SIMULATION OF SEMICONDUCTOR DEVICES AND PROCESSES Vol. 3
Edited by G. Baccarani, M. Rudan - Bologna (Italy) September 26-28, 1988 - Tecnoprint

Automatic Grid Generation for 3d Device Simulation*

Paolo Conti[†] Wolfgang Fichtner[†]

Integrated Systems Lab, ETH Zürich, Switzerland

Abstract

The concepts of an automatic 3d grid generator for the solution of the semiconductor equations are presented. The grid generator allows to mix elements of different shapes. An algorithm for the refinement of a grid according to a given point density function is outlined. Local heuristics for the improvement of the quality of the elements with worst aspect ratio obtained during the refinement are discussed. The data structure used to represent the grids is a boundary representation optimized to retrieve the neighborhood informations used in finite element assembly. Points and elements can be added and deleted in constant time.

1 Introduction

Silicon devices are inherently three-dimensional (3d) structures. While for many problems the behaviour of devices can be modeled in either one or two dimensions, for MOS devices with aspect ratios close to one, complex latchup structures or DRAM cells 3d simulation becomes necessary. The numerical solution of the semiconductor equations has been recently studied by various authors. They chose to solve the equations either with the Finite Differences (FD) method [1,2,3,4] or with the Finite Elements (FE) method restricted to prismatic elements [5]. None of these authors did worry about 3d grid generation, since FD operate on a rectangular grid while prisms for the FE method can be generated through a triangulation of the plane and a replication of the triangulation in the third dimension. The drawback of these methods resides in the serious limitations both methods impose on the device geometry.

In contrast to the above approaches, the device simulator under development at our laboratory allows the simulation of arbitrary 3d structures. In this paper, we describe ideas underlying the grid generator for this simulator. The grid generator must tessellate any 3d domain according to a given point density function. The tessellation process must be fully automatic, since 3d grids are far too complicated objects to be modified manually.

*this work was supported by Siemens AG, Berlin und München

[†]address of the authors: Institut für Integrierte Systeme, ETH-Zentrum, 8092 Zürich, Switzerland

The paper is organized as follows: In section 2, we describe the particular problems of the grid generation due to the peculiarities of the device equations. We summarize the grid qualities required for successful simulations.

In section 3, we show how well-known techniques for the local refinement of 2d triangular grids can be generalized to the 3d case. Then we describe the grid refinement algorithm implemented in our code.

Section 4 addresses the impact of the chosen data structure on our grid generator. In particular, we show how it allows to mix different element types and what properties make it suited for the exploration of new grid generation algorithms.

In section 5, we shall discuss the current implementation of our grid generator.

2 Requirements to FE grids for 3d device simulation

The correct allocation of the spatial grid is a crucial issue in device simulation. This is particularly true for 3d simulations. Since the different parts of a device have very different electrical behavior, the point density must be very high in some regions, while a much coarser grid suffices in the remaining part of the device. On the other hand, the simulation time depends directly on the number of grid points; since 3d problems lie on the edge of todays supercomputers capacity, great care must be taken to avoid the insertion of redundant points.

For a successful simulation, the generated elements must fulfil several requirements: the aspect ratio of the elements, i.e. the relation between the lengths of the longest and shortest edge, should not be greater than a given upper bound (typically 4 to 10). As a consequence, we must pass from very dense regions to coarser regions in a smooth way, avoiding steep point density gradients. Moreover, obtuse angles should be avoided as much as possible, since they cause two kinds of problems: They exacerbate any inherent roughness in the solution, and they can inhibit the convergence of the solution scheme. While recently proposed discretization techniques [6] permit to avoid the latter problem, the problem of the inherent roughness still requires to avoid very obtuse angles as much as possible.

The choice of element geometrics best suited for device simulation is still an open question. While tetrahedra allow to model very general geometries, they inherently cause an inflation in the number of generated elements. Experience will show whether this decreases the efficiency of the simulation process, since the number of linear equations to be solved depends only on the number of points of the tessellation, and not on the number of elements. On the other hand, tessellating a large area of the integration domain with bricks, using tetrahedra and prisms only to model the irregular boundary regions and the regions where we pass from high to low point densities, may accelerate both the grid generation and the assembly of the linear system to be solved.

