Algorithms and Data Structures for Handling a Fully Flexible Refinement Approach in Mesh Generation

N. Hitschfeld
Dpto. Ciencias de la Computación, Univ. de Chile
Blanco Encalada 2120, Santiago, CHILE
e-mail: nancy@dcc.uchile.cl

Abstract

This paper presents new algorithms and data structures required for the generation of grids based on mixed element trees and using flexible refinement approach. Mixed element trees is an extension of modified octrees that uses several well-shaped primitives such as cuboids, prisms, pyramids and tetrahedra as internal nodes. A flexible refinement approach fits the object geometry and fulfills the required mesh density by partitioning the elements at an optimal point at each refinement step. This allows the fitting of more general 3-D object geometries and the reduction of grid points in comparison with previous grid generators but it requires more complex algorithms and data structures in its implementation.

1 Introduction

The numerical solution of partial differential equations is invaluable in design and optimization in many fields of engineering. The spatial discretization (grid) is the key to the accuracy of the computed solution. An appropriate grid has to fulfill several requirements, for example: (1) it must provide a reasonable approximation of the geometry, (2) it has to fulfill the density requirements and (3) single cells have to fulfill the restriction of the underlying numerical method.

In order to generate such a grid, grid generators typically use iterative refinement of coarse elements to obtain a required tessellation. Iterative refinement can be implemented by either bisecting the element edges (*bisection* based approach) or dividing the element edges at an arbitrary position (*intersection* based approach). Grid generators normally used a *bisection* based approach (BBA) because of simplicity. According to our knowledge, the *intersection* based approach (IBA) was presented the first time in [1] for the generation of proper 3-D grids for semiconductor devices.

This paper presents an implementation of the *intersection* based approach on mixed-elements trees[2, 3]. It describes the algorithm to generate a grid and suggests algorithms and data structures to consistently keep the geometrical information of the grid. This is quite difficult because edges and faces are normally partitioned more than once and at different locations.

2 Term definition

This section defines a set of terms that are considered important to understand the current article.

Object geometry is defined by the boundary and material interfaces of the object. In the current application, 2-D geometries are defined using polygons and 3-D geometries using polyhedra.

Element is the term used to denote a mesh cell such as triangles or quadrilaterals in 2-D, and tetrahedra, prisms, pyramids and cuboids in 3-D.

1-irregular element is an element whose edges are split at most once. In other words each element edge has at most one green point.

Well shaped elements are elements that fulfill the restrictions of the underlying numerical method. For example: 2-D elements that do not have obtuse angles are required for meshes used with the control volume discretization method.

Aspect ratio is the value obtained by dividing the length of the longest element edge by the length of the shortest element edge.

Green point is a partition vertex that lies on an element edge but it is not an endpoint of that edge. Green points are also called non-conformal points.

Intersection points are points generated by the intersection among the element edges and faces with the object geometry.

Edge neighbors are the elements that share an edge.

Face neighbors are the elements that share a face. This is only possible in 3-D.

Neighboring elements are elements that share part of an edge or a face.

Edge partition is the term used to describe the partition of an edge in two smaller edges.

Face partition is the term used to describe the partition of a face in two or more smaller faces.

Element refinement is the term used to describe that an element was partitioned into several smaller elements. The number of new elements depends on the directions the element was refined.

Mixed element trees is an extension of modified octrees. It accepts 3-D elements such as cuboids, prisms, pyramids and tetrahedra as internal nodes (octrees accept only cubes). In 2-D, it is an extension of modified quadtrees. The mesh generator uses this data structure to handle the geometrical information of the mesh elements.

Tree node contains the geometrical information of one mesh element. A tree node is a leaf if the associated element has not been refined yet.

Parent node. An internal node n is called parent of the ones that were generated when the element stored in the node n was refined. The new generated elements are stored as sons of the node n. The parent relationship depends on the order the elements are refined.

Parent edge. An edge e is a parent edge of the edges that were generated when the edge e was refined.

Parent face. A face f is a parent face of the faces generated when the face f was refined.

