EBMesh: An Embedded Boundary Meshing Tool

Hong-Jun Kim[1] and Timothy J. Tautges[2]

[1]Argonne National Laboratory, Argonne, IL, U.S.A hongjun@mcs.anl.gov
[2]Argonne National Laboratory, Argonne, IL, U.S.A tautges@mcs.anl.gov

Summary. This paper describes a method for constructing Cartesian meshes for embedded boundary algorithms by using a ray-tracing technique. In this approach, each mesh cell is distinguished as being inside, outside, or on the boundary of the input geometry, which is determined by firing rays parallel to x/y/z coordinates. The most expensive process of the embedded boundary mesh generation, an edge-geometry intersection test, is performed for the group of edges on a fired ray line together, which decreases the computational complexity of the whole method significantly. Produced boundary cells also have edge-cut fraction information and volume cut fraction information for each material. This work is implemented to be enable to directly import various CAD-based solid model formats and as an open-source code to be used easily in many engineering simulation fields.

Key words: Embedded boundary mesh, cut-cell, ray-tracing

1 Introduction

The Finite Element (FE), Finite Difference (FD), and Finite Volume (FV) methods solve Partial Differential Equations (PDEs) by using different types of discretization for the spatial domain. FE and FV methods are popular because they can resolve complicated geometric domains through the use of body-fitted grids. FD methods are straightforward to implement, and can achieve higher-order approximations of derivatives on structured grids, at the cost of those grids being difficult to generate for complex domains. Another class of simulations, based on the Embedded Boundary method, splits the difference between these by using structured grids on domain interior, and unstructured polyhedral cells or volume fraction approximation for cells intersecting the domain boundary [1] [2] [3] [4]. For this method, the mesh generation challenge is in generating such “Embedded Boundary” grids.

Previous methods for generating EB grids rely on computing the intersection of each cell from a structured Cartesian grid with geometric model boundaries. Various approaches are used, depending on the representation of the geometric model boundaries. In the method by Aftosmis et al. [1], surface triangles of component-based geometries are preprocessed to remove interior triangle pairs and break intersected triangles to smaller ones. Remaining boundary triangles are stored in a special tree structure to reduce the time to compute cell-boundary intersections. Cartesian cells found to intersect this boundary are converted to polygons using intersection points along cell edges.

With no optimization, the cost for whole triangle-cell intersection test, which is usually most expensive task, will be approximately AN_tN^3, where N_t is the number of triangles on the boundary, N is the number of divisions on each side of the entire Cartesian mesh block. Aftosmis et al. uses optimization to reduce this time; Alternating Digital Tree (ADT) [5] gives a list of intersection candidate triangles for each cell in $\log N_t$ time and the intersection test is performed for the candidate triangles. However, since the tree search is performed for all N^3 cells, the approximate scaling behavior of intersection test for the method will be $AN^3\log N_t$.

Colella et al. [2] generate EB meshes by using implicit functions to represent geometric boundaries of a solid. Interior and exterior regions are indicated by negative and positive values of this implicit function φ, respectively; the function value is zero on the boundary. Colella et al. incorporate the definition of this implicit surface into PDE terms that evaluate fluxes over cell surfaces (or over the boundary, for cells intersecting the boundary). Although not explicitly stated, they use recursive subdivision of the Cartesian space to determine which cell is boundary out of N^3 cells and the implicit surface evaluations are performed to them. Therefore, it appears that the scaling for the whole intersection test is $AN^3\log N$ where each search is performed in $\log N$ time.

In summary, both approaches reviewed here are bounded below by AN^3, with the term either $\log N$ or $\log N_t$. Aftosmis et al.'s implementation is available under license from NASA, but it is not free for non-government use. Colella et al.'s method is available as part of the Chombo package [6]; however, the user is responsible for providing an implicit surface definition based on their definition of the surface boundary. In addition, it appears that neither method is able to handle domains with multiple materials, though this feature may be straightforward to implement with either method.

In this paper, we propose a new EB mesh generation method, with contributions in several aspects. First, our algorithm achieves $AN^2\log N_t$ scaling performance of cell-triangle intersection test; this reduces the constant factor from previous results. Second, our method handles multi-volume assembly models without modification, without repeating cell-surface intersections for surfaces shared by multiple volumes. Third, our method supports EB mesh generation not only for faceted or triangle-based surfaces, but also directly from CAD geometries. We demonstrate the robustness of the method using a variety of geometric models from several simulation applications.

The remainder of this paper is organized as follows. Section 2 describes our ray-tracing technique and how it is accelerated to get better performance. Section 3 describes the overall procedures for constructing EB meshes using our ray tracing method. Section 4 shows performance results of the proposed method with several examples. Section 5 concludes this paper and suggests further future research.

2 Ray-Tracing Using Hierarchical OBB Trees

Triangle-based ray tracing is widely used in graphics applications, since most surfaces are visualized as sets of triangular facets. Recently, we have also used triangle-based ray tracing to support Monte Carlo radiation transport [7], where radiation is tracked as individual particles propagating through absorbing media. Our EB meshing procedure is based on the same ray tracing procedure described in Ref. [7]. This procedure is summarized below.

