GRID AND GEOMETRY SERVERS FOR SEMICONDUCTOR PROCESS SIMULATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Zakir Hussain Sahul
July 1996

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Robert W. Dutton
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Kincho Law

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Peter Pinsky

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Krishna Saraswat

Approved for the University Committee on Graduate Studies:

Abstract

Semiconductor process simulation involves the numerical solution of equations describing the physics of dopant diffusion, silicon oxidation, lithography, ion implantation, etching and deposition steps, resulting in geometry and doping profiles that define a device. A number of programs, each specialized to solve a specific set of equations, are used to simulate the entire process flow. However, the various programs use different solution strategies and semiconductor wafer state representations.

Three different wafer state representations are considered: 1) geometry, 2) surface mesh and 3) volume grid. A geometric boundary representation with supporting algorithms for validation and repair is used for topography simulations. Boolean etching and deposition operations allow rapid device cross section prototyping. Surface mesh representation together with mesh generation and movement capabilities are used for physically based topography simulations. Automatic detection and removal of boundary loop and region crash anomalies enables multi-layer simulations.

A grid representation with supporting automatic grid generation, adaptation, and movement functions is used for thermal simulation. Grid generation is performed using a tree based approach. Adaptation is performed by pruning or growing the tree based on an error indicator. Grid movement is performed by changing the geometry and then regenerating the grid.

The three wafer state representations are kept consistent allowing heterogeneous tool integration. The process simulation programs SUPREM-IV and SPEEDIE, together with new Boolean topography applications and an error estimator application are seamlessly integrated into a single process simulation environment. A minimal server procedural interface is utilized to access the algorithms and data associated

with the wafer state representations, thereby allowing easy tool integration with few code changes.

Acknowledgments

No Ph.D dissertation is completed without the help, advice and support from a number of people. My advisor, Dr. Robert W. Dutton, has provided a considerable amount of each of the above for which I am extremely grateful. Moreover, as part of his group, I have learned many good lessons about life that I hope will serve me well in the future.

I would like to thank the other members of my dissertation committee - Dr. Kincho Law, Dr. Krishna Saraswat and Dr. Peter Pinsky for all their help. I would also like to thank Dr. Hector Garcia-Molina for chairing my oral defense.

Many interactions with people in industry have given me a practical perspective on research. Dr. Jue-Hsien Chern[1] of Texas Instruments was instrumental in helping me get started on the research presented in this dissertation. Matt Noell of Motorola helped test an early version of my code and made many suggestions on improvements.

Of the many people I have collaborated with, I would like to especially thank Gene McKenna and Kai Hsiau. Gene helped interface my code with SUPREM-IV while Kai helped interface the code with SPEEDIE.

I would also like to thank Dr. Jed Black and Dr. Christian Guilleminault of the Stanford University School of Medicine for giving me an opportunity to experience biomedical research and for giving me a paid month off to finish writing this dissertation.

Finally, I would like to thank my parents, Dr. M. Sahul Hamid and Mrs. Meharbaan Sahul Hamid. They have provided unwavering support through thick and thin all my life. This dissertation is a testament to the energy and time that they devoted to raising my sister and me.

[1]Now of TMA Inc.

Many people have made the halcyon graduate student days at Stanford more than just a time of studies and I am very fortunate to have met them. Jean Yang will always be special to me. Victor Kreiman, Robert and Kris Huang, and Maria Perea have been very close friends who have helped me enjoy the good times and weather the not so good times. Fely Barrera, who help has been invaluable, and Alan "Pangit" Massengale, who was the instigator behind many a practical joke, have also been great friends.

Bruce Herndon (who fought the cat people and lived to finish up before me), Richard Williams, Francis Rotella, Goodwin Chin (who answered many questions), Chiang-Sheng Yao, Dan Yergeau, Todd Atkins, Chris Quinn and many other members of the "TCADre" made life in AEL memorable. The summer softball games and the regular lunch outings made the years pass by very quickly.

Contents

List of Figures

Chapter 1

Introduction

Computer-Aided-Design (CAD) has been an invaluable tool in the research and development of new technology for several decades. The integrated circuit (IC) industry uses a number of computer programs to simulate the manufacture and operation of an IC at different levels. At a high level, devices are abstracted as switches or gates that implement a logic function. Logic simulation using VERILOG [10] is used for timing and logic analysis. At a very low level, statistical simulations using Monte Carlo simulators like DAMOCLES [19] compute electron or dopant atom densities. Technology CAD (TCAD) is at the level of semiconductor process and device design. The numerical solution of equations that describe the physics of dopant diffusion, silicon oxidation, etching, deposition, lithography, implantation and metalization is the realm of semiconductor process simulation. Solution of the electron, hole and energy transport equations in a device under given applied voltages and currents is the realm of device simulation.

Full semiconductor process simulation requires the use of several different CAD programs. SUPREM-IV [30] and FLOOPS [31] are used to simulate diffusion and oxidation. SPEEDIE [33] and SAMPLE [36] are used to simulate etching and deposition. Simple geometric Boolean operations are also used for non-physical "simulation" of etching and deposition steps. The final result of a process simulation is a wafer state that defines electrical devices.

This dissertation discusses a set of *geometry*, *surface mesh* and *grid* algorithms, implemented in the software package called *Forest*, that supports full process simulation. This chapter introduces semiconductor processing and process simulation. The concepts of geometry, surface mesh and grid are discussed in relation to their relevance to TCAD. An outline of the dissertation is also presented.

1.1 Semiconductor Processing and Process Simulation

Processing transforms a semiconductor wafer into a working device as shown schematically for a simple MOS process in Figure 1.1. While actual industrial processes include many more steps, the basic principles illustrated in the figure are equally applicable. Silicon dioxide and silicon nitride layers are first deposited on a doped silicon wafer. The nitride layer is then selectively etched as shown in Figure 1.1b. Exposing the wafer to an oxygen ambient then produces a silicon dioxide region, called a field oxide, as shown in Figure 1.1c. The field oxide serves as an isolation structure preventing parasitic transistor action between neighboring devices. The deposited oxide and nitride layers are stripped and a new gate oxide is grown. A polysilicon gate[1] is then deposited and patterned as shown in Figure 1.1d. Ion implantation then introduces dopant atoms in the device as shown in Figure 1.1e. A second implant is required to form a lightly doped drain (LDD) structure. The second implant is aligned using an oxide spacer layer. An oxide layer is first deposited and then anisotropically etched as shown in Figure 1.1f and Figure 1.1g. The implants are then driven-in by a high temperature anneal. The dopant atoms diffuse through the lattice and directly determine the junction depth (Figure 1.1h). A doped viscous glass layer is then deposited, patterned and etched resulting in the structure shown in Figure 1.1i. Finally, deposition and patterning of a metal layer results in a final working device as shown in Figure 1.1j.

Processing changes the wafer geometry - material is selectively etched from some

[1]Industrial processes include an implant step before the poly gate is patterned for adjusting the threshold voltage.

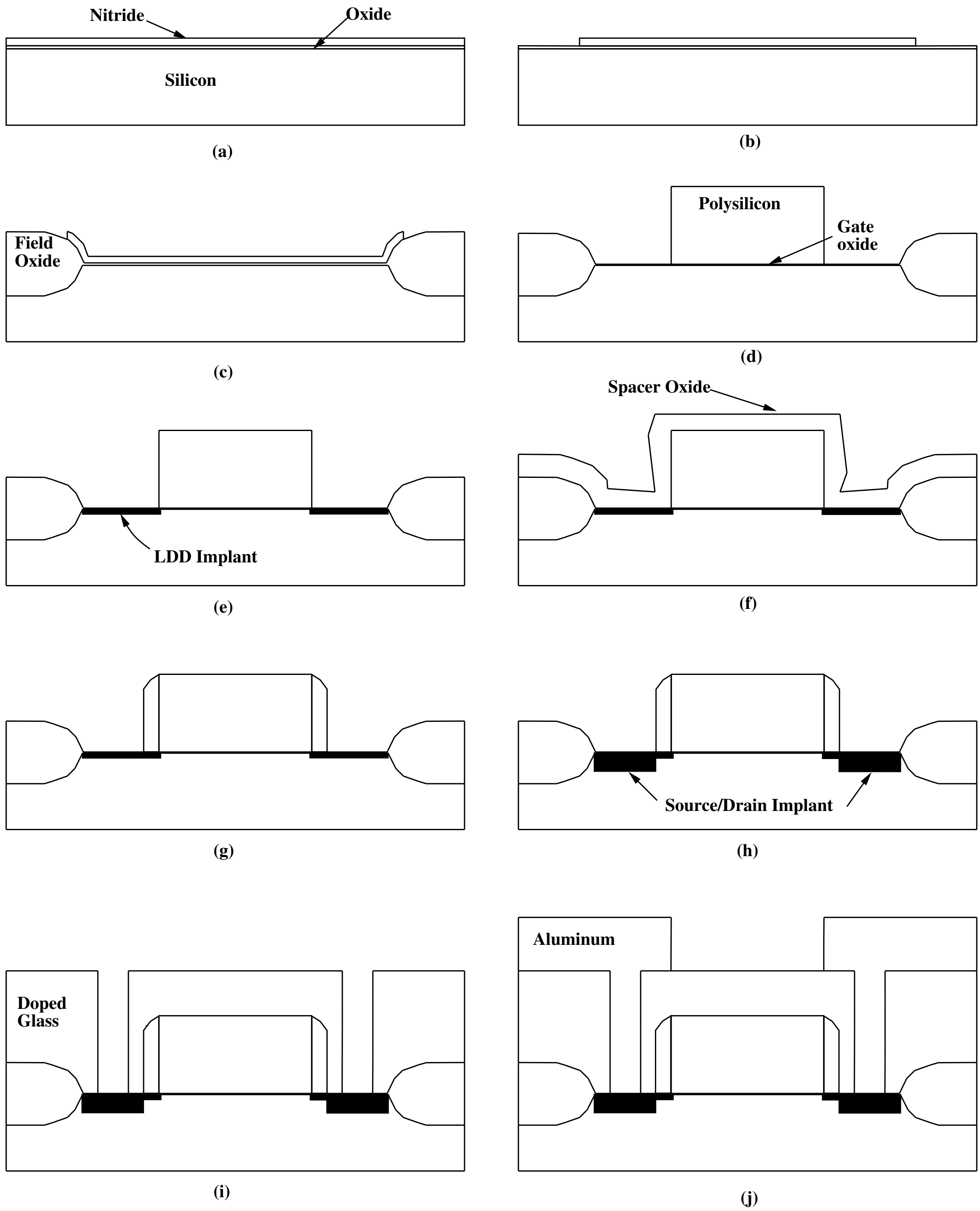


Figure 1.1: Schematic representation of transforming a semiconductor wafer into a working MOS device. Please see text for explanation of each step.

regions, different materials are deposited and the wafer surface itself is reacted to form new materials. Moreover, in the device interior, different dopants are introduced into the semiconductor crystal lattice changing local electrical properties. In order to simulate the entire process flow, each of the unit processing steps - deposition, etching, oxidation, ion implantation, and dopant drive-in - is mathematically modeled and the governing equations are solved numerically.

Etching and deposition are simulated by solving for the flux of the reacting species across the exposed wafer surface using surface integral methods. This determines the etch or deposition rate [36]. Simulation of ion implantation is performed using Monte Carlo techniques to solve an approximation to the Boltzmann Transport Equation. Simpler techniques using analytic functions are often used as good approximations. Dopant drive-in is modeled by solving diffusion equations derived from Fick's law using finite element [26] or finite volume [30] methods. Simulation of oxidation also involves the solution of a diffusion equation for the diffusion of the reacting species across any existing oxide layer. Flux of the reacting species across the reacting surface and flow of the newly formed oxide determines the actual oxidation rate. The oxide flow is modeled by another transport equation.

1.2 Geometry, Surface Mesh and Grid

As discussed above, each process step changes the semiconductor wafer state. The wafer state can be defined in terms of a geometry, a surface mesh and a grid representation as illustrated in Figure 1.2. Geometry refers to the gross outlines of the different materials in a structure. Surface mesh refers to a collection of planar elements whose union forms the geometry. A grid refers to a collection of volumetric elements whose union defines the interior and exterior of the device.

A geometry representation is used by programs that are concerned with only the extent of different material regions. As materials are deposited and etched, the geometry is manipulated to incorporate the changes. Many of the patterning steps illustrated in Figure 1.1 can indeed be performed by such "Boolean" geometry programs. However, for simulating the formation of the spacer layer shown in Figure 1.1g, greater

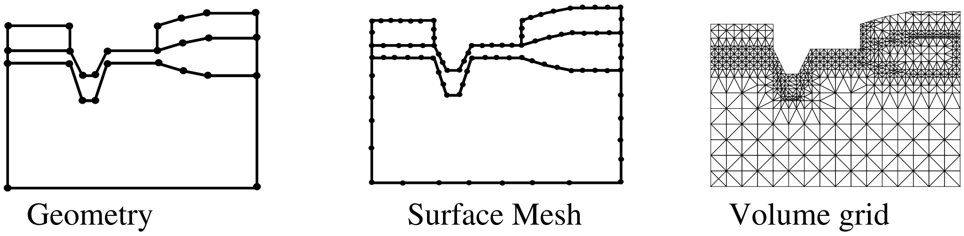


Figure 1.2: Illustration of geometry, surface mesh and grid representations in 2D.

accuracy is desired and a physically based based etch and deposition simulation is performed. The gross outlines represented by the geometry are discretized to form a surface mesh. The large number of small surface mesh elements allows finer geometrical resolution resulting in a more accurate simulation. Thus, for topography simulation, geometry and surface mesh representations together with supporting algorithms are needed.