3 Previous work

In the literature, many approaches to generate grids have been described. Most of them are triangulations of the domain (triangles in 2-D and tetrahedra in 3-D) because they have shown to be very flexible to model complex geometries and very adequate for finite element methods[4, 5, 6]. However, if the equations are solved using a control volume discretization or box method(BM)[7] a strong restriction on the angles of the elements must be imposed. The use of elements of different shape such as rectangles in 2-D and cuboids, pyramids and prisms in 3-D has shown to be more appropriate than the use of only triangles in 2-D and tetrahedra in 3-D [3]. In addition, if the object to be modeled has very thin layers of different materials, as occurs in semiconductor devices, the number of tetrahedra needed for a proper 3-D mesh can be too large if very small and large angles must be avoided. The angle is independent of the aspect ratio in some of the well-shaped elements.

A first 3-D grid generator based on mixed element trees (MET) was developed to handle several kind of elements. MET generalize the modified octree approach[8] in several aspects. One of them is that the whole device is no longer encapsulated in a single octree, but partitioned in a set of basic elements. Each of them becomes the root of a mixed element tree. Each tree keeps track of the mesh elements generated by the refinement of the element stored in the root and of the elements generated by the refinement of its descendants. The shape of the elements depends on the restrictions of the underlying numerical method. For example, in semiconductor device modeling, no Voronoi point can lie outside of the elements used to fit the device geometry. The Voronoi point is the center of the circumsphere defined by the element vertices. Elements that satisfy this property are for example: cubes, rectangular prisms, rectangular pyramids and some kind of tetrahedra.

The first implementation of a grid generator based on a MET is described in [3, 9]. The most serious problem of this implementation originates from the simple algorithm that fits the original device geometry. This algorithm generates an initial (tensor product based) grid that is a complete partition of the object geometry (i.e. mesh elements have no green points). A complete partition is required in order to use later a BBA to fulfill the density requirements. During the generation of the initial grid, small geometry features are propagated (by inserting planes in 3-D) to the boundaries of the object. A 2-D example is shown in Fig. 1 and a 3-D example in Fig. 3(a). Therefore, the initial grid contains a high number of unnecessary elements with a very bad aspect ratio. In addition, the repetitive generation of new points due to intersections between the inserted lines (planes in 3-D) and some boundary or material interfaces makes impossible to fit several object geometries (see the 2-D example in Fig. 1).

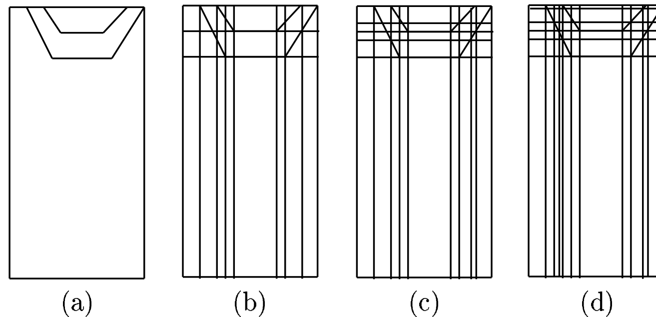

Figure 1: Fitting a 2-D object geometry using a tensor product grid. (a) Object geometry (b), (c) and (d) sequence of steps to fit the device geometry. After the step in (d) the geometry is still not optimally fitted.

4 *Intersection* based approach in grid generation

In order to improve the problems enumerated in the previous section, a new algorithm based on IBA was developed. This algorithm does not use a tensor based approach.

The geometry is fitted by refining the grid elements at the best possible point (see Fig. 2). The best point—the one whose associated refinement generates sons with the

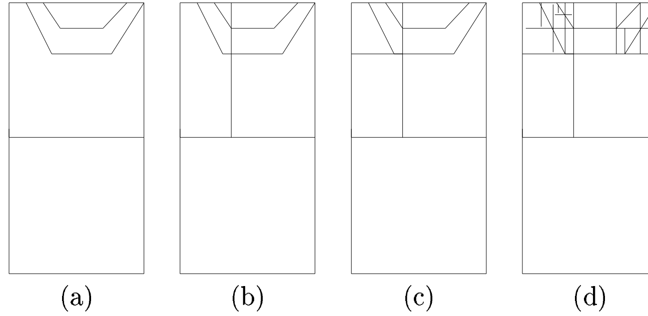

Figure 2: Fitting a 2-D device geometry using the *intersection* based approach. (a), (b), and (c) first steps to fit the device geometry (d) final initial grid.