The input to our method is a non-manifold geometric model. The model is specified in the form of a Boundary Representation (Brep), as a collection of Vertices, Faces, Edges, and Regions, with topological relations between those entities. Models are accessed through the Common Geometry Module (CGM) [8], which is the same modeling interface used by the CUBIT mesh generation toolkit [9]. We read the facet-based description of the geometric model, including the topological relationships between the original geometric model entities (represented as groups of facets), into the MOAB mesh database [10]. Ray tracing is implemented in the DagMC library provided with MOAB.

Since ray tracing accounts for the overwhelming majority of execution time in Monte Carlo radiation transport, a great deal of effort has been made to optimize that computation. This work has been reported in [7], and is summarized below.

Construction of recursive space subdivision structure could be used to optimize ray-triangle intersections. However, axis-aligned boxes would have large volumes for any collection of facets not aligned with coordinate axes, increasing the likelihood of rays intersecting those boxes. In contrast, boxes oriented to minimize the extent in at least one box dimension are much less likely to be hit by a given ray, at the cost of having to transform the ray into the oriented box coordinate system. We have found that in practice the savings in ray-box intersections is well worth the cost of ray transformations.

The advantage of the oriented bounding box for ray-tracing is amplified when it is combined with hierarchical tree structure, referred to as an OBB tree. An OBB tree is a binary tree of oriented bounding boxes of polygons. OBB trees were originally developed to accelerate collision detection problem in robot motion planning [11]. Tree construction starts from the root bounding box node encompassing a collection of polygons. The set of polygons is divided geometrically into two subsets, and a bounding box is formed for each subset of polygons. The process is repeated until each leaf box encloses a specified number of polygons. Figure 1 shows an example OBB tree.

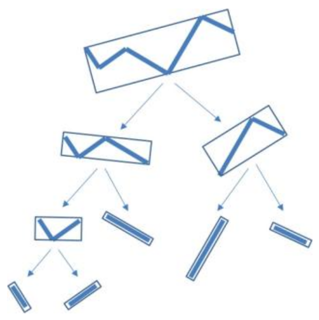
Fig. 1. OBB tree construction

In MOAB's OBBTree representation, the collection of facets for each original geometric surface forms the starting point of the OBBTree subdivision. Once the OBBTree has been formed for the facets from each geometric surface, the root nodes for all surfaces bounding each geometric region are combined, leaving one OBBTree for each Region in the original geometric model.

Ray tracing on a Region is performed by firing the ray first at the root box in the tree, first transforming the ray into the coordinate system of that node. If the ray hits a given node, the procedure is repeated down the tree, transforming the ray in the process. If any leaf nodes are intersected, the ray is tested against all triangles in that node. Valid ray-triangle intersections are returned in order of increasing distance from the ray starting point.

3 Embedded Boundary Mesh by Ray-Tracing

Given a facet-based geometric model, and the ability to compute ray-tracing on that model, it is straightforward to construct an EB mesh generation algorithm. This algorithm consists of the following steps:

- Initialize the geometric model and hierarchical OBBTree for ray-tracing
- Find the Cartesian box that surrounds all Regions of the model, and its subdivisions in cells
- Use ray-tracing along cell edges to find cell intersections with geometric boundaries
- Store the EB mesh on Cartesian cells

These steps are described in the following sub-sections.

3.1 Initializing Geometric Model and Hierarchical OBBTree

Geometric models are imported by using the Common Geometry Module (CGM) [8]. CGM provides a common interface for interacting with models stored in a variety of underlying solid modeling engines, including ACIS [12] and OpenCASCADE [13]. Both these engines support import and cleanup of IGES and STEP model exchange formats. CGM provides facet-based representations for each Edge and Face in the geometric model, and functions for evaluating overall model topology.

The MOAB mesh library [10] uses a data model consisting of mesh entities (vertices, triangles, etc.), entity sets (arbitrary collections of entities and other sets), the databased instance, and tags (a named datum assigned to the previous three data types). Entity sets can have parent/child relations with other entity sets; these are distinct from the “contains” relations described earlier.

We import the facet-based geometric model from CGM into MOAB, representing each geometric entity using an entity set, and topological relations between geometric entities using parent/child relations between the corresponding entity sets. This representation provides the basis for ray-tracing on the facet-based model.

A hierarchical OBB tree is constructed on the facet-based model in MOAB, with one tree for each geometric Region set. For simplicity, a single tree node is inserted at the top, and linked to the root node for all Regions. The OBB tree provides ray-tracing functions and the coordinates of a box surrounding the entire OBB tree.

3.2 Building a Cartesian Mesh

After constructing the OBB tree, the coordinates of the top-level box indicate the geometric extents of the model. These coordinates are used to create a Cartesian mesh encompassing the whole model. The number of divisions of that mesh in each coordinate direction is computed by using the number of facets and the box size, or from user input[1]. The Cartesian mesh is stored in MOAB's structured mesh representation [10]; this representation requires only about 24MB per million elements in the box, based on double-precision storage of vertex coordinates.