3 Local refinement of 3d tetrahedral grids

Given a point density function defined over an integration domain, there are two basically different approaches to generate a corresponding grid. One method first defines the location of the points in the domain; then the elements connecting these points are constructed. A good candidate tessellation is the Delaunay triangulation of the given points [7], since it guarantees positive effective cross sections of the mesh lines, as required by the Scharfetter-Gummel method. Unfortunately, the Delaunay partitioning of the device to be simulated has serious drawbacks: the computation of the partitioning is expensive for a large number of points (we estimate the number of points necessary for real-world simulations to be of the order of 50000); moreover, the Delaunay partitioning may generate elements ranging over regions of different materials, a new source of errors in the simulation.

With the second method, a coarse grid is first generated according to the device geometry. Then, all elements found too coarse according to the given point density function are refined via the addition of new points on their edges. Finally, all elements on which new nodes (so-called green nodes) where added during the previous step are refined. The process is iterated until all elements fulfil the given density condition or until a certain iteration depth is reached. Since this approach has been used successfully in several 2d FE codes [8,9], we have generalized it to tetrahedral elements for our prototype implementation.

In 2d, coarse triangles are refined via the addition of a new point at the midpoint of each edge. A triangle is divided in 4 similar triangles, thus the quality of the elements does not deteriorate at this step. In 3d, we add an additional point on each edge of a coarse tetrahedron. Then we chop off the four tip tetrahedra of the original element (Figure 1). These four "tips" are

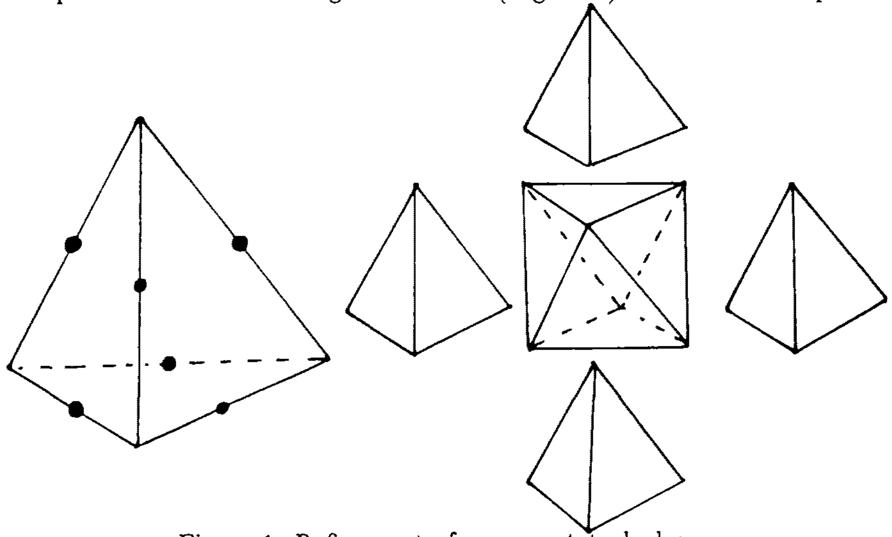

Figure 1: Refinement of a coarse tetrahedron .

similar to the original element. Finally, the remaining octahedron is divided in four tetrahedra via the insertion of an additional edge (one of the three diagonals of the octahedron). As the resulting tetrahedra are not similar to the original element, this step leads to an unavoidable quality decrease of the tessellation. In order to keep this deterioration small, we add the diagonal with length closest to the average of the edge lengths of the octahedron.