smallest aspect ratio —is chosen from the available green points and intersection points. Elements are bisected for generated sons with bad aspect ratio. Fig. 3(b) shows the initial covering of a 3-D example. Notice that this initial covering of the geometry has a lot of green-points and needs fewer points than the tensor product covering used in the old approach (Fig. 3(a)).

After the geometry is completely fitted, the initial covering is further refined until the density requirements are fulfilled. Elements are partitioned in the required direction according to the best located green point. Because of the better fitting of the device geometry, the new algorithm fulfills the required point density using fewer points (Fig. 4).

The grid is made 1-irregular before looking for proper tessellations. Subsequently, the algorithm checks the *splittable* condition, i.e., a condition that guarantees the existence of a proper tessellation. If an element is non-splittable, proper points are inserted by looking inside the element.

Once all elements are splittable, each local tessellation is computed using an algorithm to compute Delaunay tessellations inside of a convex-hull (all basic elements are convex) [10].

5 New algorithms and data structures

One of the difficulties in the implementation of the IBA is to consistently and efficiently handle the geometrical information (points, edges, faces) when an element is refined.

The refinement of an element requires the knowledge of all the grid points already inserted on edges and faces of the neighboring elements. This is a difficult task because (1) neighboring elements can lie at a different tree depth and (2) edge (face) neighbors do not need to be refined at the same point. Therefore, the sons of the refined element are not necessarily edge (face) neighbors.

The case (1) is illustrated in Fig. 5. The rectangle **A** is refined in two smaller rectangles **B** and **C**. The rectangle **B** is refined again into **D** and **E**, and the rectangle **E** is refined in **F** and **G**. The rectangles **D**, **F** and **G** are neighboring elements of the rectangle **C** but they lie at a different tree depth. Fig. 6 illustrates the case (2). The rectangle **A** is refined in two smaller rectangles **B** and **C**. **B** and **C** are edge neighbors. The shared edge is **e**. As

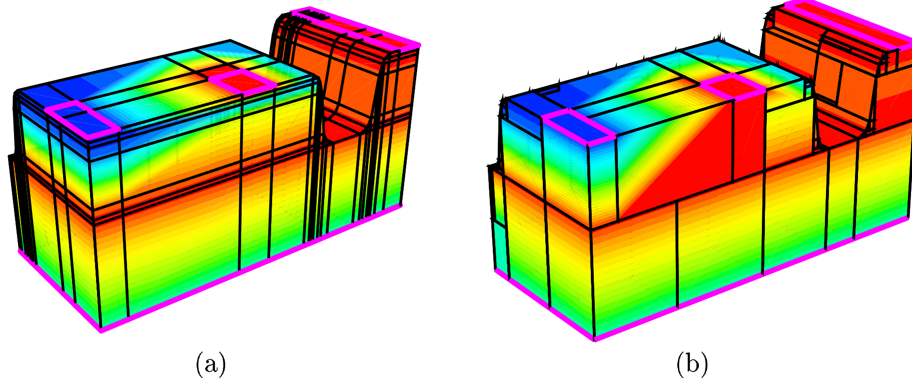

Figure 3: Fitting the device geometry for the bipolar transistor (a) tensor product grid (old approach): 2365 pts (b) *intersection* based approach: 881 pts.