3.3 Firing Rays

In order to find intersections between cells and the model Faces, rays are fired along edges of the cells. Since we use an axis-aligned Cartesian mesh, edges in a column of cells line up into straight lines. A ray trace is performed along each of these lines, with the starting position of the ray at the surface of the Cartesian mesh; intersections with the model are returned as distances along that line and surfaces in the model that are intersected. Since we use equal spacing of elements in the Cartesian grid, these distances can be converted to fractional distances along edges in the grid. The surfaces intersected are used to assign the model Regions to sections of edges between intersections, starting with a designation of "outside", since the ray begins on the boundary of the Cartesian mesh outside any model Regions.

Rays are traced in the three coordinate directions, at each interior node on the corresponding coordinate plane. This process results in ray traces along all interior edges of cells in the mesh. If the number of divisions on each side of the box is N, then only a total of $3(N-1)^2$ ray tracing operations is performed, each operation returning all Face intersections along the ray (Fig. 2). The second figure in Fig 2 shows ray-tracing covers N edge intersection test with a ray.

[1] Scaling numbers discussed in this paper assume a single number of divisions on all sides for simplicity only.

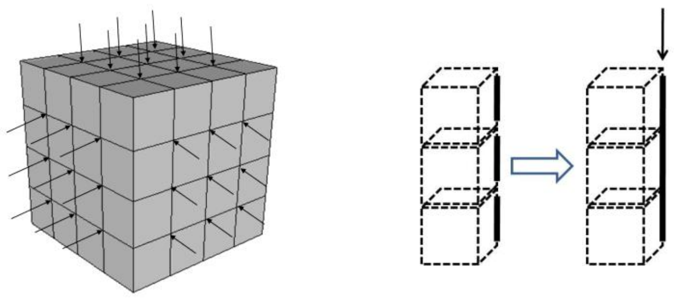

Fig. 2. Rays fired in three directions along cell edges, A ray covers N edges

If the intersection test is performed edge (cell) by edge (cell) like in other methods, abundant tree searching has to be performed to the non-boundary edges as in Fig 3. The search in the Fig 3 is not stopped until SS (Spatial Search) 4 block dose not have any facet triangles inside and conform that the edge is not on the boundary.

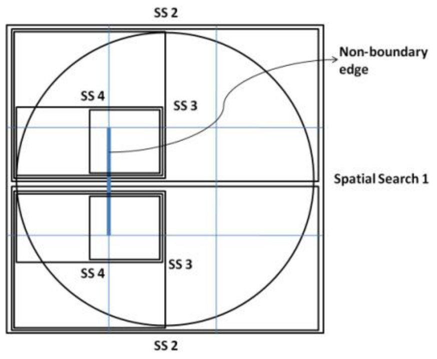

Fig. 3. Spatial search is performed for non-boundary edge

Ray-firing function of our implementation returns intersected triangles and distances from ray starting point. Since OBB tree is constructed for geometry surfaces, it also returns which surface is hit. With these information and previous edge status, each edge and element is easily determined if it is inside, outside or boundary to geometry and edge cut fraction information can be obtained for boundary elements. To avoid cell by cell job such as setting all element status, element status default value is set and only the elements near intersections are considered to the status determination and have real values.
In rare cases, ray tracing fails to find correct intersections with surfaces that they should. This failure is due to the use of non-watertight faceting returned by CGM and precision issues detecting ray-triangle intersections.

Modifying the faceting of a non-watertight geometric model is outside the scope of this paper, but is reported elsewhere [14] [15]. If the precision issues are detected such that ray intersect triangle edges or vertices, the ray is displaced slightly and the ray trace is repeated. This strategy fixes all examples we have seen of this problem. The intersection along this modified ray is moved to the original edge at the appropriate position, with that offset very small relative to the cell edge length.

3.4 Storing Information

MOAB allows applications to assign tag values to the entities in a mesh. Tags are uniquely identified by a name, and are referenced by a tag handle for efficiency. MOAB provides tags of three storage types: “sparse” tags, stored in (entity handle, tag value) tuples; “dense” tags, stored as an array of tag values for contiguous sequences of entity handles; and “bit” tags, in which each tag value is a user-defined number of bits. Dense tags are more memory-efficient when assigning a tag to many entities in a mesh. Tags of all three storage types can be created with default values; for cases where the majority of entities will be assigned the same value for a given tag (e.g. all cells on the interior of a solid), a default value eliminates the storage for this value on those entities.