Dopant atoms, introduced into the crystal by implantation steps and then driven-in with an anneal step, reside in the continuous interior of the device. Since it is impossible to solve for quantities at every spatial location within a continuum, doping concentrations are defined on a set of "grid points" that represent the continuum. The governing partial differential equations are also transformed into a set of simultaneous algebraic equations defined over the grid points. Finite element and finite volume solution strategies, used for solving the differential equations, also require partitioning the device volume into a set of elements whose vertices coincide with the grid points. Thus, a grid representation together with supporting algorithms is required for implantation and diffusion equations.

The work presented in this dissertation considers the question of *how* to efficiently represent and manipulate each of the three representations of the wafer state. Algorithms for geometry manipulation, grid and surface mesh generation, grid adaptation, and grid movement are presented and analyzed.

1.3 Dissertation Outline

Geometry representation and algorithms needed to support simple topography simulation are discussed in Chapter 2. Rapid prototyping of a DRAM process is illustrated with Boolean operations. The issue of surface motion and its use in physically based topography simulation is discussed next in Chapter 3. Example simulations with SPEEDIE are presented.

Grid requirements for process simulation and a tree based grid generation algorithm are presented in Chapter 4. The chapter also discusses grid adaptation for ion implantation and dopant diffusion simulations. Moving boundary problems present the most difficulty in terms of grid and geometry services. Solutions are computed on a grid but the solutions represent material movement and geometrical change. Algorithms to support the use of geometry, and grid consistently for boundary motion are also discussed.

Tool integration is discussed in Chapter 5 as many different programs are often used to simulate the entire process flow. Some programs use only a grid representation while others use only a geometry or surface mesh representation. A new methodology that enables these different programs to seamlessly change and produce a final semiconductor wafer state is presented. Appendix A contains a proposed minimal server application procedural interface (API) that facilitates tool integration. Finally, Chapter 6 contains conclusions and recommendations stemming from the work presented in this dissertation.

Chapter 2

Geometry Representation and Manipulation

2.1 Introduction

As discussed in Chapter 1, several semiconductor processing steps change the wafer geometry. Materials are deposited over the surface, selected areas are etched, and surface reactions with ambient molecules produce new materials like silicon dioxide and silicon nitride [28]. A process TCAD program must, therefore, simulate geometrical change. Hence, it must represent the device with appropriate data structures and provide algorithms to change the geometry. Geometry representation and geometrical algorithms as implemented in the program *Forest* are the subject of this chapter.

One of the most popular data structures for representing a geometry, the wire frame boundary representation model, is discussed in detail– comparing its benefits and drawbacks. Algorithms to perform non-physical geometrical etching and deposition simulations in 2D are presented. Such capability allows a designer to prototype a device cross section rapidly without expensive simulations. Several geometry utility algorithms to support geometrical etching and deposition share the common needs for region intersection, geometry validation and geometry repair operations. Discussion of geometry movement that arises in physically based simulation of etching and deposition, including supporting algorithms, is deferred until Chapter 3. These algorithms and data structures comprise a general purpose "geometry server" that can be used by a variety of process simulation tools.

2.2 Geometry Representation

Several data structures have been used to represent a geometry. The two most popular in the TCAD community are the boundary representation model and a new "boundary free level set" model. These schemes represent different compromises in memory usage, programming ease, and ensuing algorithm robustness. The boundary representation model, as implemented in the *Forest* program, is described in this section. The boundary free representation is also discussed briefly.

2.2.1 Boundary Representation Model

The most commonly used and intuitively simple representation of the geometry is a wire frame representation of the different materials in a simulation domain as shown in Figure 2.1. The geometry is stored as an hierarchical collection of points, edges, boundaries and regions. Points are spatial locations; edges are connections between two points; boundaries are a collection of edges that form a closed loop. Regions are a collection of boundaries, forming a material outline in 2D. In 3D, surfaces are collections of boundaries that form one face of the material outline and volumes are collections of surfaces to form a fully closed material outline. The hierarchy as implemented in *Forest* is maintained by using pointers. Edges hold pointers to each of their end points. Boundaries hold a set of pointers to edges that form a closed loop. Regions hold pointers to several boundary objects, etc. While no other assumptions are needed to represent any arbitrary geometry, two orientation schemes are enforced in *Forest* to reduce algorithm complexity. Edges are defined as vectors that originate at a "starting point" and end at an "end point". Pointers to edges within each boundary are ordered such that the area is to the left as the edges are traversed from the first to the last; i.e., the points comprising a boundary are ordered in a counter-clockwise manner.

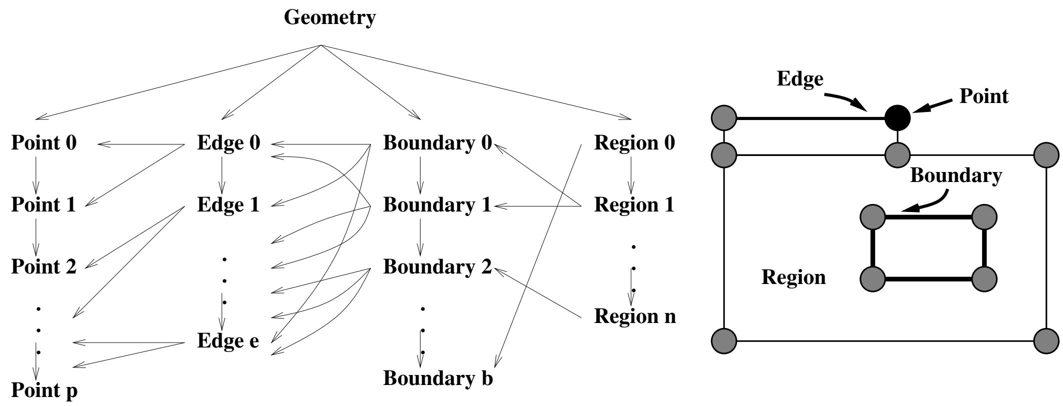


Figure 2.1: Data structures required to represent a geometry using the boundary representation model in 2D. The geometry is a collection of regions that are bounded by boundaries. Boundaries are closed loops consisting of edges. Edges are directed line segments defined by points.

None of the objects within the representation is duplicated. When a new object is added, the program searches through the existing object lists to find a matching object. A new object is added only if a matching object does not exist. Additionally, an attribution mechanism is also used for all objects in the geometry. Thus, boundary conditions can be easily specified on edges, doping values can be defined on regions, etc. This attribution mechanism is discussed in detail in Chapter 5. In the ensuing discussions in this and subsequent chapters, it is sufficient to note that arbitrary attributes can be attached to any geometrical object.

Since a hierarchical representation is used to store the geometry, the memory requirements of a boundary representation model can be substantial. Consider, for example, the representation of an area enclosed by a loop of N points. The hierarchy is described by the N points, N edges connecting the points, one boundary describing the loop and one region. The point objects would each require three floating point values for representing the three coordinates. The edge objects would each require two pointer values and the boundary object would require N pointers for the N edges and the region object would require one pointer for the boundary. Assuming no other information is stored at any of the objects and that floating point values use the same storage space as pointers, the memory requirement is thus $3 * N + 2 * N + N + 1$

storage units.

Without a hierarchy, if a closed loop of N points was ordered such that the N points were stored consecutively as the loop is traversed, then the ordering of the points alone is sufficient to describe the area. There is no need for separate edge, boundary or region objects. Taking into account the extra integers required for keeping track of how many points are in the loop, the storage requirement is only $3 * N + 1$. However, if there are several loops and several of the points are shared by the loops, then the duplication of points in the non-hierarchical scheme would add extra overhead.

Alternative hierarchical schemes can also be used to strike a different trade-off between lower memory use and increased computational efficiency with greater information storage. Some algorithms described in following sections, for example, would execute faster if a "double hierarchy" is established - points hold pointers to all edges that connect to them, edges hold pointers to all boundaries that contain them, etc. Such a top down and bottom up double hierarchy was not used in this work to save memory usage. Thus, parent query operations for the lower dimensional objects is not provided. For example, adjacency information between two boundaries is determined by checking to see if the boundaries hold pointers to the same edge.

If memory usage needs to be reduced further, an even more reduced hierarchy can be implemented. For example, the edge object can be eliminated with boundaries containing pointers to a set of points that form a loop. The boundary object may be eliminated if all regions are constrained to have only one boundary. However, edge objects are extremely useful for process simulation as they are the natural objects upon which one can specify boundary conditions for the governing equations. Specifying boundary conditions on points is awkward and may cause problems when the same point is shared by surface segments with different boundary conditions. The boundary object was used in this work to allow representation of regions with voids.

Returning to the discussion of the full hierarchical representation, it is readily seen that if the number of points describing the loop were doubled to $2N$ for greater resolution, the memory requirements would then linearly increase to be $3 * (2N) + 2 * (2N) + (2N) + 1$ storage units. Even in 3D, memory requirements grow linearly as illustrated in Figure 2.2. Representing a tetrahedra with four points requires 41

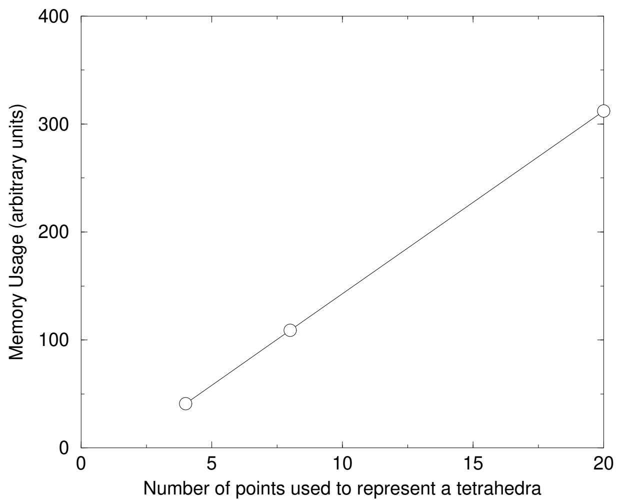


Figure 2.2: Memory requirement for the boundary representation model to represent a tetrahedra in 3D as a function of the number of points.

storage units. Representing the same tetrahedra with eight points requires 109 storage units while representing it with twenty points requires 312 storage units. However, one must be careful to distinguish between doubling the number of points in the geometry and doubling the "resolution." In 2D, doubling the number of points indeed doubles the resolution. In 3D, the number of points needs to be more than doubled or tripled to double the "resolution," depending on the primitive unit cell as illustrated in Figure 2.3.

The geometry hierarchy imposes strict constraints on adding or deleting objects. Consider, for example, the deletion of a point as the geometry is smoothed. The operation involves removing the two edges that end at the deleted point. A new edge should be created to replace the two edges. Boundaries that point to the two deleted edges must now be modified to point to the new edge instead. An insertion operation is similar. In this regard, the hierarchical representation imposes a large computational overhead as well.

The advantage of the boundary representation model, however, is in the ability to represent any arbitrary geometry. There are no constraints in the shape (convex,

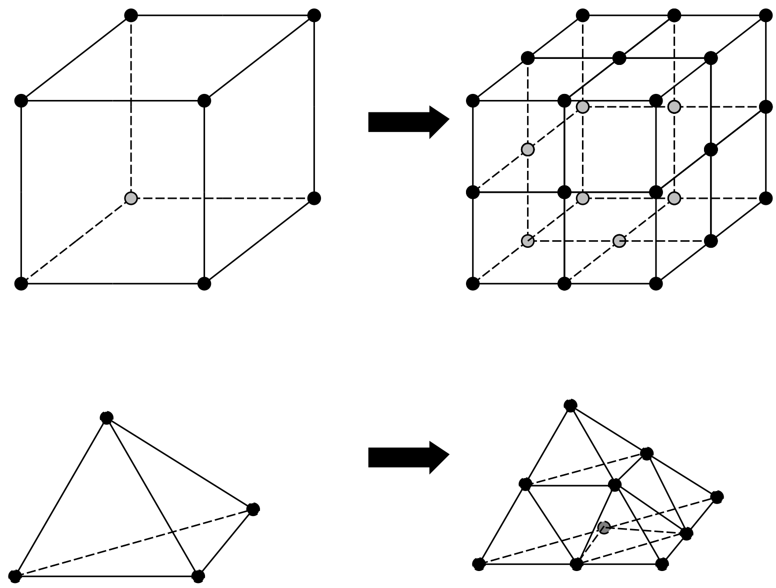


Figure 2.3: Doubling the resolution in a 3D geometry adds at least twice as many points depending on the primitive unit cell.

concave, simple, smooth, etc) of the geometrical objects. Moreover, the rigid consistency between the objects ensures a valid geometry at all times, particularly when the geometry moves. The coordinates of the points can be changed to simulate motion and since the hierarchy is still intact, the geometry is valid if the displacement is small. The small displacement stipulation is required so that different materials do not crash into each other or boundary loops do not form from geometry motion. Algorithms to recover from such anomalies are described in later chapters.

2.2.2 Boundary Free Level Set Representation

A recently proposed method for representing geometries is the "boundary free level set" representation [32]. The geometry of the device is represented as the distance $\psi(x, y)$ from each grid point on a regular grid to the nearest point on each of the material boundaries as illustrated in Figure 2.4. Grid points within the boundary are defined to have positive distance values while those outside the boundary are defined to have negative distance values. The contour, $\psi = 0$, thus defines the surface. The geometry is in essence a "solution" on grid points similar to dopant values. The

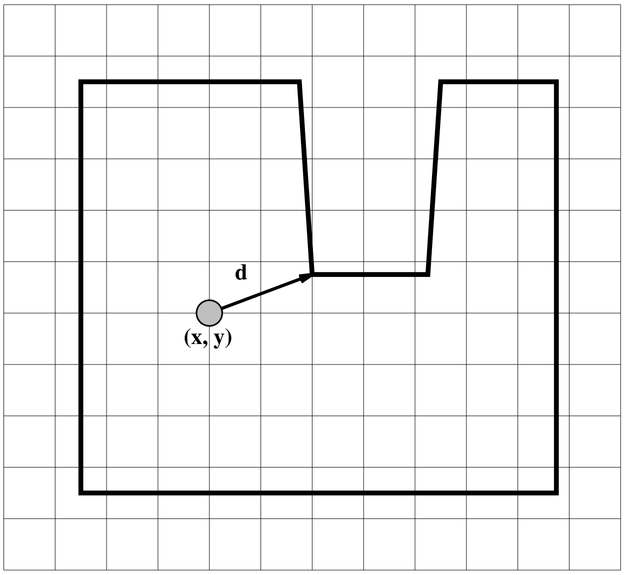


Figure 2.4: Illustration of the boundary free geometry representation model. The boundary is represented on grid points by a scalar parameter which is the shortest distance from each grid point to the boundary.

advantage of this representation stems from the fact that the constraint to boundary displacement is built into a differential equation. Moreover, since the geometry consists of values stored on a grid point, the values can be changed easily. Proper solution of the differential equation assures that the geometry is never made invalid, and thus, geometry repair algorithms are not needed. As robust geometry repair algorithms are extremely difficult to implement especially in 3D, the boundary-less representation is ideal for problems involving geometry motion. The disadvantage of this representation is in greater memory usage and reduced computational efficiency.