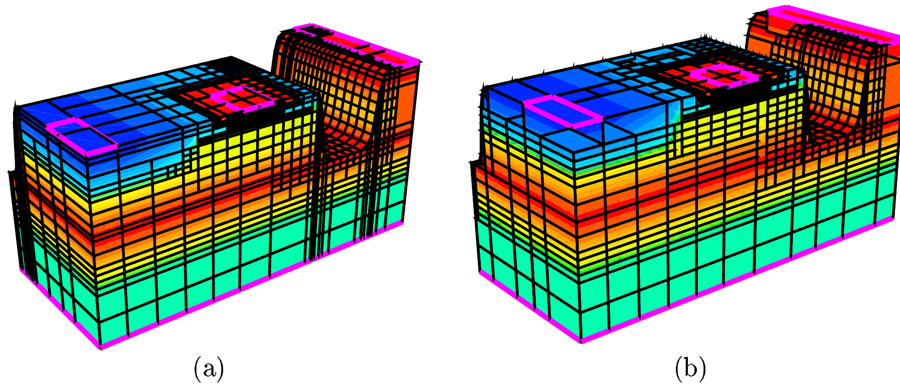

Figure 4: Achieving the desired mesh density for the bipolar transistor (a) *bisection* based approach (old approach): 10,376pts (b) *intersection* based approach: 7,097pts.

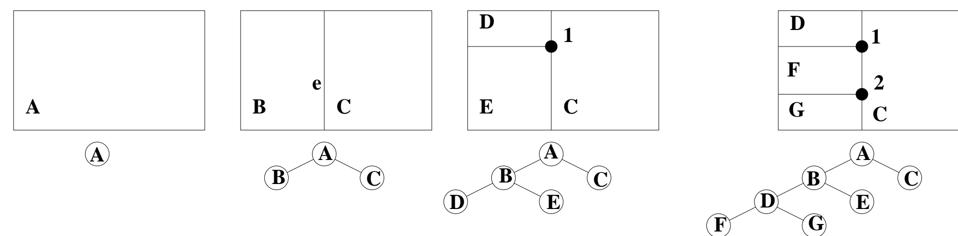

Figure 5: Neighboring elements. Example of an element refinement and the associated tree structure.

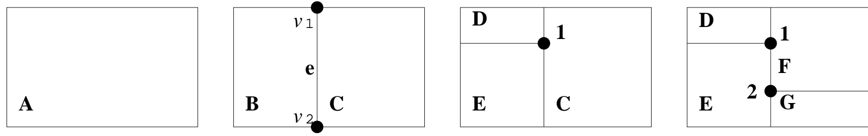

Figure 6: Refinement of a rectangle.

the refinement of the rectangle **B** generates the point **1** on edge **e** and the refinement of rectangle **C** generates the point **2** on edge **e**, the sons of rectangles **B** and **C** are not edge neighbors.

The refinement of an element generates a partition of its edges and faces. The algorithms and data structures presented in this section guarantee that each new partition of an edge or a face generates all the necessary information in such a way that any further edge or face partition has access to all the points – between its endpoints or inside the face – generated in previous partitions. Subsequently, the information required to refine an element can be locally obtained by looking at the edges and faces of this element and the edge and face descendants.

5.1 Storage of the geometrical information of the grid

In many grid generators, grid points, edges, and faces are stored explicitly in tables. Each grid point is associated with an index that indicates its location in the point table. Edges are represented by the endpoint indices. The lower index defines its entry index in the edge table where the higher index and related information is stored. Edge information inserted at the same entry is stored in a sorted list. Faces are described by their point indices. The lowest index is used as the entry index in the face table. Collisions are solved by storing the information in a sorted list according to the second lowest face index.

The data structure designed to represent a mixed element tree node contains information about the element type and the global indices of each element corner. As the corner indices are stored in a pre-defined order the information of its edge endpoints and face points can be easily obtained.

5.2 Edge data structure

Every edge stores only one green point. This green point corresponds to the location in which the edge was refined the first time. The other green points are obtained by looking at its edge descendants. For example, the edge e of Fig. 6 if defined by the endpoints v_1 and v_2. e stores only the green point 1. The green point 2 is stored between the green point 1 and v_2 which is an edge descendant of e.

Fig. 7 shows the edge data structure. It contains information about its endpoints, the first point (**first_green_pt**) inserted between its endpoints, its parent edge list, and a list of the mixed element tree leaves (**tree_nodes**) whose elements contain this edge. An edge can have different parent edges when neighboring elements share a part of an edge (face). Fig. 8 shows a 2-D example where the rectangles **D** and **E** share the edge e. The edge e has two parent edges: e_1 and e_2.