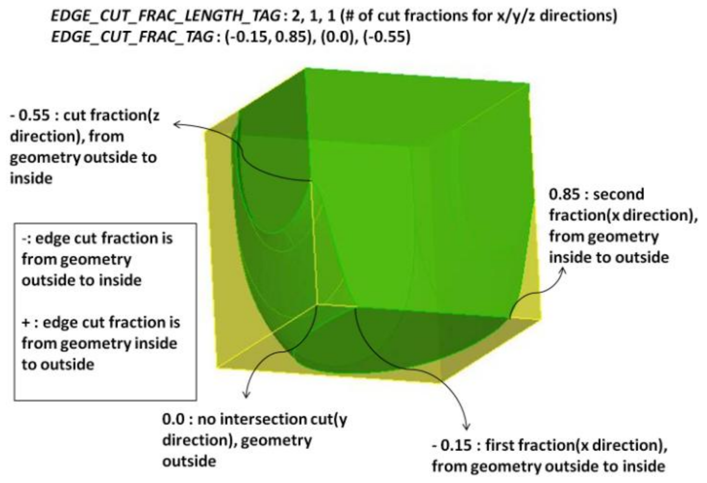
EDGE_CUT_FRAC_LENGTH_TAG : 2, 1, 1 (# of cut fractions for x/y/z directions)
EDGE_CUT_FRAC_TAG : (-0.15, 0.85), (0.0), (-0.55)
- 0.55 : cut fraction(z direction), from geometry outside to inside
0.85 : second fraction(x direction), from geometry inside to outside
-: edge cut fraction is from geometry outside to inside
+ : edge cut fraction is from geometry inside to outside
0.0 : no intersection cut(y direction), geometry outside
- 0.15 : first fraction(x direction), from geometry outside to inside

Fig. 4. An example of edge cut fraction tag information on boundary cell.

The EB mesh generation method described here stores two types of information. First, each cell in the Cartesian mesh is marked as inside (0), outside (1), or boundary (2), using the 2-bit tag E*LEM_STATUS_TAG*.

Second, for cells marked as boundary, the position of intersection along each edge of the cell must be marked. Although each cell is bounded by 12 edges, only three edges for each cell must be marked, corresponding to the left, bottom, and front edges; the other edges are marked by neighboring cells. Edge fractions are measured from the left, front and bottom corner of boundary cell to x/y/z directions as in Fig. 4.
If a ray hits the boundary from inside to outside, the fraction value is stored with a negative sign. In contrast, for the case of a ray intersection from outside to inside, the value is marked as positive. Since multiple intersections can exist in each direction, the *EDGE_CUT_FRAC_LENGTH_TAG* is created to store the numbers of edge fraction values on the x, y, and z edges connected to the bottom/left/front corner of each cell. The *EDGE_CUT_FRAC_TAG* stores an arbitrary number of double precision values, listing all the intersection points along the x, then y, then z edges of a cell. Edges that are completely outside (inside) the Region are designated with zero (one) for edge intersection fractions.
All tags except *ELEM_STATUS_TAG* are sparse tags, whose default values are chosen to indicate an interior cell. This tag is assigned explicitly to boundary and exterior cells. This approach saves substantial amounts of memory when there is a majority of interior cells, which is usually the case (since the Cartesian box is chosen to fit the geometric model tightly).

3.5 Calculating the Volume Fraction

Multiple material compositions are required in each mesh cell for many simulation analyses, such as deterministic radiation transport. As an approximation to volume fractions within a cell, we use a method similar to that of Riper [16] [17], where ray tracing is performed on a subdivision grid over the cell. The volume fraction for a given Region intersected by the cell is equal to the total ray traversal length in that Region, summed over all rays fired over the subdivided cell, normalized by the total ray traversal length over the cell. The volume fraction calculation is performed by firing sub-rays to 3 directions parallel to x/y/z coordinates in boundary cell. All ray lengths inside each material geometry are summed and divided by all fired ray length sum as in equation (1) [17].

$$F_M = \left(\Sigma_j L_{Mj} \right) / (DJ)$$

F_M : fraction of material M in a cell
L_{Mj} : ray path length in material M for ray j
J : total number of rays

$$D : \text{sum of all ray lengths} \tag{1}$$

The number of divisions in the subdivision of each boundary cell is equal in our implementation and is assigned by the user.
To store these information for boundary cells, *MAT_FRAC_ID_TAG* and *VOL_FRAC_TAG* are created as sparse tags for each material. Each tag has an arbitrary number of material ids and volume fractions.

3.6 Export Meshes or Query Function as Library

The Cartesian grid, edge intersection data, and (optionally) volume fractions stored on that grid, can be exported from MOAB in a variety of formats. Alternatively, the data can be queried directly from MOAB by using existing functions for accessing mesh and tag information.
In addition, in order to avoid the overhead of interacting through files, the EB meshing method can be linked as a library directly to applications. Along with functions for specifying the Cartesian box divisions and for requesting the generation of the EB mesh, there are several query functions for retrieving cell inside/outside status, edge-cut fractions and volume fractions of boundary cells. Currently, these functions are planned to be used for electromagnetic analysis [18] and radiation transport simulation [19]. The API specification for these functions appears in the Appendix of this paper.

4 Results

The algorithm described in Section 3 has been tested on several example models of varying complexity. Performance data is measured by using a Linux workstation with Xeon 3GHz CPU and 16GB of memory.