Considering the memory requirements, it is readily seen that a grid is needed to store a geometry. Compared to a wire frame representation, the number of points in a grid will be significantly larger. Moreover, each region boundary in the structure requires a floating point distance value to be stored at each of the grid points. Computationally, a displacement differential equation needs to be solved in order to move the boundary. For a wire frame representation, the solver computes velocities on the points of a boundary, and the points are simply moved by changing their coordinate values. For the boundary free representation, the solver still needs to compute the velocities on the boundary points. These boundary velocities are then used to compute the new distances to the boundary at each grid point by solving a differential equation.

Moreover, there is no straight forward manner in which a non-physical geometrical change can be affected without a velocity based displacement computation. Since the geometry is stored in a diffuse manner over grid points, it is difficult to specify an exact change to it. The only available option is to convert the representation into a wire frame representation, make the changes on the wire frame representation and then to convert back to the level-set approach. Thus, the level set approach trades off storage and computational efficiency for robust geometry movement algorithms.

2.3 Geometric Deposition

A wire frame geometry representation alone can be used to simulate the sculpting of the semiconductor wafer. In a TCAD environment, the process designer is often interested in using simple geometry operations for non-physical etching and deposition. For example, in order to pattern a photoresist layer, a blanket deposition (spin-on) is performed following by exposure and development. A full physically based simulation of the spin-on process or of resist development step will be computationally expensive and of marginal value in the overall process simulation. A simple non-physical deposition and etch "simulation" would suffice. Two modules within the *Forest* program provide this capability.

Pseudo-code for the deposition algorithm implemented in *Forest* is as shown in Algorithm 2.1[1] and illustrated in Figure 2.5.

[1]Algorithms in this dissertation are presented in standard Pidgin Algol format. Some algorithms are presented in a simpler pseudo-code format. It is a simple but tedious task to translate Pidgin Algol code to actual program code in a high level language. For more information on Pigdin Algol, the interested reader is referred to Aho-Hopcroft-Ullman [2]. In addition, several primitive data structure constructs and functions as described in Preparata-Shamos [43] are defined and used. A set S and an element u of a universal set S^U are defined. S is defined to be a subset of S^U. The following set operations are defined:

- IS_MEMBER(u, S). Is $u \in S$?

Deposition proceeds by first extracting all edges (line 2 in Algorithm 2.1) that are exposed to the ambient (an attribute of the edge object). If there are N edges, this is a simple one time traversal of all the edges in the geometry and executes in $O(N)$ time. The exposed edges are then ordered (lines 8-14) such that the ambient lies to the left as the edges are traversed. The ordering procedure starts by marking the first exposed edge as the current edge (line 6). The algorithm then finds another edge that shares the same point as the current edge (line 9). The current edge is removed from the list of exposed edges (line 12) and the edge that has been found is updated as the current edge (line 13). The recursive search procedure orders one end of the exposed edges. Restarting the entire procedure orders the other end (line 5). Line 9 hides a potentially quadratically complex algorithm. If the FIND routine is $O(M)$, then, in the worst case, the algorithm runs in $O(M(M+1)/2)$ time for M exposed edges. Since the ordering is performed based upon the geometrical connectivity, the only way to ameliorate the quadratic nature of the algorithm is to store additional connectivity information. If every point contains a list of edges that contain it, then the FIND routine executes in $O(1)$ time since for most points, only two edges will contain them. Thus, the quadratically complex algorithm need not be executed at all when a richer double hierarchy is used for representing the geometry. This algorithm illustrates the trade-off between memory usage and execution efficiency of a minimal geometrical hierarchy.

Returning to the algorithm, the orientation of the string of exposed edges with respect to the ambient is determined so that the deposited material is properly oriented

---

- MEMBER(i, S). Return the ith member of S.
- INSERT(u, S). Make u an element of set S.
- INSERT(u, S, w). Insert u in set S after element w.
- DELETE(u, S). Remove element u from set S *if* u is a member.
- FIND(S, L). Find and return an element u in the set S that meets the criteria L.
- MIN(S). Return the minimum element of S.
- MAX(S). Return the maximum element of S.
- DUPLICATE(S). Duplicate the set S.

(lines 15-19). Note that the geometry hierarchy imposes a strict counter-clockwise edge ordering scheme as the boundary of a region is traversed. Thus checking the orientation of *any* exposed edge in a region determines the orientation of the string of exposed edges. If the edge is oriented in the same manner, both in the region boundary and on the exposed edges string, then the ambient is to the right of the exposed edges string. Otherwise, the ambient is to the left.

Once the edges are properly ordered and the orientation of the ambient is determined, a new set of points that lie directly perpendicular to the exposed surface above each point on the string of exposed edges is allocated (line 20). The distance from the exposed surface to these points is equal to the deposition thickness. This determines the coordinates of the points (lines 22-24). Edges are formed along the new string of points. Note that there will be $M+1$ new points and $M+2$ new edges (lines 25-32). A new boundary and a new region are then formed creating a new material above the old exposed surface. These operations all execute in $O(M)$ time. Thus, the complexity of the overall algorithm is $O(N)+O(M^2)$.

Algorithm 2.1 Deposition

procedure Deposition(material, thickness)
Input: current geometry, deposited material and thickness
Output: new geometry
begin

    (find all the exposed edges in the geometry *)*
1. $E := \emptyset$;
2. **for** *edge* := first edge **until** last edge **do**
        if *edge* is exposed **then**
            INSERT*(edge, E);*

    (order the exposed edges *)*
3. $N := \emptyset$;
4. *firstedge* := MEMBER*(0, E);*
5. **for** i := *1* **until** *2* **do**

```
    edge := firstedge;
    INSERT(edge, N);
    repeat
        nextedge := FIND(E, edge that shares point with edge);
        if (nextedge ≠ ∅) then
            INSERT(nextedge, N);
            DELETE(edge, E);
            edge := nextedge;
    until (nextedge = ∅);

(* find the orientation of the exposed edges string *)
ned := number of exposed edges;
region := a region containing an exposed edge;
if (exposed edge is oriented the same manner in the region
   boundary and in the exposed edges string) then
   ambient_orientation = RIGHT;
else ambient_orientation = LEFT;

(* allocate new points for top boundary of deposited region *)
(* and set their coordinates *)
P := array of new points;
for i := 1 until i < ned do
    n := unit vector normal to exposed edge i;
    q := starting point of exposed edge i;
    coordinates of P(i) := coordinates of q + n * deposited material thickness;

(* connect these new points with edges *)
E := array of new edges;
starting point of E(0) = starting point of first exposed edge;
ending point of E(0) = P(0);
for i := 1 until i ¡ ned do
```

```
        starting point of E(i) = P(i);
        ending point of E(i) = P(i + 1);
starting point of E(ned) = end point of last exposed edge;
ending point of E(ned) = P(ned);

connect these new edges and the exposed edges to form
  a new boundary and a new region
end
```

As can be seen from Figure 2.5, the result of a geometrical deposition is a crude approximate view of an actual deposition. If greater accuracy is desired, then the number of exposed points along the geometry should be increased or a physically based simulation must be performed as discussed in Chapter 3.

2.4 Geometric Etching

Etching is performed by a Boolean geometric operation. The etch proceeds by defining an etch region. This region is subtracted from each region in the geometry as shown in Figure 2.6. The algorithm is illustrated in Figure 2.7 and shown in Algorithm 2.2.

The intersection i, between the etch region e, and a region in the geometry r, is first computed (line 3 in Algorithm 2.2). If the etch region is fully enclosed within the geometry region, then the etch is completed by simply defining the geometry region to have two boundaries (lines 4-5). The outer boundary is the old boundary and the inner boundary is the etch region's boundary. If the etch region fully contains the geometry region, then the geometry region is simply deleted (lines 6-7). Otherwise, the intersection between the etch region and the geometry region will be as shown in Figure 2.7. The geometry region r, is to be replaced with the many regions that lie outside the etch region e. The algorithm proceeds as follows. First a point p, of region r's boundary that is not part of the intersection (member of set P), is found (line 13). Starting from this point, all subsequent points of R that are also members of P are collated to form a new boundary B, until the intersection with the etch region is found

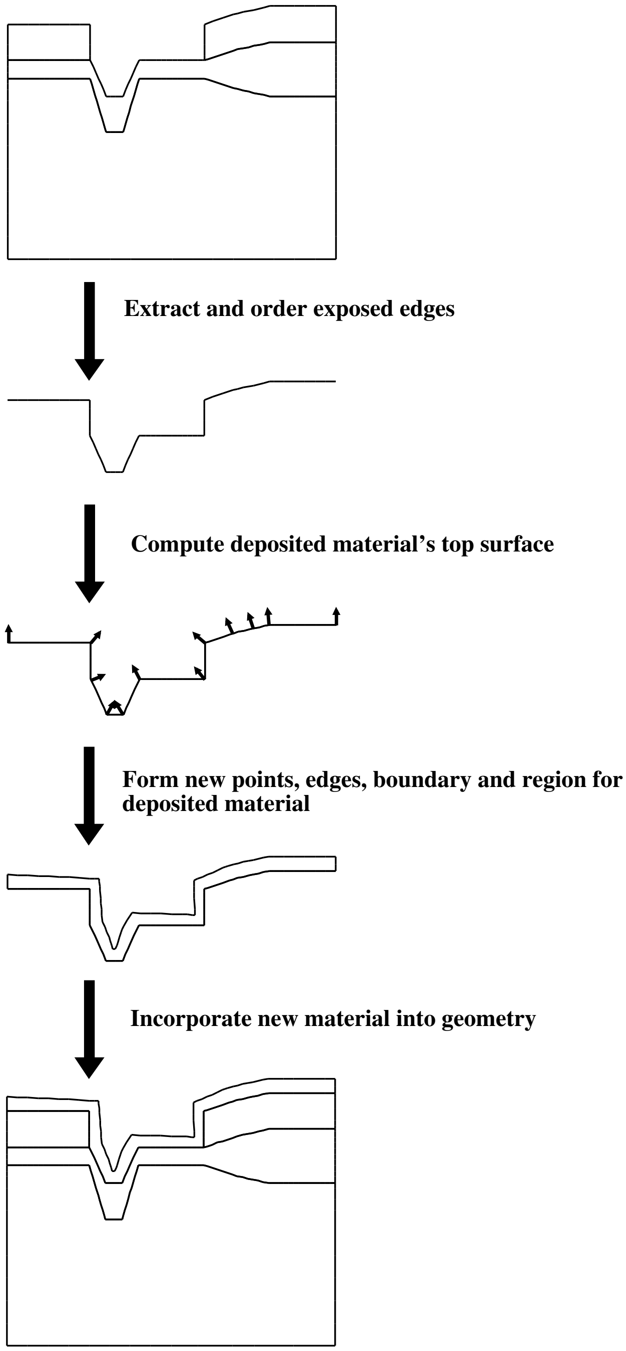


Figure 2.5: Illustration of the deposition algorithm for a boundary rep geometry model.

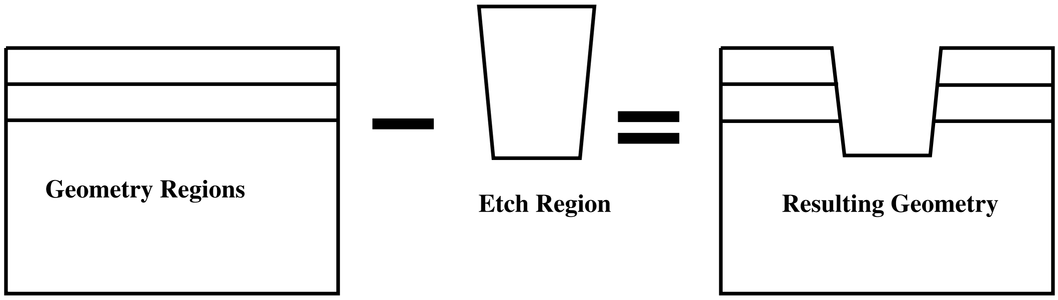


Figure 2.6: Illustration of Boolean geometric subtraction. The etch region is subtracted from each region in the geometry.

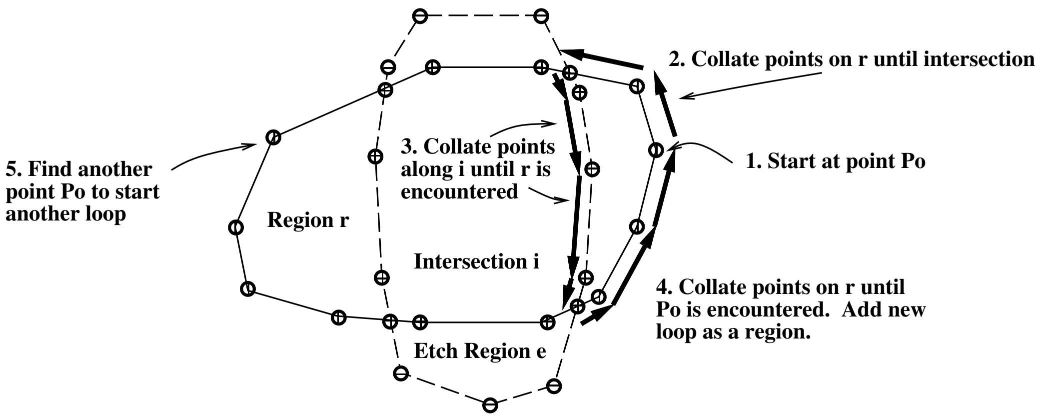


Figure 2.7: Illustration of Boolean geometric subtraction algorithm for etching. An etch region e is subtracted from the geometry region r. The intersection of e and r is first computed and points comprising loops outside the intersection are collated to form new regions.