Edges are deleted only if they do not belong to any mesh element (the list **tree_nodes** is empty) and none of their parent edges is alive (the list **parent_edges** is empty). At some time, an edge may not belong to any element, but it can be later generated again if one of its ancestor is still alive. The edge e of Fig. 8 does not belong to any element.

The *Edge* structure allows the searching of edge neighbors; in other words, elements that share an edge or a part of an edge. The pointers to the elements sharing a particular

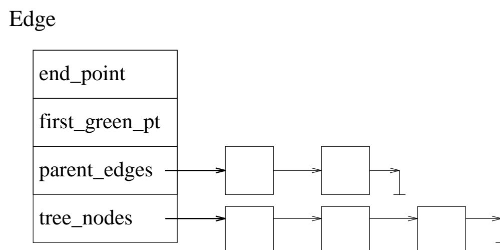

Figure 7: Information stored for each edge

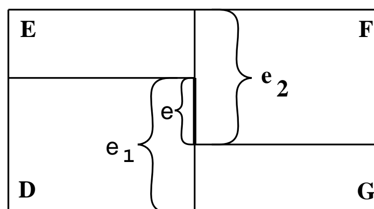

Figure 8: The edge e has two parents p_1 and p_2

edge are found in the **tree_nodes** list. The **tree_nodes** field associated with the edge descendants gives the pointers to the elements sharing a part of the edge.

5.3 Algorithm to insert a point on an edge

Fig. 9 shows the algorithm that inserts a point p in an edge with endpoints $\{v_1, v_2\}$. The algorithm is divided in four cases: (1) if the current edge is not partitioned, two new edges are generated (see Fig 10(a)) and v_1 and v_2 stores p as its first green point, (2) if the edge is already partitioned at that point, no information is added, (3) if the point lies between v_1 and the current green point gp, the point is recursively inserted between v_1 and gp. Notice that the edge defined by v_1 and gp can already have green points. For this reason the same algorithm is applied to insert the point between v_1 and gp. When this finishes, a new edge is created with endpoints $\{p, v_2\}$ and green point gp. Edges $\{v_1, p\}$ and $\{p, v_2\}$ belong to the sons of the refined element. The insertion of gr in $\{p, v_2\}$ guarantees that the new son edges have access to all the points already existing between their endpoints. (4) The symmetrical side of (3), (see Fig 10(b)).

5.4 Face data structure

In 3-D grid generation, besides the edge information, it is necessary to store similar information for each face. Instead of points, edges are inserted when an element face is refined. The face partition also originates the insertion of points on its edges.

Fig. 11 shows the face data structure. It contains information of the face points, the first edge used to divide the face (**first_gr_edge**) and the two generated son faces. In addition, it contains a list of the parent faces and a list to the tree leaves whose elements contain the face. The elements sharing a face (face neighbors) are obtained from the list of tree leaves. Elements sharing a part of a face are found by searching in list of tree leaves of the face descendants.

```
Integer InsertPointInEdge(edge, point){
  /* edge = (v1,v2) */
  index = Search(edge, point);
  if ( index equal to green_point ) return index;
  if ( edge without green_point )
    point_index = new entry for point;
    green_point = index;
    Create and link son edges. Update parents;
    /* Figure 10(a) */
  else if ( point lies between v1 and green_point )
    index = InsertPointInEdge( (v1,green_point), point)
    create new edge (index, v2) with green_point as its
    green_point. Update parents;
  else  /* point lies between green_point and v2 */
    index = InsertPointInEdge( (green_point,v2), point)
    create new edge (v1,index) with green_point as its
    green_point. Update parents;
    /* Figure 10(b) */
  end if
  return index;
}
```