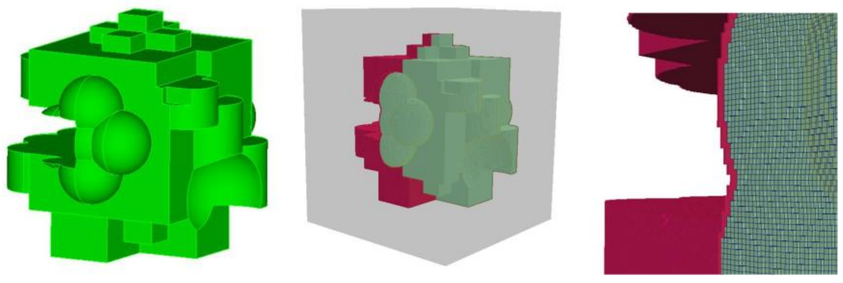

Fig. 5. Example model and Cartesian meshes created from many primitives

An example model used to test the method for a model of moderate complexity was created by combining many primitive geometries as in Fig. 5. Cartesian mesh elements, distinguished as different colors by attached element status tags, are also shown in the Fig. 5.

Cartesian meshes were produced with different mesh sizes; their computation times and maximum memory usages are plotted in Fig. 6. Mesh generation took 4.64 seconds including geometry importing time to produce 10.5 million elements with a maximum memory of 432 MB.

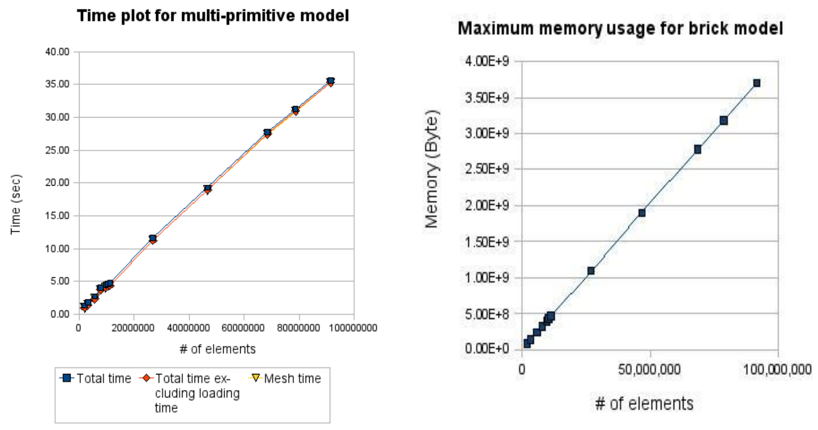

Fig. 6. Timing and memory usage results for multi-primitive model

The next examples are standard STL format files produced by 3D scanning, which have complex boundary representations as in Fig. 7 [20].

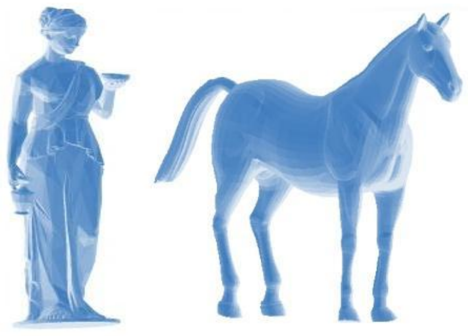

Fig. 7. STL 3D “statue” and “horse” models [20]

Figure 8. shows their Cartesian meshes produced by EBMesh.