(lines 15-19). Then, the collation proceeds along the intersection from the set I until the region r is encountered (lines 20-25). The points of the region r in set R are then collated until the first point is encountered (lines 26-29). This closed loop then forms a new region and is added to the geometry data structures (line 30). The procedure is repeated until all the points of the geometry region's boundary that are outside the intersection are part of new regions (lines 14, 31-32).

Algorithm 2.2 Etching

```
procedure Etch
begin
1.  e := Etch Region;
2.  r := Geometry Region;
3.  i := RegionIntersect(e, r);
4.  if i = e then (* etch region inside region *)
5.     add boundary of e as internal boundary of r;
6.  elseif i = r then (* geometry region inside etch region *)
7.     delete r;
    else
8.  E := Points of e;
9.  R := Points of r;
10. I := Points of i;
11. P := R - I; (* set of points of r that are not part of the intersection *)
12. B := ∅;
13. first := p := MEMBER(0, P);
14. while (P ≠ ∅) do
          (* get the points that are outside *)
15.       INSERT(p, B);
16.       np := FIND(R, point next to p);
17.       if np ∈ P then
18.          DELETE(p, P);
19.          p := np;
```

```
        else
            (* get the points that are part of the intersection *)
            np := FIND(I, point on edge from p to np);
            INSERT(np, B);
            repeat
                np := FIND(I, point next to np);
                INSERT(np, B);
            until np is on r;

            (* loop back to get all the other points of r *)
            p = FIND(R, point next to np);
            while p ≠ first do
                INSERT(p, B);
                DELETE(p, P);

            add the new region formed by points in set B;
            B := ∅;
            first := p := MEMBER(0, P);
        delete region r;
end
```

Analysis of Algorithm 2.2 reveals that it is $O(N + K)$, where N in the number of points of the region and K is the number of points of the etch region. Computation of the region intersection is shown to be of linear complexity in Section 2.5.3. All points of r and e are members of the set R, I or P. The size of R is N. The largest sizes of P and I is $N + K$. The points of P, R and I are visited only once before being collated into N. Thus, the worst case performance of this algorithm is $O(2(N + K) + N)$ or $O(3N + 2K)$.

There are, however, several FIND operations to locate another point meeting certain criteria within the main **while** loop of the algorithm. If these FIND operations entail a linear search through all the points within a set, then the algorithm will be

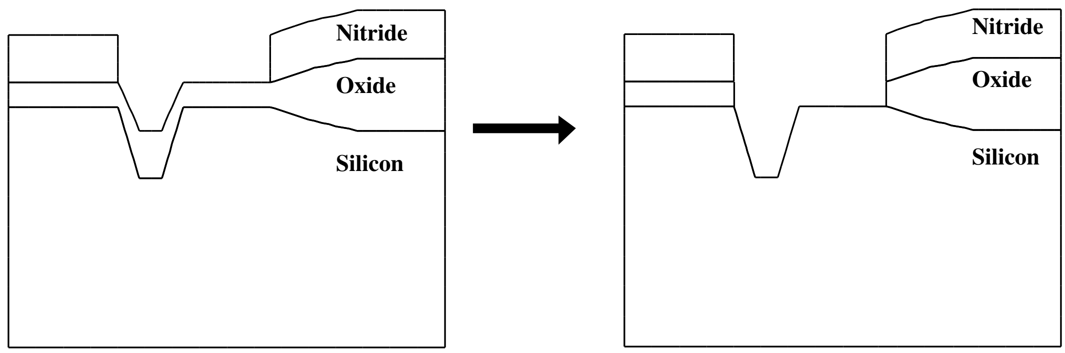


Figure 2.8: Example illustrating geometric etching. In this example, only the oxide is etched away under the nitride mask.

$O(N^2)$. Careful preprocessing reduces the complexity of these FIND operations to be $O(1)$ and thus, the whole algorithm is still $O(3N + K)$. The FIND operations on lines 16, 23 and 26 simply find a point of a region next to a given point. Since a geometrical hierarchy orders the points, this operation simply follows the hierarchy. The loop stop condition on line 25 and the FIND operation on line 20 are both $O(1)$ if the region intersection routine attaches attributes to the intersection points. For the loop stop condition, the region intersection routine attaches a region attribute to each point of I. Thus, the stop condition can be evaluated by simply checking this attribute. This attribute is also used to compute, in linear time, the set P of points on r that are not part of i. For the FIND operation, the region intersection routine must attach the intersection points as attributes to the edges of r and e. Thus, the intersection point corresponding to any edge of r or e can be found in $O(1)$ time. Thus, every step of the algorithm is linear. A sample use of this algorithm is shown in Figure 2.8 illustrating non-physical etching.

2.5 Miscellaneous Geometry Utility Algorithms

The etch and deposition algorithms discussed above rely on several utility geometry algorithms to insure that the regions are convex, their boundaries are oriented in a counter-clockwise manner, and other similar properties. The etch algorithm also requires a function to compute the intersection between two regions. These utility geometry algorithms are the subject of this section.

2.5.1 Boundary Orientation

Several algorithms in the geometry server depend on an implicit counter-clockwise ordering scheme. The region is defined to be on the left as a boundary is traversed. If this orientation is ever reversed, due to user error or due to geometry movement, then the geometry server automatically detects the reversal and fixes it. In this section, the algorithm to determine the orientation of a boundary is presented in Algorithm 2.3. Reversing the orientation of a boundary once an anomaly is detected is simple - edges are simply ordered in the opposite manner.

All the points comprising the boundary are traversed once and four indexes are set. The indices *right*, *top*, *left*, and *bottom* correspond to the point index corresponding to the right most, top most, left most, and bottom most points respectively. If at least three of the indices *right*, *top*, *left*, and *bottom* are distinct from one another, then their ordering determines the orientation. Starting from the lowest index, if consecutive indices increase as all the indices are traversed counter-clockwise, then the boundary is oriented in a counter-clockwise manner. Otherwise, the boundary is oriented in a clockwise manner. This simple algorithm is $O(N)$ with N being the number of points on the boundary. In the degenerate case with only two distinct indices, *right* and *left* are fixed to their original values while *top* and *bottom* are set to the next top most and the next bottom most point indices. Rerunning the algorithm gives the proper orientation.

Algorithm 2.3 Boundary Orientation

procedure BoundaryOrientation
Input: boundary
Output: clockwise or counter-clockwise orientation
begin
    right := top := left := bottom := 0;
    (set the indices by traversing through boundary points *)*

```
for i := 0 until number of boundary points do
    if x[i] < x[left] then
        left := i;
    if x[i] > x[right] then
        right := i;
    if y[i] > y[top] then
        top := i;
    if y[i] < y[bottom] then
        bottom := i;
end

Boolean A = (top ≥ right);
Boolean B = (right ≥ bottom);
Boolean C = (bottom ≥ left);
Boolean D = (left ≥ top);

if ((A and B and C) or (A and B and D) or
   (A and C and D) or (B and C and D)) then
   orientation = COUNTERCLOCKWISE;
else
   orientation = CLOCKWISE;
return (orientation);
end
```

2.5.2 Removing Holes and Making Regions Convex

The general boundary representation model allows users and client programs to specify general polygonal regions. Such regions may in general be concave, have holes in them, etc. Most computational geometry algorithms are $O(N)$ or at least $O(N \log(N))$ for convex regions but are $O(N^2)$ for general polygons [37]. Thus, it is advantageous to preprocess the geometry and to reduce general polygons into convex polygons. In this

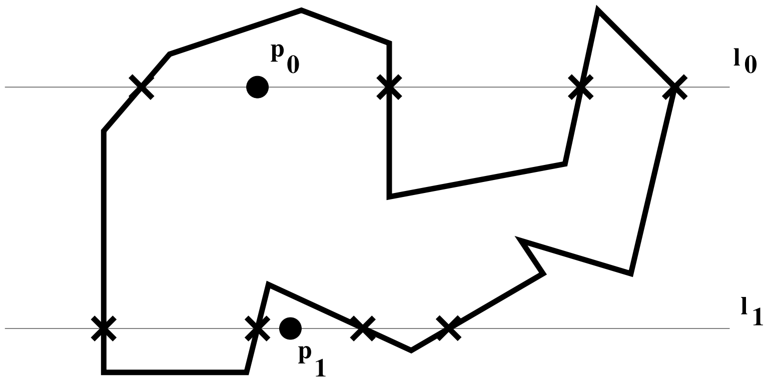


Figure 2.9: Algorithm to determine if a point is inside a polygon. A horizontal line passing through the point is drawn. If the line intersects the polygon an odd number of times (as for p_0), the point is inside the polygon. Otherwise, the point is outside.

section, the preprocessing of non-convex regions is considered. Polygons with holes are divided into two, such that each of the resulting polygons has no holes. Polygons with vertex angles greater than 180 degrees are also divided such that all vertex angles are less than 180 degrees.

A region with holes in the interior has more than one boundary. The algorithm to reduce the number of boundaries to one, first determines which of the region's boundaries is the outer boundary. The test for outer boundary is as follows. A point inclusion test is performed with the first point of each boundary and each of the other boundaries. The boundary whose point is not inside any other boundary is the outermost boundary. A single point inclusion test for a general polygon can be performed in linear time using a very simple algorithm [37] illustrated in Figure 2.9. A horizontal line passing through the point is drawn from negative infinity to positive infinity. The number of intersections that this line makes with the polygon as the line is traversed from negative infinity to positive infinity is counted. If there are an even number of intersections before the test point is encoutered, then the test point is outside the polygon. If an odd number of intersections are encountered, then the test point is inside the polygon.

Once the outermost boundary has been determined, the algorithm proceeds as illustrated in Figure 2.10 and shown in Algorithm 2.4. Starting from any edge $\vec{e}$

between points p and q of the inner boundary, the angular bisectors $\vec{be}_p$ and $\vec{be}_q$ between $\vec{e}$ and the previous edge $\vec{e}_p$ and the next edge $\vec{e}_q$ are computed (lines 9-10). The edges of the outer boundary, $\vec{ie}_p$ and $\vec{ie}_q$, that intersect with $\vec{be}_p$ and $\vec{be}_q$ (lines 11-12) are found. New edges $\vec{e}_{ip}$ and $\vec{e}_{iq}$ are formed connecting the vertices p and q with the beginning points of $\vec{ie}_p$ and $\vec{ie}_q$ respectively (lines 15-16). Edges are collated to form two new boundary loops L_1 and L_2. L_1 starts with edge $\vec{e}$, and $\vec{e}_{iq}$ (lines 18-19). All the edges of the outer boundary between $\vec{ie}_q$ and $\vec{ie}_p$ are then collated into L_1 (lines 21-22). Finally, $\vec{e}_{ip}$ finishes the boundary loop (line 23). L_2 starts with $\vec{e}_{ip}$ and $\vec{ie}_p$ (lines 26-27). All subsequent edges of the outer boundary are then collated into L_2 until $\vec{ie}_q$ is reached (lines 29-30). Edge $\vec{e}_p$ is collated. Then all edges of the inner boundary from $\vec{e}_q$ until $\vec{e}_p$ are collated into L_2 (lines 33-34). The finished boundary loop is then added as a new region.