The triangulation of the neighbors of refined elements is significantly more complicated in 3d than in 2d. In 2d, a triangle with an additional point on one, two or three of its edges can easily be subdivided in triangles. But in 3d, in addition to the points which where added in the middle of some edges, new edges may have been added on the surface of the tetrahedron during the refinement of its neighbors. In general, the resulting tetrahedron cannot be triangulated without the addition of new points. As an example, consider the tetrahedron T of Figure 2. The tetrahedron sharing the right face of T has been

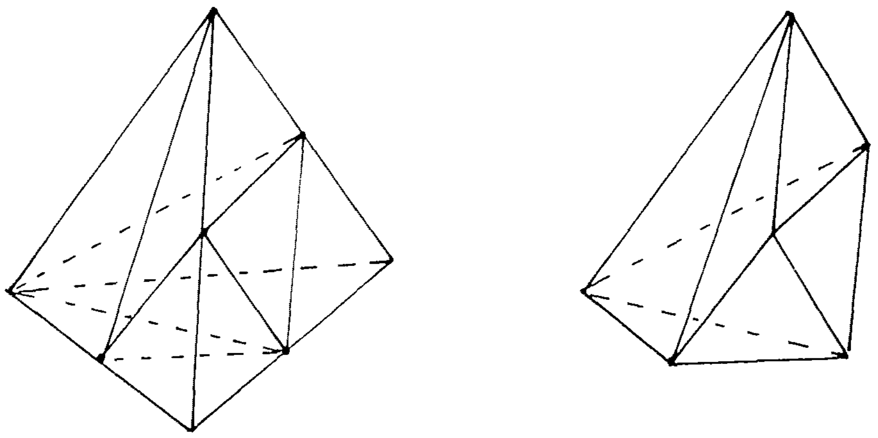

Figure 2: Refinement of a tetrahedron with green nodes

properly refined; hence, three new points and three new edges where added on the right face of T. Moreover, one tetrahedron incident in the lower left edge has been refined; an additional point was added this way on this edge of T. The tetrahedra incident in the three other faces of T have already been refined, taking into account these new points. Five new edges were added on the surface of T during this process. As a result, T cannot be triangulated without the addition of new points; we can chop off the tip tetrahedra in the front and right corner of T, but the remaining octahedron (Figure 2) cannot be triangulated.

In our code, we triangulate tetrahedra with green nodes as follows: First the hull is triangulated. Then, all tip tetrahedra are chopped off. Finally, an additional point is added in the center of the remaining convex polyhedron, and additional edges and faces are added to join this new vertex with each of the vertices and edges of the polyhedron. Any tetrahedron with green elements can be triangulated that way. In general, the resulting tetrahedra have a worse aspect ratio than the original.

We can summarize our triangulation algorithm as follows:

```
Generate initial triangulation;
Collect coarse tetrahedra;
repeat
    Divide coarse tetrahedra by halving their edges;
    Refine tetrahedra with green knots, adding a knot in
      the center if necessary;
    Collect coarse tetrahedra among new ones;
until All elements are fine enough;
```

4 Enhancement of the grid quality

During the refinement process of a triangular grid, new elements with bad aspect ratios are generated, in particular in the regions where the point density varies. In 2d, point smoothing and edge smoothing techniques have been successfully applied to enhance the overall quality of grids [8]. In the first case, points are moved around a little, without modifying the topology of the triangulation. Given a badly shaped triangle, one (or more) of its vertices is moved in order to increase the quality of this triangle. A common approach is Gaussian smoothing, where a point is moved to the geometrical center of all its topological neighbors. Figure 3 illustrates the effect of point-smoothing in the 2d case.

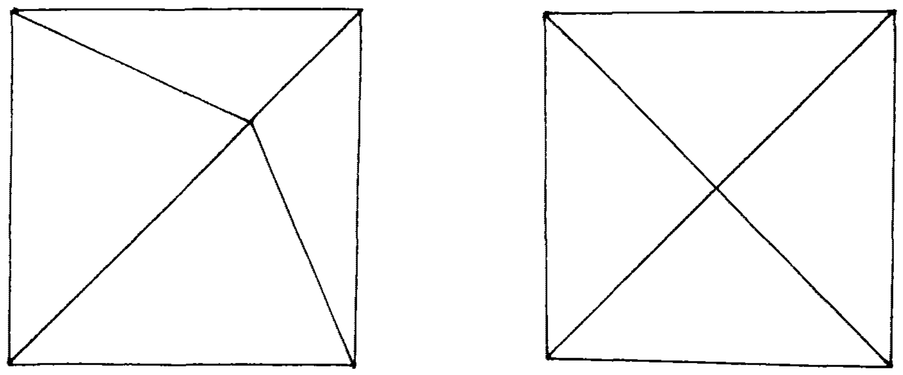

Figure 3: Point smoothing in 2d

The method can be generalized directly to the 3d case. It must be applied very carefully, since it tends to regularize the grid and thus to move the points away from where they are actually needed. In general, one or two iterations over all bad elements can be performed.