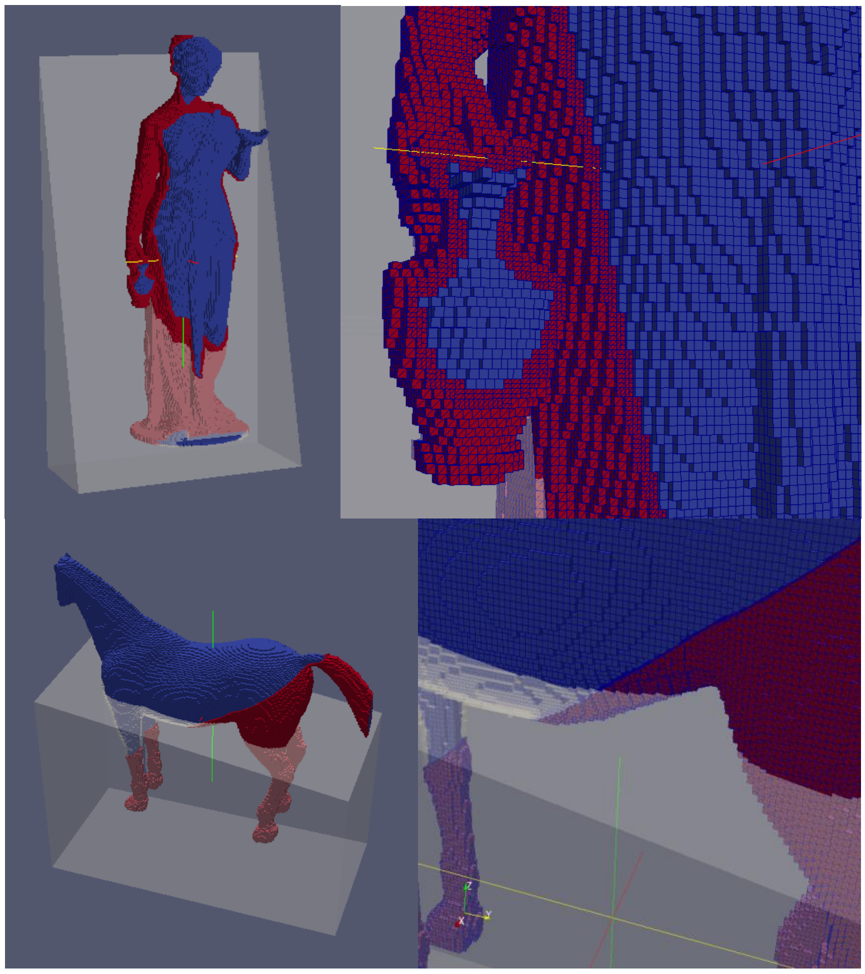

Fig. 8. Cartesian mesh elements of 3D STL models

Some performance results are also plotted with different mesh sizes in Fig. 9. It takes 6 seconds to produce 11.8 million elements with a maximum memory of 480 MB for the first statue model and 2.3 seconds for 8.8 million elements with a maximum memory of 360 MB for the second horse model.

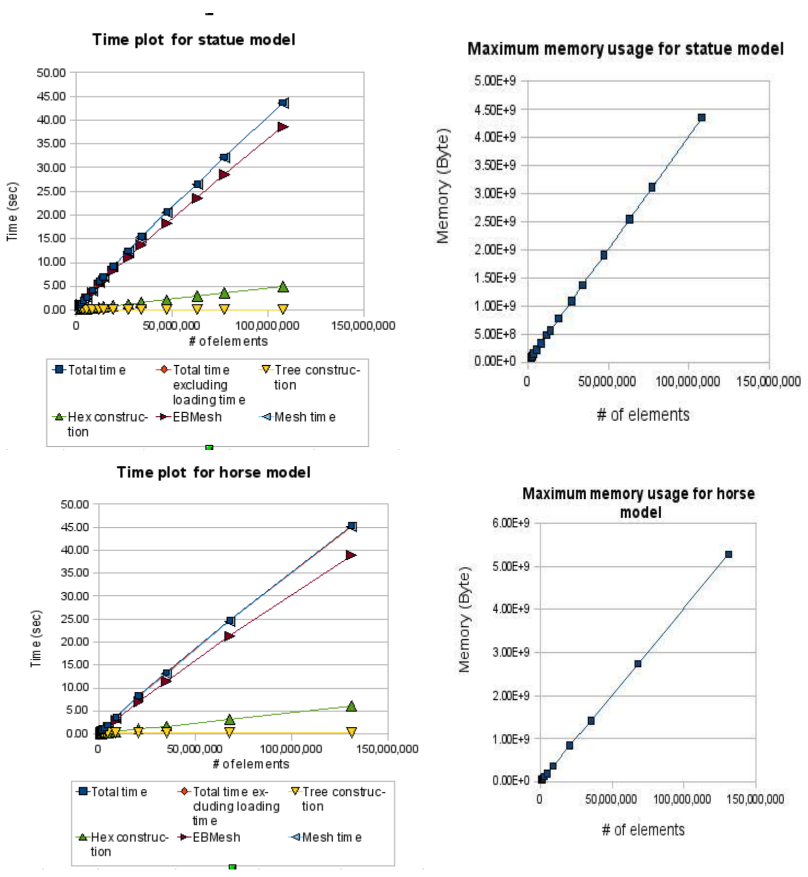

Fig. 9. Timing and memory usage results for “statue” and “horse” STL models

The last example is an accelerator cavity model for the International Linear Collider (ILC), used to optimize electromagnetic performance of the cavities by adjustment of geometric design parameters [21]. In Fig. 10, the model geometries are shown, including the complex higher-order mode (HOM) coupler region.

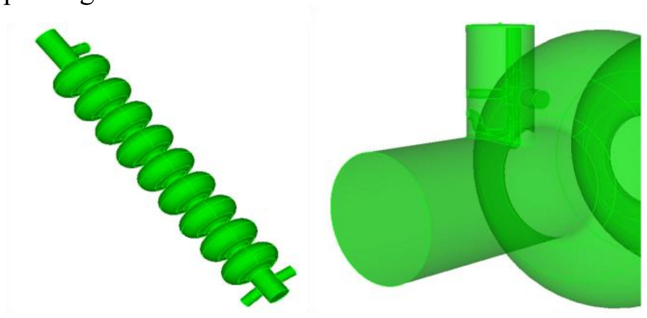

Fig. 10. Accelerator cavity models

Their meshed results are shown with different element status in Fig. 11.