Algorithm 2.4 Remove Holes

procedure RemoveHoles
Input: region
Output: new regions without holes
begin

1. B_{in} := inner boundary of region;
2. B_{out} := outer boundary of region;
3. E_{in} := set of edges of B_{in};
4. $\vec{e}$:= any edge in E_{in};
5. $\vec{e}_p$:= FIND(E_{in}, edge before $\vec{e}$);
6. $\vec{e}_q$:= FIND(E_{in}, edge after $\vec{e}$);
7. p := vertex point between $\vec{e}$ and $\vec{e}_p$;
8. q := vertex point between $\vec{e}$ and $\vec{e}_q$;
(compute angle bisectors *)*
9. $\vec{be}_p$:= bisecting edge of angle between $\vec{e}$ and $\vec{e}_p$.
10. $\vec{be}_q$:= bisecting edge of angle between $\vec{e}$ and $\vec{e}_q$.
11. $\vec{ie}_p$:= FIND(B_{out}, edge that intersects with $\vec{be}_p$);
12. $\vec{ie}_q$:= FIND(B_{out}, edge that intersects with $\vec{be}_q$);

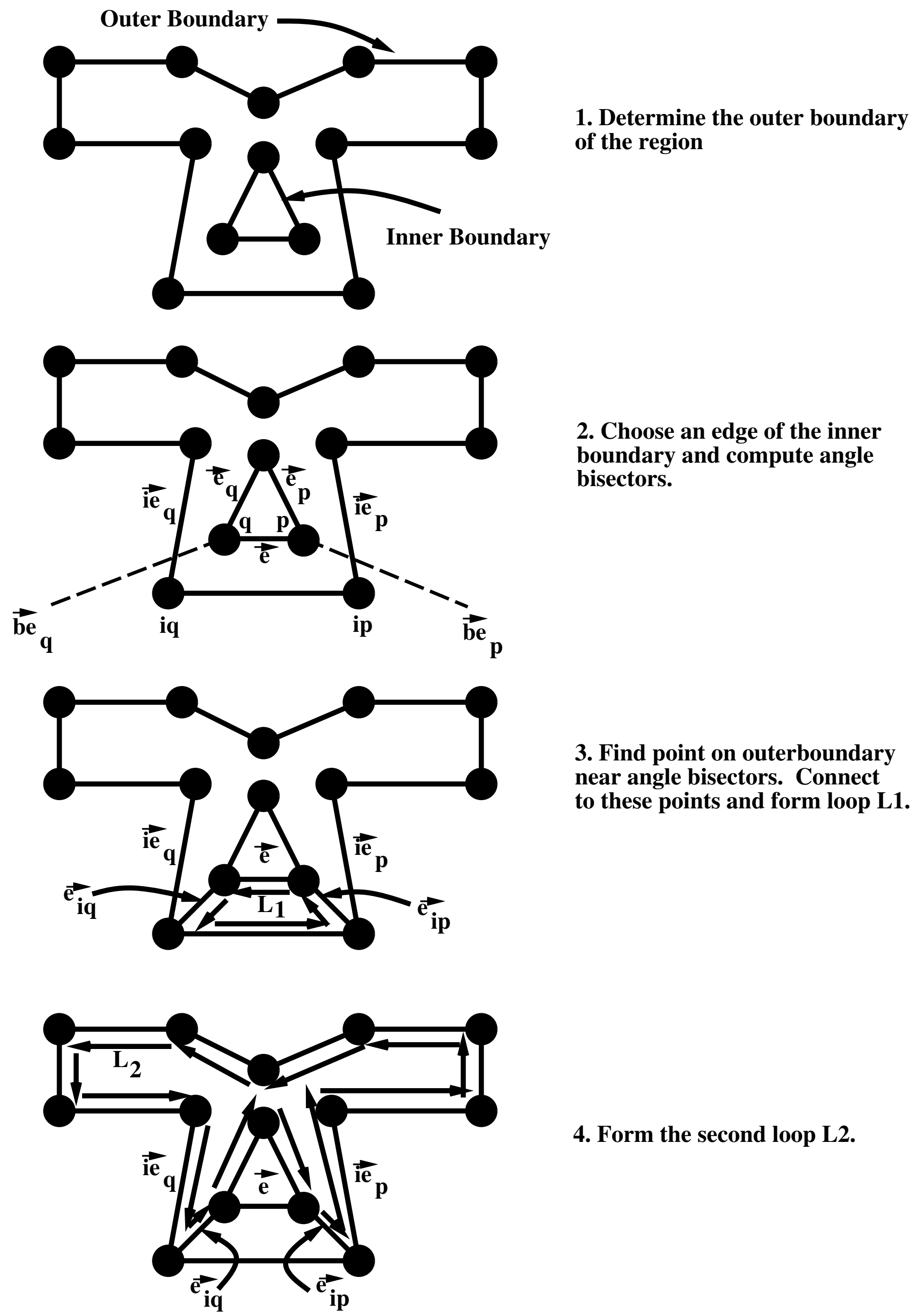


Figure 2.10: Algorithm to remove holes from a region. Two new regions are formed to replace the region with the hole.

13. ip := start point of $\vec{ie_p}$;

14. iq := end point of $\vec{ie_q}$;

15. $\vec{e_{ip}}$:= new edge connecting p and ip;

16. $\vec{e_{iq}}$:= new edge connecting q and iq;

(form a new loop *)*

17. $L_1 := \emptyset$;

18. INSERT$(\vec{e}, L_1)$;

19. INSERT$(\vec{e_{iq}}, L_1)$;

20. $\vec{ed}$:= FIND$(B_{out}$, edge after $\vec{e_{iq}})$;

repeat

    begin

21.   INSERT$(\vec{ed}, L_1)$;

22. $\vec{ed}$:= FIND$(B_{out}$, edge after $\vec{ed})$;

    end

until $(\vec{ed} = \vec{e_{ip}})$;

23. INSERT$(\vec{e_{ip}}, L_1)$;

24. add L_1 as a new boundary and add corresponding region;

(form a new loop *)*

25. $L_2 := \emptyset$;

26. INSERT$(\vec{e_{iq}}, L_2)$;

27. INSERT(ie_p, L_2);

28. $\vec{ed}$:= FIND$(B_{out}$, edge after $\vec{ie_p})$;

repeat

    begin

29.   INSERT$(\vec{ed}, L_2)$;

30. $\vec{ed}$:= FIND$(B_{out}$, edge after $\vec{ed})$;

    end

until $(\vec{ed} = \vec{ie_q})$;

31. INSERT$(\vec{e_{iq}}, L_2)$;

32. $\vec{ed} = \vec{e_q}$;

repeat

```
        begin
33.         INSERT(ed⃗, L₂);
34.         ed⃗ := FIND(B_in, edge after ed⃗);
        end
35. until (ed⃗ = e⃗_p);
36. add L₂ as a new boundary and add corresponding region;
end
```

The algorithm executes in $O(N)$ where N is the number of edges of both the inner and outer boundaries. Each edge on both the outer and inner boundaries is visited only once for collation. Thus, the formation of the two new loops is $O(N)$. The FIND operations on lines 12 and 13 involve a linear search through all the edges of the outer boundary. Since these lines are executed only once, the execution time is still $O(N)$. All the other FIND calls are $O(1)$ as they use the geometry hierarchy.

A similar algorithm is used to convert non-convex polygon regions into convex regions. The angle that makes the polygon non-convex is bisected. The intersection of the bisector with an edge of the polygon is then found. A new edge is formed from the concave vertex to the end point of the intersecting edge. Two new boundary loops are formed to replace the current loop and region. Figure 2.11 shows an example use of the algorithms discussed in this section. A device cross section containing polygonal regions with a hole and several concave angles is reduced to a set of convex polygons.

2.5.3 Region Intersection

The etch algorithm above requires computation of the intersection between two regions. Several ways of computing this intersection exist. In 2D, algorithms that grow linearly with the number of edges in the regions have been reported for convex regions [43]. For non-convex regions, only a simple quadratically complex algorithm is possible. In this work, a modified version of the algorithm proposed by O'Rourke et. al. [37] is used for convex regions. The main procedure for computing the intersection as shown in Algorithm 2.5 is identical. The sub-procedures to advance the appropriate boundary edge and to determine which polygon is on the "inside" are

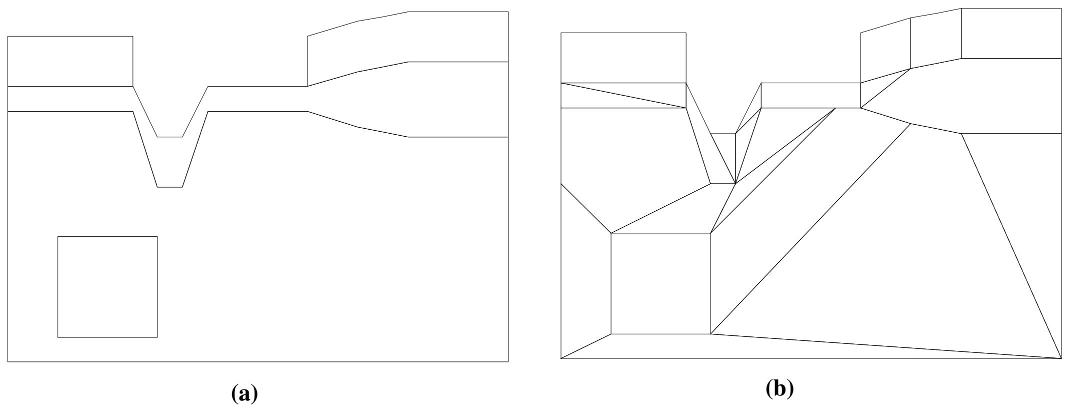


Figure 2.11: Example use of hole removal and concave region reduction algorithm. The original geometry (a) is reduced to a geometry consisting of convex regions only (b).

modified versions as implemented in the *Forest* geometry server. These modifications are essential for handling a number of degenerate cases that routinely occur in process simulation. (Bold faced line numbers in the algorithm denote the modified portions of the algorithm.) The modified algorithm is discussed herein. Correctness of the algorithm is rigorously proved in [37].

The algorithm depends on the observation that the intersection is on the left half plane of all the edges of both regions. The algorithm starts with the first edge of each of the regions (lines 1-2 in Algorithm 2.5). If the edges are not parallel, the intersection of the two edges is computed. If the edges do not intersect, then the edges of the regions are advanced until an intersection is found (line 11). If there is an intersection, then the intersection point is returned as output and one of the regions is marked as being on the "inside" (line 10). The algorithm stops if the first intersection point is found again denoting a complete traversal around the intersection (lines 6-7, 41-42 and 47-48) or if the edges have been advanced around each region at least twice (line 12). The edge advancement rules and marking which region is on the "inside" are the essence of the algorithm. These rules find the common left half planes of all the edges.

Algorithm 2.5 Region Intersection

procedure RegionIntersection
Input: Region P and Region Q
Output: Region $I := P \cap Q$
begin
1. $\vec{p}$:= First edge of Region P;
2. $\vec{q}$:= First edge of Region Q;
3. **repeat**
4.   **if** $\vec{p}$ and $\vec{q}$ are not parallel **then**
5.   **if** $\vec{p}$ and $\vec{q}$ intersect **then**
6.   **if** intersection is the same as the first intersection and the
7.   first intersection was not found in the last iteration **then halt**;
8.   **else**
9.   output intersection point
10.   *SET inside;*
11.   ADVANCE $\vec{p}$ or $\vec{q}$
12. **until** loop has been executed $2 * (\mid P \mid + \mid Q \mid)$ times;
end

Figure 2.12 illustrates the edge advancement rules shown in lines 13-25 in the algorithm. If edges of both regions point towards the left half planes of each another, then the intersection is on the left of the left-most edge. Hence the right edge should be advanced as in Figure 2.12a (lines 16-17 and 23-24). If both edges point away from each other's left half planes, then the intersection is to the left of the edge whose end point is in the other's left half plane. Thus, the outer edge should be advanced as in Figure 2.12c (lines 18 and 25). If one of the edges points towards the other left half plane while the other points away, then the one that points away should be advanced as in Figure 2.12b (lines 14-15 and lines 21-22). For parallel edges that do not point towards one another's left half plane, special rules are necessary as follows. If one edge is ahead of the other, then the leading edge should be advanced as shown in Figure 2.12d (lines 26-29). If one of the edges is clearly on the left as shown in Figure 2.12e, then the right edge should be advanced (lines 35-37). Figure 2.12e

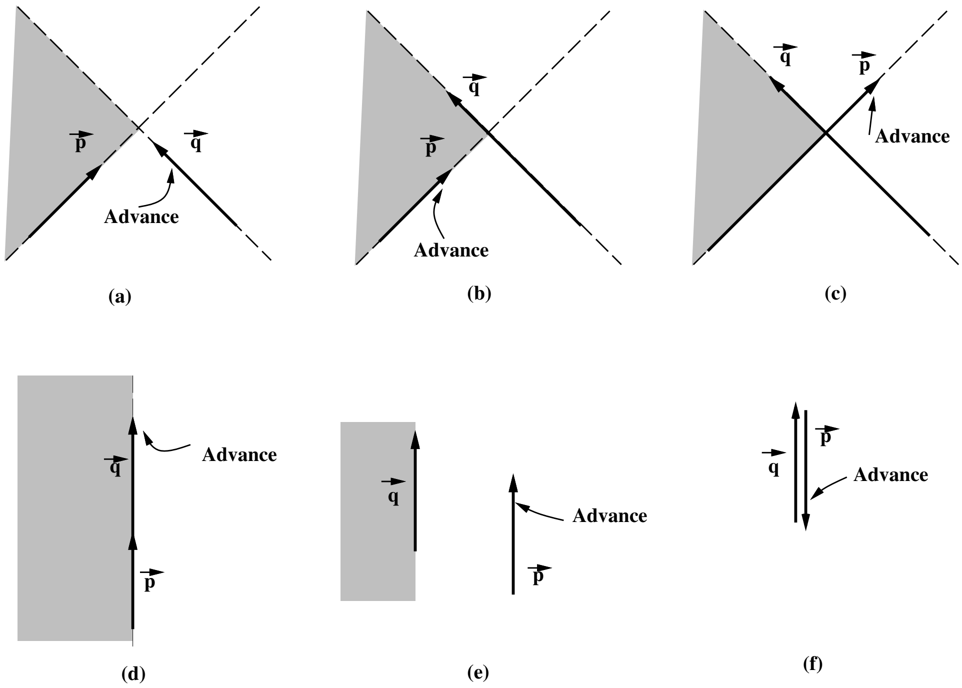


Figure 2.12: Advancement rules for the region intersection algorithm. Cases (d), (e) and (f) are modifications from the O'Rourke algorithm to handle parallel edges.

illustrates the very common case of non-intersecting regions whose edges overlap. In such instances, the edge of the region on the right should be advanced (lines 30-32). These rules will advance the proper edge until the next edge intersection is found.