During edge smoothing, no point is moved; only the topology of the grid is modified. In 2d, edges are swapped if the sum of opposite angles in a pair of triangles exceeds 180°. By flipping the diagonal, the sum of opposite angles is made less than 180°, thus insuring better convergence of the Scharfetter-Gummel discretization. Edge-smoothing can partially be generalized to 3d. Consider the two tetrahedra in Figure 4: they can be replaced by three tetrahedra with better shape, adding the edge which connects the top and bottom vertices; the common face of the two tetrahedra is replaced by three new faces incident in the new edge. The operation is reversible.

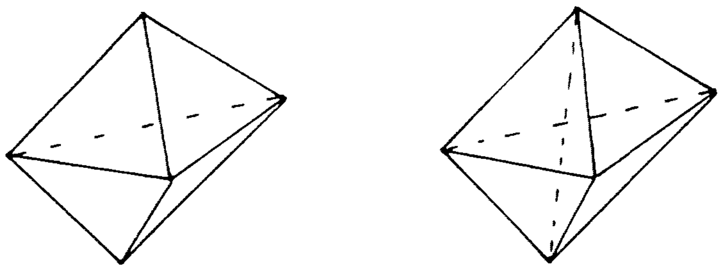

Figure 4: Edge smoothing in 3d

Notice that two tetrahedra can be traded for three new ones only if the new edge lies within the their common face. Figure 5 shows an example where two badly shaped elements cannot be replaced by three better ones without modifying the hull of the two elements.

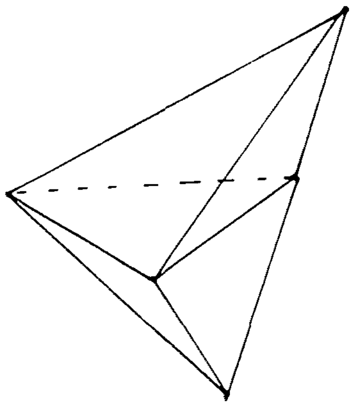

Figure 5: No edge smoothing can be performed

We do not know yet the exact general properties of a 3d triangulation after edge smoothing has been consequently applied. It is nonetheless obvious that the method allows to improve the quality of a 3d triangulation in linear time.

Therefore we plan to integrate it in our code shortly and to measure its impact on the convergence behaviour in simulations of different structures.

5 Data structure

The data structure used to store the grid during the generation and refinement process must suffice to different constraints. As stated above, we plan to use mixed element types (in particular tetrahedra, prisms and parallelepipeds) in our simulator. Moreover, since the grid must be modified incrementally, the insertion and deletion of new points, edges etc. must be doable in constant time, i.e. without a sweep over the entire data structure.

We have chosen to base our grid generator on the Hexblock data structure [10]. Hexblocks permit to represent any tessellation of a 3d domain by its boundary. Thus different element types can be mixed easily. The addition and deletion of new points necessitates only local modifications in the data structure. Moreover, all kind of neighbor information (e.g. the list of elements incident in a given edge, the list of neighbor elements of a given element etc.) can be retrieved in constant time; thus, the data structure is adapted for the experimentation of different grid refinement strategies.

The most serious drawback of the Hexblock data structure is its memory greed. The price to pay for the availability of all neighbor information in constant time is a high redundancy in the data structure. While we are convinced that Hexblocks are a good choice for the development of grid refinement algorithms, we do not know yet if the performance will be satisfactory for repeated simulations in a production environment.

