cell facet, the vertex order of an odd cell is opposite that of the order direction on even cells.



(3) Finally, the cells that tally exactly one intersection with a ray are now considered a pathologic case, and evoke special testing. We originally considered this event to be a tangential touch between the ray and the cell. However, diagnostics revealed that this condition accounted for many of the object path length intersection shortages, so that assumption has been changed. We now consider this to be an indication that a ray and surface may be nearly parallel. As such, the circumstance triggers a ray "bumping" algorithm which slightly shifts both ray endpoints (up to 8 different directions), until at least two intersections are tallied, or all eight bumping directions have been exhausted without achieving two or more intersections.

As one might infer from the improvements above, those enhancements may mitigate potential numeric inaccuracies, but by no means do they assure correct ray intersection path lengths. In order to compensate for the limitations of our intersection method (and of floating-point hardware), we have introduced a ray "combing" process, as a final step to the ray tracking procedure. The combing process entails very little overhead because it makes use of one of an existing tracking product requirement (return the ray/mesh intersection segments ordered by distance along the ray). Thus far, this practice has produced very accurate path length results for tests.

(4) The ray combing step is initiated after all intersection segments for a ray have been calculated and ordered by unit position along the ray (0. to 1). The ray combing process "walks and preens" an ordered ray segment list in its entirety. The following combing techniques are applied to each ordered list of ray intersection segments:

- Detect duplicate ray coverage segment sequences. Keep the segment combination that produces the "best fit", and longest ray segment span. Delete the duplicate coverage segments.
- Find contiguous segments with overlapping ray coverage, but where each segment offers a unique coverage length along the ray. Crop both segments so as to eliminate the overlap.
- Find gaps in the ray path coverage. For logical mesh situations where simulation voids are prohibited, extend adjacent segment lengths to fill the ray coverage gap.

Conclusions

- A new ray tracing method has been developed for HADES. The procedure tracks rays through a three dimensional, hexahedral mesh.
- Numeric inaccuracies became apparent during early tests of the algorithm.
- Using a variety of tactics, we appear to have been conquered these errors, at least for a limited set of small tests.

- The addition of a function to comb through the ordered mesh path segments along each ray provides assurance against erroneous intersection events that can arise due to numeric inaccuracies.

Recent Developments and Future Plans

The new ray tracing algorithm (as are many others) is "embarrassingly parallel". Only the ray combing function depends on data that is compute-order dependent.

We have recently developed a parallel version of the algorithm, implemented as a stand-alone "driver" program. A single source code contains both serial and parallel logic. The parallel flow control is written using a master/slave paradigm. It invokes functions from an MPI library.

There are various trade offs to consider when designing a multiprocessing model. The master/slave model has a glaring disadvantage. If executed on a small number of processors, the master process (which, in this case, is a significant fraction of compute resources) is idle much of the time. This is because the master process simply orchestrates the work of slave processors. During long, compute-intensive jobs, idle processors can significantly increase wall clock time to completion. On the other hand, a key advantage of the master/slave paradigm is that slave processor idle time is minimized, since the master immediately reassigns new work to an idle slave processor. This model can be efficient for runs that enlist many processors.

We decided to implement a master/slave model because important customers are using massively parallel computers. Furthermore, our ray tracing application is organized to minimize memory cache swaps; decomposing the work into equal sized compute tasks is difficult. It is impossible to predict what the "typical" number of processors to be allotted to next years "typical" HADES 3D, parallel run will be. If "normal usage" evolves into 8-processor HADES runs, then mediocre compute time gains might follow. This scenario could warrant integration of a second, simpler, round robin parallel model. In this alternative model, all processors would compute ray tracks. Each processor would track all rays through every *n*'th mesh domain, where *n* is the number of processors allocated to the job. Both models could coexist in HADES, with the round robin method invoked internally for parallel jobs enlisted with 8 processors or less.

We have run a few tests to validate the master/slave parallel implementation. The tests were run on IBM and HP/Compaq multiprocessor platforms. The biggest test intersects 65000 rays with 62000 cells. The 62000 cells comprise one quarter of a hemisphere. Because the geometry is symmetric about the X and Y axial planes, each cell is internally reflected about the X- and Y-axis (a common HADES option), resulting in a simulated 248000 cell hemisphere. This test is far smaller than simulations that will be

run by customers. Timings for that test are summarized in Table 1 (for an HP/Compaq computer), and in Table 2 (for an IBM computer). All runs produced accurate path length results. The observed load balance is excellent on the IBM computer, and somewhat less so on the HP/Compaq computer run. On the IBM, notice that, as more processors are applied to the run, time of completion continues to drop.

Table 1: HP/Compaq ES45 EV68 @ 1 GHz.[a]

Number of CPUs	CPU time [longest] (sec)	CPU time [shortest] (sec)[b]	Speedup Factor[c]
Serial	790.58	--------	
4	389.10	(333.48)	2.03x

[a] Each node contained 4 processors and 3.2 GB memory. MPI software provided shared memory MPI only.
[b] Compare longest and shortest processor compute time for an indication of processor load disparity.
[c] Speedup Factor times longest CPU time = Serial run time.

Table 2: IBM Power4 p655 @ 1.5 GHz.[a]

Number of CPUs	CPU time [longest] (sec)	CPU time [shortest] (sec)[b]	Speedup Factor[c]
Serial	610.82	--------	
4	269.80	(269.35)	2.26x
8	127.17	(126.16)	4.80x
16	67.50	(67.10)	9.05x

[a] Each node contained 8 processors and 16GB memory. Federated switch for inter-node MPI communication.
[b] Compare longest and shortest processor compute time for an indication of processor load disparity.
[c] Speedup Factor times longest CPU time = Serial run time.

In the near future, we plan to:

- Implement both the serial and parallel versions of the ray tracing algorithm into the next HADES floor version for further testing.
- If the algorithm survives the rigors of customer testing, the algorithm will be adapted into the HADES C++ code rewrite.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contractno W-7405-Eng-48.

References

O'Rourke, J., *Computational Geometry in C, Second Edition*, (Cambridge University Press, Cambridge, UK, 1998), chapters 1 and 7.