Algorithm 2.5 Region Intersection (continued)

procedure ADVANCE $\vec{p}$ or $\vec{q}$

begin

13. **if** $\vec{q} \times \vec{p} > 0$ **then**

14.   **if** $\vec{q}$ is in the right half plane of $\vec{p}$ **then**

15.   *advance := q;*

16.   **else**

17.   **if** $\vec{p}$ is in the left half plane of $\vec{q}$ **then**

18.   *advance := q;*

19.   *else advance := p;*

else if $\vec{q} \times \vec{p} < 0$ **then**
  if $\vec{p}$ is in the right half plane of $\vec{q}$ **then**
    advance := p;
  else
    if $\vec{q}$ is in the left half plane of $\vec{p}$ **then**
      advance := p;
    else *advance := q;*
else if $\vec{q} \times \vec{p} = 0$ **then**
  if $\vec{q}$ and $\vec{p}$ overlap and $\vec{q}$ is ahead of $\vec{p}$
    advance := q;
  else if $\vec{p}$ and $\vec{q}$ are anti-parallel
      advance := vector of right most region;
  else
      if $\vec{p}$ is in the left half plane of $\vec{q}$ **then**
      advance := q;
      else *advance := p;*

if *advance* $= p$ **then**
  if *inside* $= p$ **then**
    output starting point of $\vec{p}$;
    if $\vec{p}$ was advanced at least once and
      the starting point of $\vec{p}$ is the first intersection **then** *halt;*
  $\vec{p}$:= next edge of P;
else if *advance* $= q$ **then**
  if *inside* $= q$ **then**
    output starting point of $\vec{q}$;
    if $\vec{q}$ was advanced at least once and
      the starting point of $\vec{q}$ is the first intersection **then** *halt;*
  $\vec{q}$:= next edge of Q;
end

As the edges are advanced, a flag is maintained denoting which region is on the *inside*. The *inside* region borders on the intersection. If an edge of the *inside* region is advanced, then the starting point of the edge is output as a point of the intersection (lines 38-40 and 44-46). The rule for determining which region is on the *inside* is a simple test that determines which edge is in the left half plane of the other edge (lines 66-69). For non-degenerate cases, the result of this test will not be ambiguous. For degenerate cases, additional tests are necessary. If one edge ends at the beginning of the other, then the left half plane test is performed with respect to the lagging edge (lines 62-64).

If the edges are parallel and if they overlap, *inside* is set to the region whose edge lags the other (lines 50-55). If the edges are anti-parallel or if one region is clearly to the left of the other, then *inside* is set to the left most region (line 57). Note that these rules are complimentary to the rule used for edge advancement for similar cases. If the edges are not parallel but both edges end at the same vertex point, then the region whose edge is on the outside should be set as the *inside* region (lines 59-61). Since the regions are convex, any intersection between the regions indicates that the *inside* region is now headed outside the intersection. If it does not head outside the intersection, then this error will be corrected at the next iteration.

Algorithm 2.5 Region Intersection (continued)

procedure SET *inside*
begin
50. **if** $\vec{q} \times \vec{p} = 0$ **then**
51.   **if** $\vec{q}$ and $\vec{p}$ overlap **then**
52.   **if** $\vec{q}$ is ahead of $\vec{p}$ **then**
53.   *inside := p;*
54.   **else**
55.   *inside := q;*
56.   **else**
57   *inside* := left most region;
58 **else**

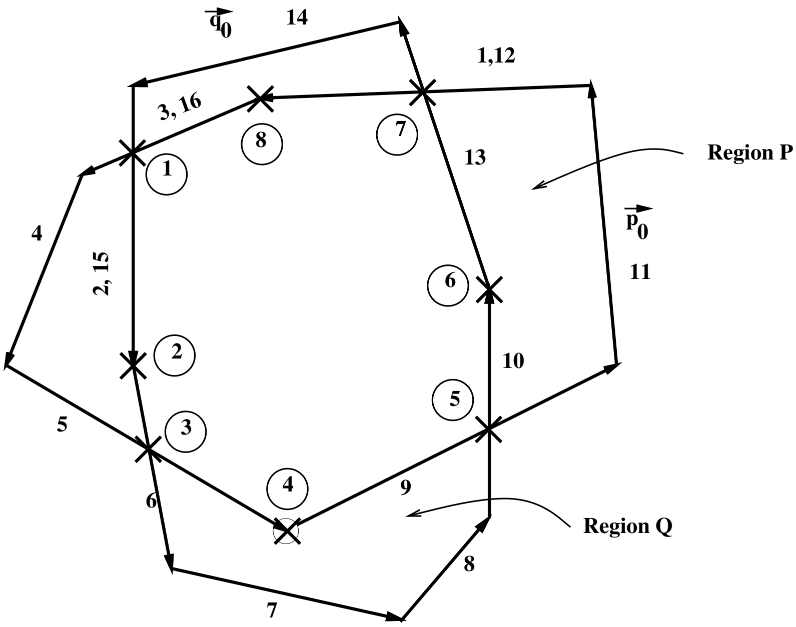


Figure 2.13: Illustration of the region intersection algorithm. The algorithm starts with the initial edges $\vec{p}_0$ and $\vec{q}_0$ from regions P and Q respectively. Edges are then advanced as indicated by numbers next to the edges. Points of the intersection are then found as indicated by circled numbers next to the points.

if end points of $\vec{p}$ and $\vec{q}$ coincide **then**
    if $\vec{q} \times \vec{p} \geq 0$ **then** *inside := q;*
    else *inside := p;*
else if end point of $\vec{p}$ coincides with the beginning point of $\vec{q}$ **then**
    if $\vec{q} \times \vec{p} \geq 0$ **then** *inside := p;*
    else *inside := q;*
else
    if $\vec{p}$ is in the left half plane of $\vec{q}$ **then**
        inside := p;
    else
        inside := q;
end

end

Figure 2.13 illustrates the use of the above algorithm. Starting with the initial edges $\vec{p}_0$ and $\vec{q}_0$, the algorithm advances each edge as indicated by numbers next to the edges. The intersection points are then found as indicated by circled numbers next to the points.

As discussed in Section 2.4, several additional attributes are useful when computing a region intersection. As an intersection between two edges is found, the two edges may be stored as attributes of the intersection point. If advancement of an edge of the *inside* region, then the region may be stored as an attribute of the advanced point. As discussed earlier, these attributes reduce the complexity of the etch algorithm. Other roles played by this algorithm will be further elucidated in Chapter 4 on grid generation.

2.6 Summary and Conclusions

In this chapter, geometry representation and algorithms to manipulate the geometry are discussed. The data structures in the *Forest* server represent the geometry using a boundary representation model. Simple geometric etching and deposition algorithms are discussed. Utility geometry algorithms are also presented illustrating the complexity of building a full scale geometry modeler even in 2D. However, these algorithms are sufficient to build a reasonable representation of the wafer after processing.

Figure 2.14 illustrates the power of such virtual processing for a trench capacitor DRAM process. Structure (a) shows a silicon wafer with 350Å oxide and 1200Å nitride layers. A four micron deep trench is then etched resulting in the structure (b). An 800Å oxide deposition and etch back using two masks result in structures (c), (d) and (e) respectively. Structure (f) shows the result of a 500Å poly deposition. This poly layer abuts the right side of the trench where the source implant will be performed. The poly is etched back in structure (g) and a thin dielectric nitride of 50Å is deposited as shown in structure (h). A blanket poly deposition fills the trench forming the other plate of the trench capacitor. The poly is then etched back to a height of 1500Å above the surface resulting in structure (i). An oxide layer is then deposited (structure (j)) and patterned (structure (k)). Poly and nitride are then etched using the oxide mask resulting in structure (l) and (m). An additional oxide layer is deposited and etched to prevent a short between the capacitor plates resulting in structure (n). Finally, the access transistor gate poly is deposited and patterned resulting in the final structure (o). Thus, by using only geometric deposition and etching, an estimate of the resulting geometry can be obtained. This geometry can then be gridded and used for subsequent physically based device simulation.

While the algorithms presented in this chapter are all formulated in 2D, a number of authors have extended both the representation and the associated algorithms to the 3D case. However, geometry manipulation in 3D is very computationally intensive and robust algorithm implementation is difficult. TCAD developers have relied on using existing geometry modelers including the IBM Geometry Engine [29] and ACIS [1] to perform virtual processing. Promising results have been reported recently [51].

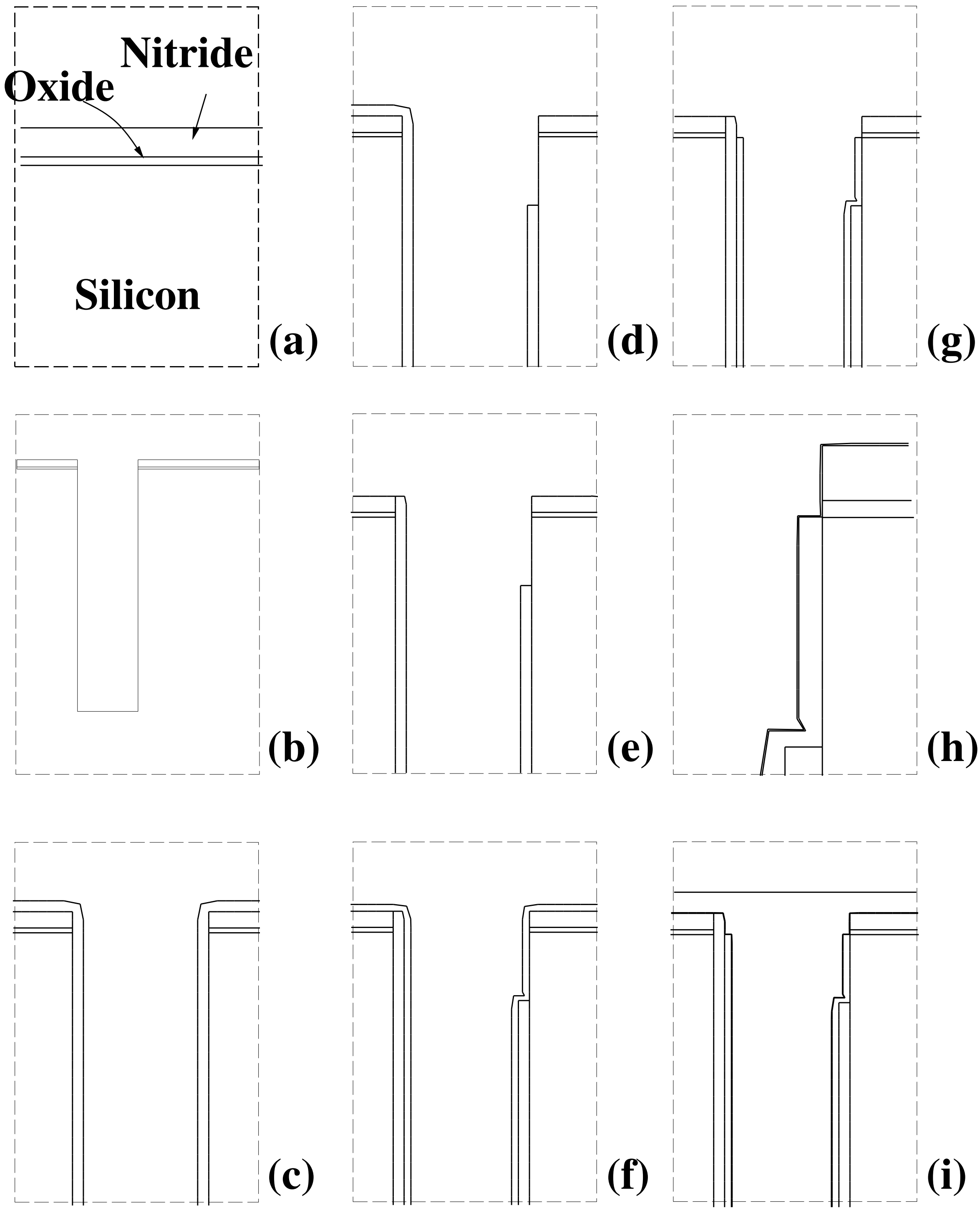


Figure 2.14: Example of virtual processing using geometry operations. Please see text for explanation of steps.

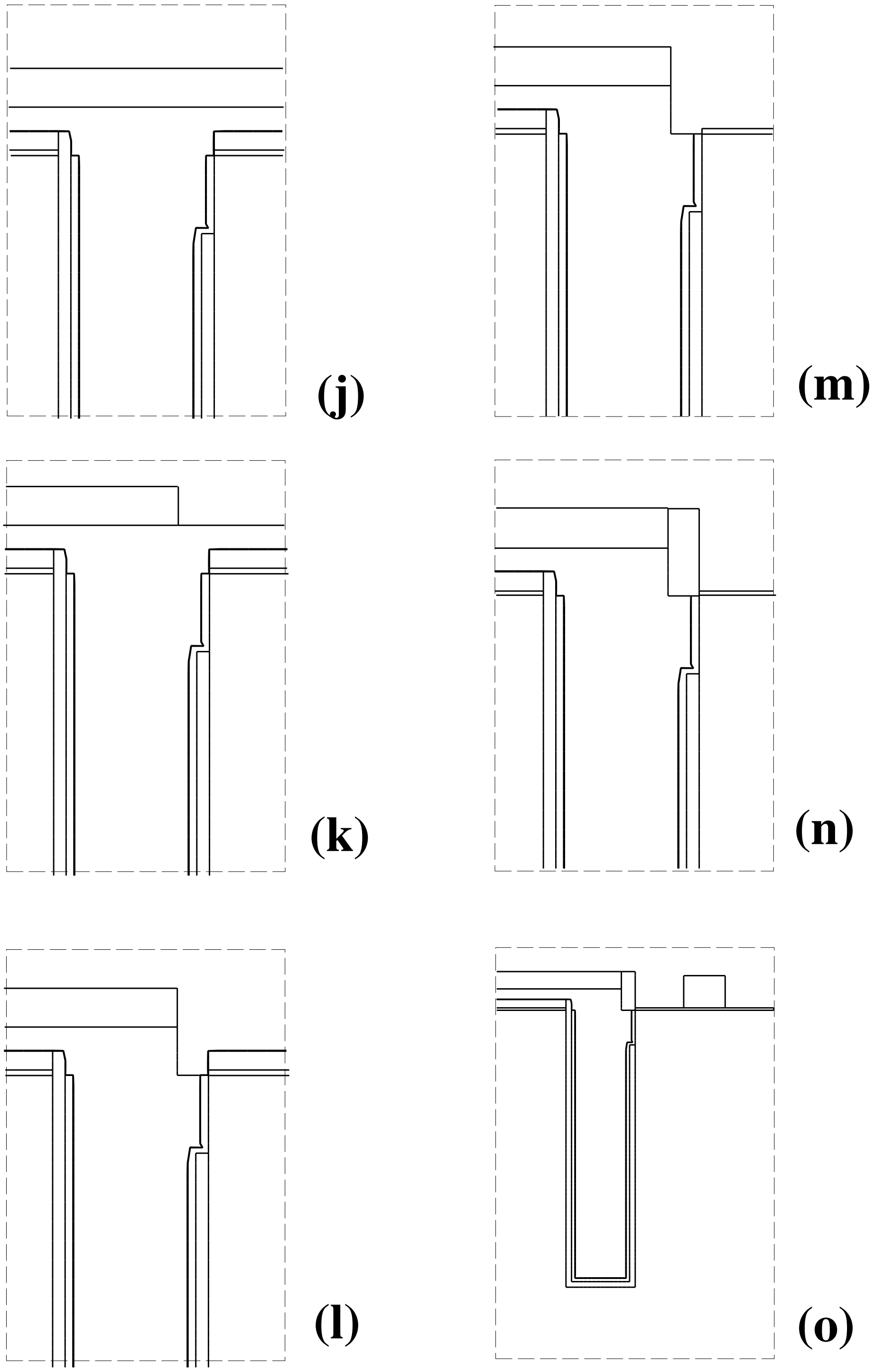


Figure 2.15: Example of virtual processing using geometry operations (continued). Please see text for explanation of steps.