Figure 9: How to update the edge information

5.5 Algorithm to insert an edge on a face

The algorithm that inserts an edge on a face is much more complicated than the one described for the insertion of a point on an edge. Fig. 12 shows the most important cases (shaded faces represent recursive calls): (a) corresponds to the most simple case. If a face is not partitioned, the current edge becomes a green edge and two new faces are created, (b) represents one of the cases where the face is already partitioned and its green edge is parallel to the edge being inserted. The edge is recursively inserted in the corresponding son face. When this process finishes, the face that represents the current partition is created and its son faces are linked, and (c) represents the case where the green edge is perpendicular to the edge being inserted. The current edge is intersected by the green edge and divided into two parts which are inserted recursively in the corresponding son face. Once this finishes, the

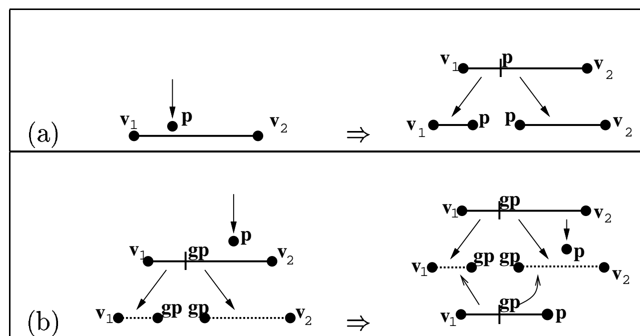

Figure 10: Handling green points on an edge

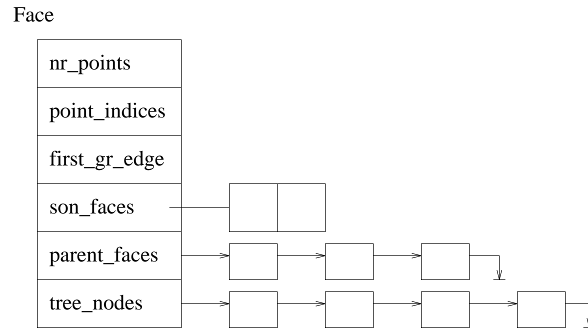

Figure 11: Information stored for each edge

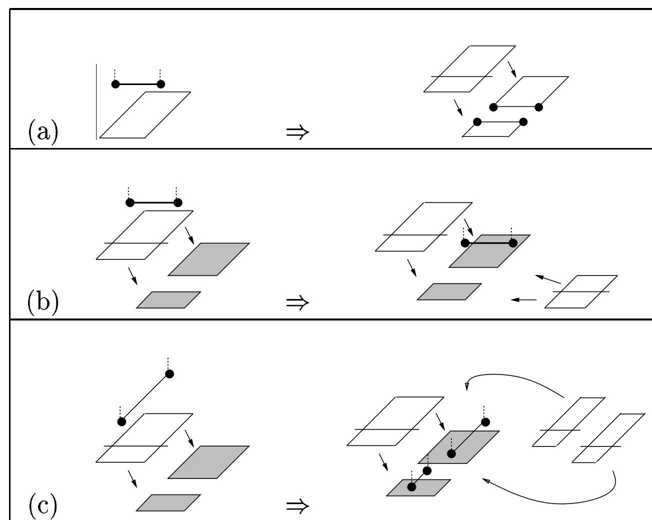

Figure 12: Handling green-edges and green-points on a face

two new faces corresponding to the current face partition are created. Each one is divided by the part of green edge by which is intersected.

The previous algorithm guarantees that at every time a face is refined, all the required information can be obtained by only looking at it and its face descendants.

6 Analysis and comparison

The algorithms and data structures presented in the previous section permit to decide locally (i.e, inside of each element) at which point an element is refined and only the necessary point information is looked at. As shown in the following, the cost of a refinement is $O(n)$ (n is the total number of green points of the current element).

Let E be an element of m edges. Let $n_i, \{i : 1, m\}$ be the number of green points on each edge. The refinement of an element requires:

1. to obtain the refinement point, and
2. to update the information of the grid with the new points, edges and faces generated by the refinement.