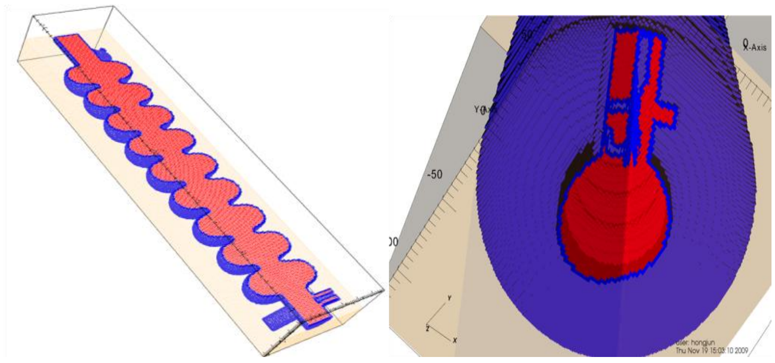

Fig. 11. Cavity model mesh results

In Fig. 12, timing results and maximum memory usages are plotted with different mesh sizes. Mesh generation time is 68.8 seconds for this model, including geometry loading time for 10.7 million elements with a maximum memory of 582 MB. Since the cavity model is very complex, the time for importing, faceting and tree construction takes large portions of the overall computation time.

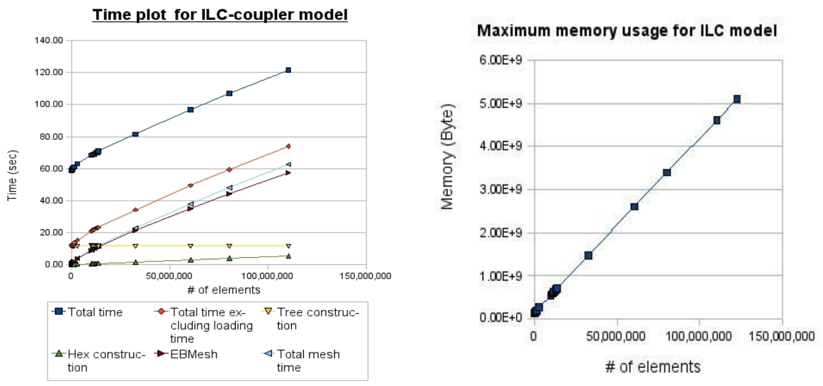

Fig. 12. Timing and memory usage results for ILC-coupler cavity model

The complexity of our algorithm is estimated to be scaled by N^2, as discussed in Section 3. That is, total computation time can be expressed as $T = A_1N^2\log N_t + A_2N_t + A_3$. The first term corresponds to the EB meshing time. The second term is for triangle faceting and tree construction, and depends mostly on geometric model complexity. The third term is for some constant-time jobs involved. When the same input geometry is used for timing, the second and third terms are constant, with the remaining time proportional to N^2. Therefore, a log-log plot of N and meshing time T_M is expected to have slope 2. In Fig. 12, the meshing time for the ILC-coupler model is plotted when N is increased; it has a slope of approximately 2, as expected.

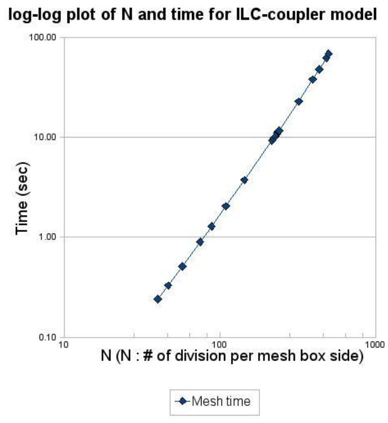

Fig. 13. Log-log plot of N and meshing time has a slope of 2

5 Conclusions and Future Work

We have presented an embedded boundary Cartesian mesh generation algorithm that produces better performance than traditional EB approaches for complex geometries. It reduces mesh computation time as much as of 1/N compared with the cell-by-cell method by checking intersections edge line by line. As a result, all mesh elements are distinguished as being inside, outside, or on the boundary of the geometry model. Boundary cells also have edge-cut information stored as a tag. Optionally, volume fraction information about each material for boundary cells can be calculated. The mesh and related information can be written to a variety of mesh file formats or used directly by an analysis program. This work is implemented as an open-source code in Ref [22].

We envisioned several avenues of future work. For example, it may be useful to perform adaptive mesh refinement in the region of rapidly changing curvature geometry. The current volume fraction calculation function is similar enough to the refinement that we can approach it as a start. With

curvature information directly obtained from geometry, we will produce better refinement meshes.
Another future activity is generating a hex-dominant mesh by making polyhedral meshes instead of boundary hexes. Combined with OBB, which makes meshes oriented to natural geometry direction, it may produce good-quality meshes independent of geometry transformation.
Lastly, parallel implementation of EBMesh and actual CAD surface intersection test will be considered for faster and more accurate results.

Acknowledgements

This work was supported by the US Department of Energy's Scientific Discovery through Advanced Computing program under Contract DE-AC02-06CH11357.