Chapter 3

Geometry Motion for Topography Simulation[1]

3.1 Introduction

Geometry representation and its associated algorithms are sufficient for simulating all geometry changing steps in a process simulation. Every geometry changing step can be reduced to a Boolean addition (deposition) or subtraction (etching). Indeed, the SWR system [12] did not provide any other means for changing the geometry. For topography and oxidation simulations, however, the method of Boolean addition and subtraction is cumbersome as the geometry changes over several time steps. A naturally intuitive approach to changing the geometry would be to simply change the coordinates of the points using the computed displacement or velocity vectors. The motion of the points thus directly corresponds to material growth or etch.

In this chapter, algorithms to support geometry motion for topography simulation are presented. The topography program, SPEEDIE [34], is used to compute velocity vectors using physically based simulations. The *Forest* server then provides the necessary geometry operations. The geometry hierarchy, however, places several

[1]The help of Ze-Kai Hsiau in interfacing the geometry server *Forest* with the topography simulator SPEEDIE is gratefully acknowledged.

constraints on boundary motion. The hierarchy must be valid at all times. Boundaries of the geometry should not crash into one another, self looping structures should not form, and edges should not overlap. Algorithms to detect such anomalies and to recover from them are also presented.

3.2 Topography Simulation with SPEEDIE

The SPEEDIE topography program simulates material growth (or etch) in several consecutive steps as illustrated in Figure 3.1. First the geometry is discretized into a string of points called the surface mesh representing the top exposed surface (Figure 3.1b). Second, a transport equation is solved in the "Gas" layer above the exposed surface. This solves for the flux of a reactive species across the wafer boundary (Figure 3.1c). Next, using the flux, a velocity for the points along the surface is computed (Figure 3.1d). Finally, the surface is moved using an appropriate time step (Figure 3.1e).

Solution of the transport equation to compute the reactive species flux and the resulting growth velocity is done by several different methods. The most common method involves the solution of a ray tracing algorithm with a distributed source. The boundary is divided into a number of small elements. Flux contributions from all incident rays from the source and scattered rays from other elements are then summed forming a dense scattering matrix. The unknowns are the fluxes over the edge elements and the right hand side is the total flux as a function of the species concentration and edge orientation angle. Matrix inversion then gives the actual fluxes across each element. The resulting growth velocity is then computed as a (potentially non-linear) function of the flux and the underlying material.

The boundary representation model of the geometry server is ideally suited for topography simulation using SPEEDIE. The controlling program is SPEEDIE with *Forest* acting as a library. In the following discussion, discretization of the geometry into a string of surface points and motion of these points is presented. Several post-processing algorithms are discussed in subsequent sections.

Boundary discretization is necessitated by a device geometry that is often as coarse

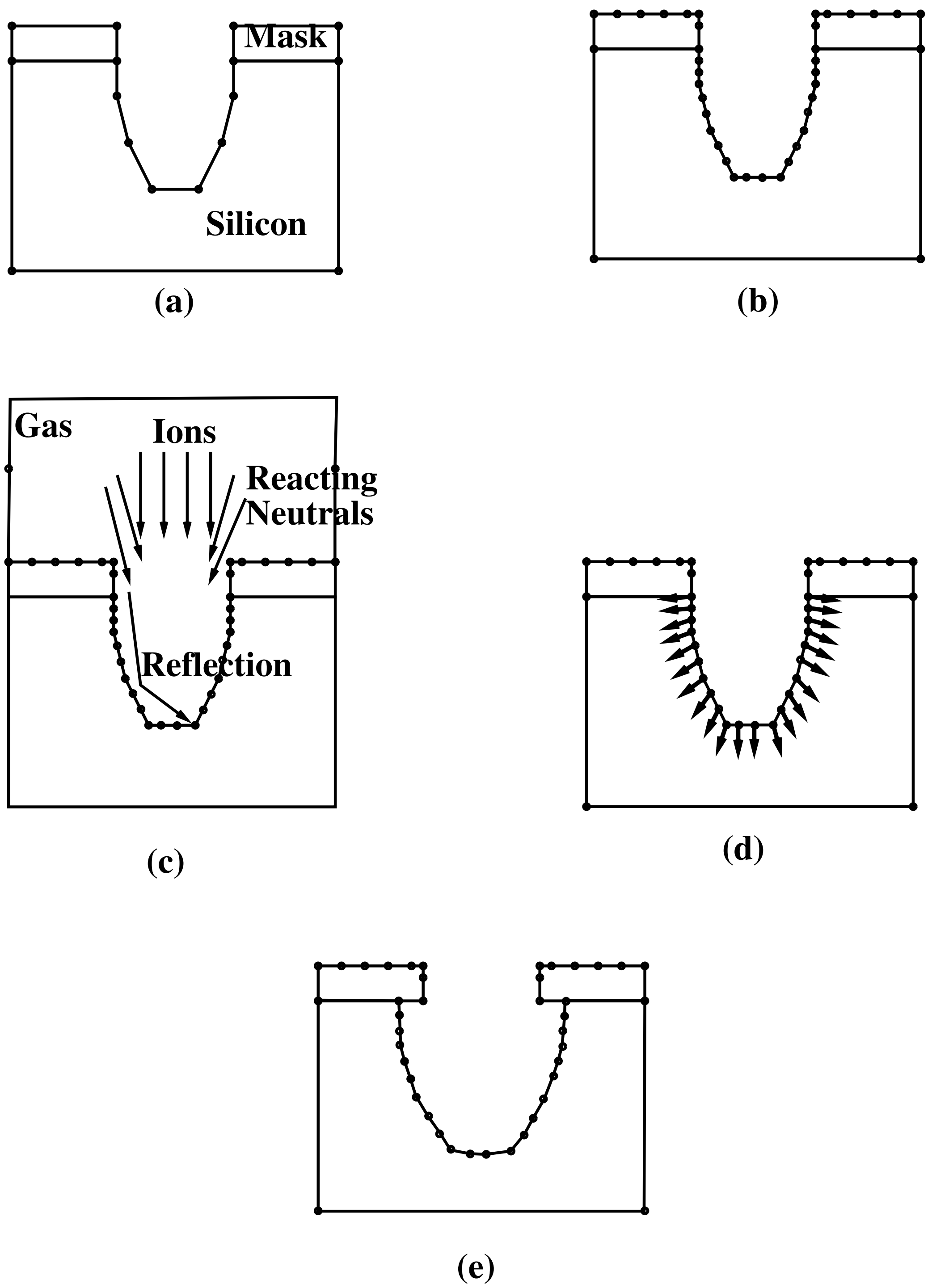


Figure 3.1: Illustration of SPEEDIE's etching/deposition algorithm. (a) The initial geometry. (b) Discretization of the geometry. (c) Solution of the transport equation. (d) Computation of the surface velocity. (d) Surface motion.

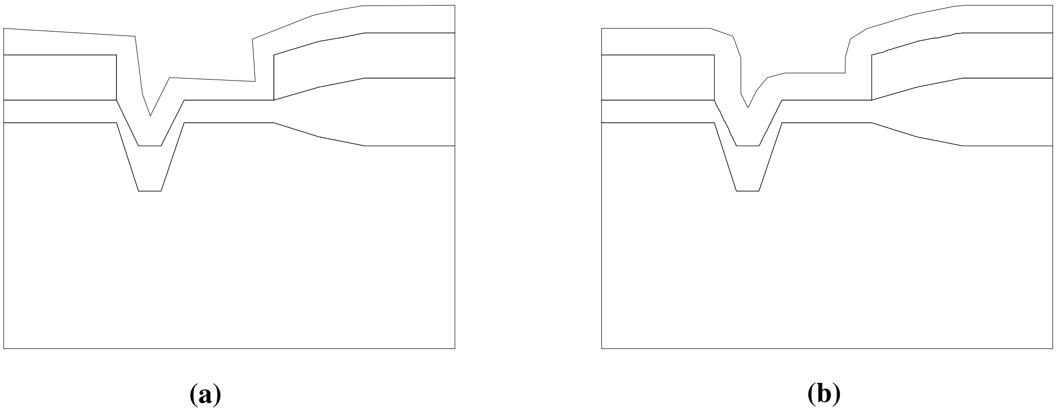


Figure 3.2: Effect of a coarse boundary representation on geometric Boolean deposition. (a) Large edges at corners tend to produce distorted surfaces. (b) The same geometric deposition with smaller edges produces better deposition profiles.

as possible to save memory. A straight line segment, for example, is represented by only one edge, regardless of the edge length. However, the granularity of the geometry is generally too coarse for obtaining accurate geometry changing simulations. As shown in Figure 3.2a, even for geometric Boolean deposition, a coarse boundary representation can lead to poor results. Boolean geometric deposition merely builds a new surface based on the old surface. Angular bisectors are drawn at each vertex and new vertices are allocated at a distance equal to the deposition thickness. For straight line horizontal segments, the angle bisectors accurately represent normal growth along the material. At corners, however, the geometry buckles as seen in Figure 3.2a. The problem occurs at both convex and concave corners, The same Boolean geometric deposition, using finer edges along the surface, yields more realistic surfaces (Figure 3.2b).

For physically based simulations, in addition to problems associated with sharp corners, coarse boundaries cause errors in the numerical computation. The ray tracing method of computing fluxes, for example, assumes a constant flux of material over an edge. The contribution to each of the other elements due to scattering is computed from this constant flux at the center point of the edge. However, if the edge element is too large, the contribution from the near end of the edge will be much different from the contribution from the far end of the element. Thus, the approximation of constant flux will lead to large errors in the solution. Thus, edges need to be adaptively made smaller.

Several criteria can be used to determine the appropriate surface edge length. Ideally, the criterion should be based on a rigorous error estimate. Considering all possible sources of error in a growth calculation, it is seen that error could be due to the flux calculations or the velocity calculations. However, error estimation for topography simulation is still an active area of research [6]. In this work, the user supplies the appropriate edge length. Small edges may be allocated in areas of sharp corners or at the surfaces of different materials, etc.

Once the appropriate edge length is determined, Algorithm 3.1 is used to discretize the geometry. Note that the algorithm uses the geometry hierarchy to visit each edge (lines 1-3). If the edge length is too large, then the edge is split into smaller pieces (lines 4-5). The split procedure first searches for this edge in every boundary of the geometry (line 7-8). It then merely replaces the edge with new smaller edges (line 9). Note that each edge is visited only once to test for largeness. However, the split procedure in searching for all boundaries that contain the edge makes the algorithm $O(N^2)$ where N is the number of edges. If a double hierarchy with parental information is maintained, then the second search procedure is eliminated and the algorithm will be $O(N)$.

Algorithm 3.1 Discretize the Geometry

procedure GeometryDiscretize
Input: Geometry, G, and maximum edge length, l_{max}
Output: New Geometry with fine edges
begin
1. **for** each region $r \in G$ **do**
2.   **for** each boundary b of r **do**
3.   **for** each edge e of $\in G$ **do**
4.   **if** e length $> l_{max}$ **then**
5.   SplitEdge(e, l_{max});
end

procedure SplitEdge*(e, l_{max})*
begin
6. $R :=$ new edges along e smaller than l_{max}
7. **for** each boundary $b \in G$ **do**
8. **if** $e \subset b$ **then**
9. remove e from boundary list and replace with R
end

Once the geometry has been appropriately discretized, then the top layer of exposed points is extracted using the same algorithm discussed in the previous chapter in conjunction with Boolean deposition.

If a deposition simulation is performed, a thin layer of material is deposited on the surface using the Boolean deposition algorithm presented in the previous chapter. This layer is grown using SPEEDIE calculations. This step assumes that the initial layer is deposited extremely fast and that material-dependent sticking rates are negligible. The velocities from SPEEDIE are then used to simply change the coordinates of the surface points. Figure 3.3 shows the use of this approach for deposition and etch back of an oxide spacer layer.