The step (1) needs to obtain all the points already inserted on the element edges. Therefore this process is $O(n_i)$ for the edge e_i. The total cost for the element is $O(n)$. Once a refinement point is chosen (among the current green points or —in case of fitting the object geometry— among geometry or intersection points), new green points may be inserted (2). The fact that each edge only stores one green point and there is an implicit binary search tree among an edge and its descendants allows – on average – to search for one point (or to insert it) in $O(\ln(n_i))$ on the edge e_i (see algorithm in Fig. 9). The worst case is $O(n)$. The worst case rarely occurs because of the way the refinement point is chosen.

The total cost of a refinement is $O(n)$ because of the step (1). In order to chose the best partition point it is necessary to obtain all the green points of the element.

Algorithms and data structures required for an BBA implementation are much more simple than for an IBA because every refinement is always done at edge midpoints. If an edge has already a green point it means that this bisects the edge. A new point is only inserted if the current edge has not been divided yet. Depending on how it was implemented the refinement can be done in time close to constant.

7 Conclusions

Algorithms and data structures for an IBA implementation on mixed element trees are still under research. It is still not possible to compare the ones presented in this paper with others inside of the same context because this is the first implementation. But their importance can be seen in the fact that an IBA can be implemented and successfully used for the generation of grids of more complex geometries, using less grid points and —for complicated geometries— less CPU-time than a BBA. For example, the fitting of the geometry using an IBA is four times faster than a BBA and the number of points is strongly reduced. (Fig. 3). Currently, a set of empirical results are being obtained for the whole mesh generation process.

Acknowledgments

This work was partially supported by FONDECYT project No. 1940323 in 1994.

References

[1] N. Hitschfeld and W. Fichtner, "3-D Grid Generator for Semiconductor Devices using a fully flexible Refinement Approach," in *Int. Conf. on Semiconductor Devices and Processes, pub. in Simulation of Semiconductor Devices and Processes*, vol. 5, pp. 413–416, Springer-Verlag, 1993.

[2] N. Hitschfeld, S. Müller, and W. Fichtner, "Generation of 3-d Delaunay Meshes for Complex Geometries using Iterative Refinement," *Ifip Transactions. Algorithms, Software, Architecture. Information Processing 92.*, vol. I, pp. 388–394, 1992.

[3] N. Hitschfeld, P. Conti, and W. Fichtner, "Mixed Elements Trees: A Generalization of Modified Octrees for the Generation of Meshes for the Simulation of Complex 3-D Semiconductor Devices," *IEEE Trans. on CAD/ICAS*, vol. 12, pp. 1714–1725, November 1993.

[4] M. Bern and D. Eppstein, *Mesh Generation and Optimal Triangulation.* Palo Alto Research Center. Xerox, March 1992.

[5] M.-C. Rivara, "Local Modification of Meshes for Adaptive and/or Multigrid Finite-Element Methods," *Journal of Computational and Applied Mathematics*, vol. 36, pp. 79–89, 1991.

[6] M. S. Shephard and M. K. Georges, "Automatic Three Dimesional Generation by the Finite Octree Technique," in *International Journal for Numerical Methods in Engineering*, vol. 32, pp. 709–749, 1991.

[7] R. E. Bank, D. J. Rose, and W. Fichtner, "Numerical methods for semiconductor device simulation," *IEEE Trans. on El. Dev.*, vol. ED-30, no. 9, pp. 1031–1041, 1983.

[8] M. A. Yerry and S. Shephard, "Automatic Three-dimensional Mesh Generation by the Modified-Octree Technique," *Int. J. Numer. Methods Eng.*, vol. 20, pp. 1965–1990, 1984.

[9] N. Hitschfeld, *Grid Generation for Three-dimensional Non-Rectangular Semiconductor Devices.* PhD thesis, ETH Zürich, Series in Microelectronics, Vol. 21, 1993. PhD thesis published by Hartung-Gorre Verlag, Konstanz, Germany.

[10] N. Hitschfeld, "Generación de Particiones que satisfacen la Condición de Delaunay para Poliedros Elementales 1-irregular," in *Por aparecer en Actas de VI Encuentros de Geometría Computacional*, (Barcelona, España), 5-7 Julio 1995.