6 Implementation

The grid generator is implemented in the object-oriented programming language C++ [11]. The choice of C++ has proven to be a good choice: the powerful data hiding mechanisms of the language permit to hide many details of the Hexblock data structure in lower modules. Obviously clean and structured programming is possible with any programming language. Nontheless our experience shows that the use of a language with powerful data hiding mechanisms makes things a lot easier when dealing with complicated geometries and data structures.

Consider as an example the program segment below. The current partition is accessed via a pointer encapsulated in an object of type `Tessellation`. The partition is accessed and modified only through the member functions of this class.

Member functions, e.g. `Neighbours()` always apply to the element of the tessellation currently pointed to. In the last few lines of our example, we collect all neighbors of an element and refine all of these neighbors which contain green nodes. Notice how concisely such a complex operation is described in C++.

The efficiency of the grid generator is not satisfying yet. The generation of a locally refined grid for a 3d abrupt MOS diode with 5735 points and 30030 tetrahedra took 7 minutes CPU time on a SUN 3/260. Since all used algorithms

```
  /* declaration of a class, including the functions
     which access the data fields */

  class Tessellation {
   private:
     Pointer_to_hexblock   ptr;

   public:
     void          Add_vertex();
     void          Refine_element();
     void          Refine_green_element();
     int           Is_green_element();
     List_of_tess  Neighbours();
                  .
                  .
                  .
}

/* declaration of variables */

Tesellation   current_element;
List_of_tess  neighbours;

/* collect all neighbors of the current element and refine
   the elements with green nodes on their surface */

neighbours = current_element.Neighbours();
for (current_element in neighbours) {
   if (current_element.Is_green_element()) {
      current_element.Refine_green_element();
   }
}
```

are linear in time, linear extrapolation of these performances gives a rough estimate of the cpu times needed for bigger examples. However, it must be noted that when the memory needed to store the grid becomes much (e.g. 4 times) larger than the real memory allocated to the process, the performance decreases significantly due to page thrashing.

7 Conclusions

We have presented a first version of an automatic grid generator for 3d device simulation. An algorithm for the local refinement of a triangular grid according to a given point density function has been presented; the issue of how the quality of an existing triangular grid can be improved has been addressed. The presented implementation is based on the Hexblock data structure. Hexblocks have proven to be a very flexible basis to explore new algorithms, while the question whether the code based on Hexblocks will be efficient enough for real life examples is still open.

References

[1] W. Fichtner, R. L. Johnston, and D. J. Rose, "," in *Proc. 1981 Device Research Conf.*, 1981.

[2] A. Yoshii *et al.*, "A three-dimensional analysis of semiconductor devices," *IEEE Trans. on Electron Devices*, vol. ED-29, pp. 184–190, 1982.

[3] T. Toyabe, Y. Ohkura, and H. Masuda, "Methods of three dimensional transient simulation and their applications to VLSI reliability problems," in *Proc. of NASECODE V Conf.*, pp. 74–84, 1987.

[4] M. Thurner and S. Selberherr, "The extension of MINIMOS to a three dimensional simulation program," in *Proc. of NASECODE V Conf.*, pp. 327–332, 1987.

[5] E. M. Buturla *et al.*, "," in *IEEE Solid-State Circuits Conf. Dig. Tech. Papers*, p. 76, 1980.

[6] J. F. Bürgler, R. E. Bank, W. Fichtner, and R. K. Smith, "A new discretization scheme for the semiconductor current continuity equations,". Submitted to IEEE Trans. on CAD.

[7] F. P. Preparata and M. I. Shamos, *Computational Geometry — An Introduction.* New York: Springer, 1985.

[8] M. R. Pinto, C. S. Rafferty, and R. W. Dutton, "PISCES-II—Poisson and continuity equation solver," Tech. Rep., Stanford Electronics Lab., 1984.

[9] R. E. Bank, "Pltmg users' guide–edition 5.0," Tech. Rep., Dept. of Mathematics, University of California at San Diego, 1988.

[10] C. E. Buckley, "The hexblock modelling system,". To be published.

[11] B. Stroustrup, *The C++ Programming Language.* Reading: Addison-Wesley, 1986.