References

1. Aftosmis MJ, Berger MJ, Melton JE (1998) Robust and efficient Cartesian mesh generation for component-based geometry, AIAA Journal, 36(6), pp 952-960
2. Colella P, Graves D, Ligocki T, Trebotich D, Straalen BV (2008) Embedded boundary algorithms and software for partial differential equations, SciDAC 2008, Journal of Physics, Conference Series 125
3. Nieter C, Cary JR, Werner GR, Smithe DN, Stolz PH (2009) Application of Dey-Mittra conformal boundary algorithm to 3D electromagnetic modeling, Journal of Computational Physics, 228(21), pp 7902-7916
4. Pattinson J (2006) A cut-cell, agglomerated-multigrid accelerated, Cartesian mesh method for compressible and incompressible flow, Ph.D. thesis, University of Pretoria
5. Bonet J, Peraire J (1991) An Alternating Digital Tree (ADT) Algorithm for Geometric Searching and Intersection Problems, Int. Journal of Numerical Method and Engineering, 31, pp 1-17
6. Colella P et al. (2009) Chombo Software Package for AMR Applications Design Document, Applied Numerical Algorithm Group, Lawrence Berkeley National Laboratory, Berkeley, CA, April
7. Tautges TJ, Wilson PPH, Kraftcheck JA, Simth BM, Henderson DL (2009) Acceleration Techniques for Direct Use of CAD-Based Geometries in. Monte Carlo Radiation Transport, Proceedings of International

Conference on Mathematics, Computational Methods & Reactor Physics, Saratoga Springs, NY, May 3-7

8. Tautges TJ (2005) CGM: A geometry interface for mesh generation, analysis and other applications", Engineering with Computers, 17, pp 486-490
9. Sjaardema GD, Tautges TJ, Wilson TJ, Owen SJ, Blacker TD, Bohnhoff WJ, Edwards TL, Hipp JR, Lober RR, Mitchell SA (1994) CUBIT mesh generation environment, Volume 1: Users manual, Sandia National Laboratories, May
10. Tautges TJ, Meyers R, Merkley K, Stimpson C, Ernst C (2004) MOAB: A Mesh-Oriented Database, Sandia National Laboratories
11. Gottschalk S, Lin MC, Manocah D (1996) OBBTree: a hierarchical structure of rapid interference detection, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp 171-180
12. ACIS 3D Modeling (2010), Spatial Technology Inc., http://www.spatial.com/products/3d-acis-modeling
13. OpenCASCADE Technology (2000-2010), http://www.opencascade.org/
14. He XJ, Chen YH (2006) A Haptics-guided Hole-filling System Based on Triangular Mesh, Computer-Aided Design & Applications, 3, (6), pp. 711-718
15. Boonma A (2006) Haptic-Based Sharp Edge Retaining and Gap Bridging Algorithms for Computer Aided Design (CAD) and Reverse Enineering (RE), Master. Thesis, North Carolina State University
16. Riper KAV (2003) Mesh and Volume Fraction Capabilities in Moritz, KAV, Workshop on Common Tools and Interfaces for Deterministic Radiation Transport, for Monte Carlo, and Hybrid Codes (3D-TRANS-2003), 2003, Issy-les Moulineaux, France, pp 25–26, September
17. Riper KAV (2006), Moritz Geometry Tool, http://www.whiterockscience.com/moritz.html
18. Stoltz P, Veitzer S, Niether C, Messmer P, Amys K, Cary J, Lebrun P, Amundson J, Spentzouris P, Kim HJ, Tautges TJ (2010) Recent Progress in Accelerator Physics Simulations with the VORPAL code, SciDAC 2010, in preparation
19. Smith B, Wilson PPH, Sawan ME (2007) Three dimensional neutronics analysis of the ITER first wall/shield module 13, 22nd IEEE/NPSS Symposium on Fusion Engineering-SOFE 07, Piscataway, NJ: Institute of Electrical and Electronics Engineers Inc.
20. STL files (2000-2010), Jelsoft Enterprises Ltd., http://forum.carvewright.com/showthread.php?t=12452&page=2

21. Ko K et al. (2005) Impact of SciDAC on Office of Science Accelerators through Electromagnetic Modeling, SciDAC2005, June
22. EBMesh Tool, http://trac.mcs.anl.gov/projects/fathom/wiki/EBMesh

Appendix A

Presented here is the API specification for the query functions.

- bool get_grid_and_edges (
  @return double boxMin[3] : entire grid box minimum
  @ return double boxMax[3] : entire grid box x maximum
  @ return int nDiv[3] : number of divisions
  @ return vector<int> cutCellIndices: boundary cell index vector
  @ return vector<int> cutFracLength : number of edge cut fractions for boundary cells
  @ return vector<double> cutFraction : edge cut fraction vector
  );

- bool get_volume_fractions (
  @ return vector<int> materialID : material id vector
  @ return vector<int> boundaryCell : boundary cell vector
  @ return vector<double> volumeFrac : volume fraction double vector
  );