3.2.1 Delooping

The deposition algorithm above can produce boundaries that loop over themselves as illustrated in Figure 3.4. Self looping structures arise whenever new material is deposited or grown at concave corners. The problem arises due to “directly perpendicular to the surface” growth. Growth from a vertical wall can then easily crash into growth from a horizontal wall. These conditions occurs in physical simulations of oxidation as well. A related anomaly occurs due to one region crashing into another. This always occurs during etching simulation. As a material is completely etched away, the motion of a string of points representing the material that is being etched away will crash onto a buried material. One of the easiest ways to recover from such anomalies

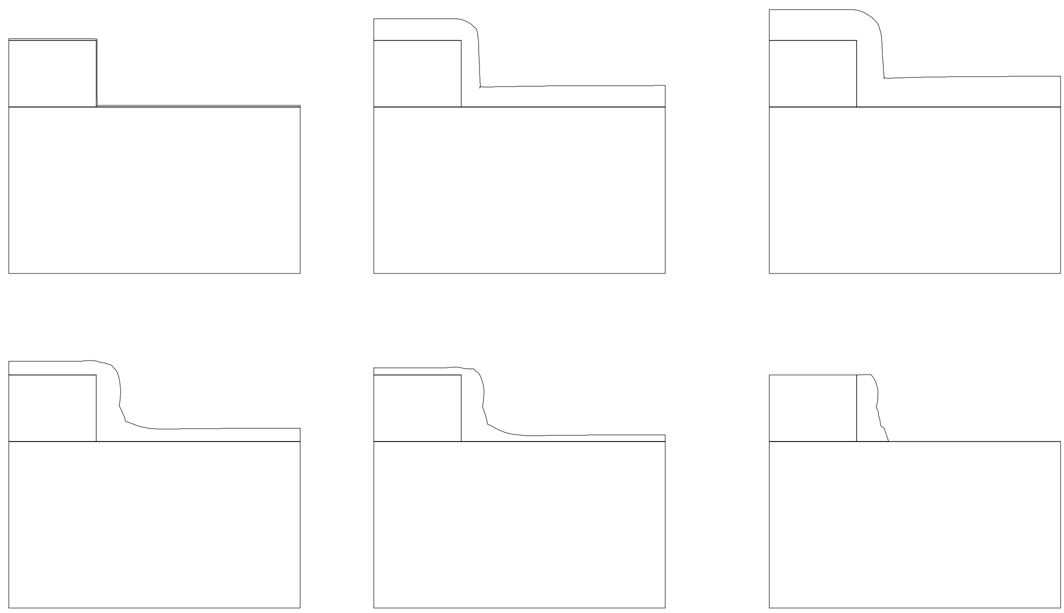


Figure 3.3: Use of SPEEDIE and *Forest* for deposition and etching of a 0.25 micron thick oxide spacer layer.

is to perform "delooping" whereby the loop is detected and excised as illustrated in Figure 3.5 and shown in Algorithm 3.2.

First, self loops of a boundary closing in on itself are detected and excised. Then, crashes between different boundaries are detected and excised. Self loops are detected by searching for intersections between non-consecutive edges along each boundary (lines 2-4 in Algorithm 3.2). This search for intersections can be carried out in $O(N \log(N))$ time for N edges [43]. If intersections are found, then, all the edges between the intersecting edges are necessarily part of the self-loop. These edges are deleted (lines 6-7). Moreover, the two intersecting edges are modified to end at their intersection point (line 8) as illustrated in Figure 3.5. This procedure is repeated until all the loops are excised. Note that the smallest loop between nearby edges is excised first. Detecting and excising loops between different boundaries is similar. For each boundary in the geometry, an intersection with another boundary is found. From that intersection to the next intersection, all the edges of one boundary are replaced with edges of the other boundary.

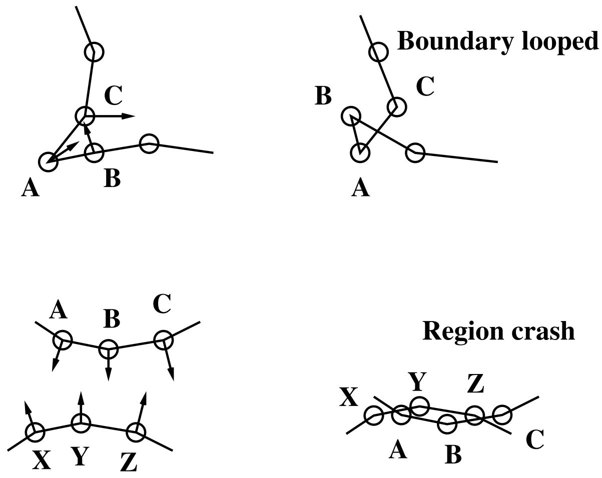


Figure 3.4: Illustration of loop formation during deposition simulation and of region crashing during etching simulation.

Algorithm 3.2 Delooping

procedure Deloop

begin

1. **for** each boundary **do**

2. E := edges of boundary;

3. find all pairs of non-consecutive edges in E that intersect

4. *for* each intersecting edge pair (e_i, e_j) **do**

6. **for** $k := i$ **until** j **do**

7. delete edge e_k from the boundary;

8. end point of e_i *and* e_j := to their intersection point.

end

Figure 3.6 shows a close-up view of a device during a time step in the spacer deposition simulation presented in the previous section. A small loop in the corner is visible. The loop is excised and the simulation is continued.

While delooping greatly increases the class of problems that can be solved, it has several drawbacks. One of the drawbacks is illustrated in Figure 3.7. The deloop procedure overfills materials in sharp trenches. Physically, one would expect a void

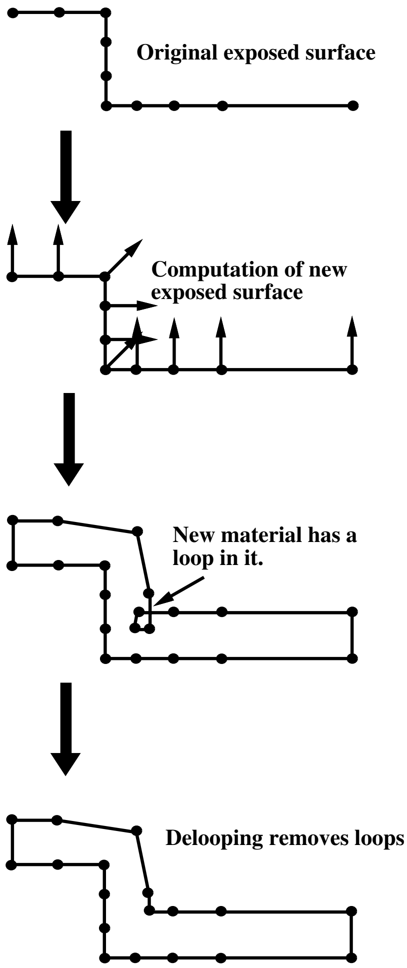


Figure 3.5: Illustration of loop formation during deposition simulation. The loop is detected and excised.

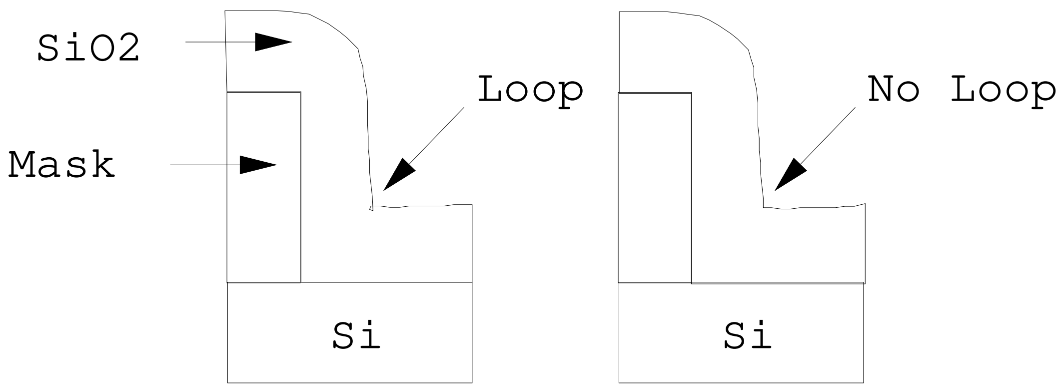


Figure 3.6: Boundary loop formation during a spacer deposition. The loop is detected and excised.

formation. The deloop procedure however, cannot distinguish between this case and that of loops at concave corners presented in Figure 3.6. Thus, one must be careful when using delooping during simulation.

More fundamentally, however, delooping is not a good procedure to use in the context of physically based simulation. Rigorous mathematical models derived from first principles are used to compute the deposition rates. Fitting parameters from actual experiments are used to calibrate the models. The growth rates depend non-linearly on the edge angles and their visibility. However, all this physical rigor is lost when delooping, as a result of *non-physically* geometry change, accommodates what is inherently a boundary representation and movement problem. All implementations of surface motion based on a string of points with velocities will suffer from this problem as it is impossible to compensate for the delooping operation *a priori* in the computation. As discussed in the previous chapter, a recently proposed method for overcoming this problem is based on an entirely new boundary free representation of the geometry [32]. The geometry is represented on a grid as distances from each grid point to each of the boundaries. The contour of this function $\psi = 0$ defines the boundaries. The boundary velocities computed by SPEEDIE ($\frac{\partial x}{\partial t}$ and $\frac{\partial y}{\partial t}$) are then used to compute the new distances to the boundary at each grid point by solving the

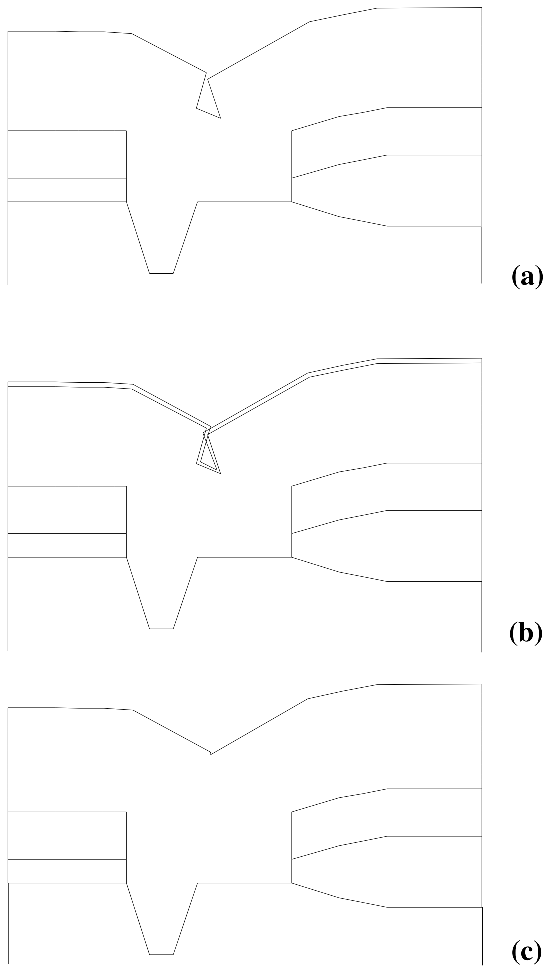


Figure 3.7: Delooping may fill in extra material when physically a void should be formed. (a) Initial geometry. (b) Loop forms after deposition. (c) Deloop removes the dip by filling with material.

following differential equation [25]:

$$\psi_t + \frac{\partial \psi}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial \psi}{\partial y}\frac{\partial y}{\partial t} = 0; \tag{3.1}$$

As long as the solution to the above equation is well behaved, the geometry will always be valid. Appropriate time integration insures that loops do not form at concave corners. Moreover, since delooping is not performed, void formation is accurately modeled.

3.3 Smoothing the Boundary

In this section, the inverse problem of making the geometry coarser is considered. After a topography simulation has been finished, the number of edges in the geometry is generally much larger than necessary. The fine granularity of geometry edges required for etching and deposition simulation is generally not needed or desirable for subsequent simulations. In particular, a complex geometry can cause excessive allocation of volume grid in the interior of the device, causing high computation times for dopant diffusion calculations. Thus, the geometry is generally smoothed after an etch or deposition simulation. Two different but related algorithms smooth the geometry and are discussed in this section. The first algorithm removes small edges from the geometry. The second one collapses collinear edges.

The motivation for removing small edges is illustrated in Figure 3.8. The boundary consists of two long edges separated by a smaller one that provides a horizontal jump. The shape of the geometry is not significantly altered if the small edge is removed and the long edges are made to end at the midpoint of the small edge. However, what constitutes a small edge? In this work, small edges are judged in relation to the original perimeter of the region. If the edges are smaller than a user specified percentage of the perimeter, then they are removed. However, edges that separate distinct regions are not removed as illustrated in Figure 3.8.

The complete algorithm for removing small edges is shown in Algorithm 3.3. The algorithm first finds each edge in the boundary and checks to see if it is smaller than

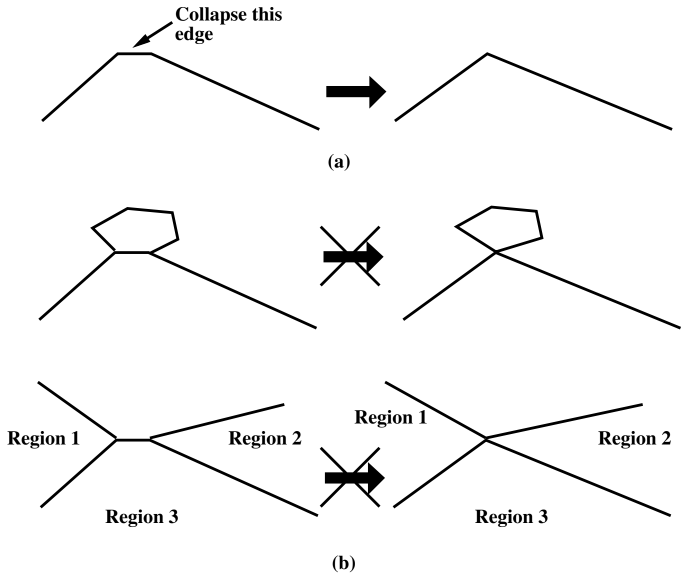


Figure 3.8: (a) Small edges in the geometry may be removed to produce a smoother boundary. (b) They are not removed if they separate different regions

a user specified fraction of the boundary perimeter (lines 1-5). If it is, then procedure **CanCollapse** is called to determine if collapsing the small edge will compromise the geometry. If it will not, then the edge is collapsed by calling the procedure **Collapse**.

The **CanCollapse** procedure performs two tests corresponding to the two cases shown in Figure 3.8. The first (lines 11-19) checks to see if collapsing will distort more than two regions. Such a distortion could inadvertently increase the area of one region at the expense of the other two. Thus, the procedure loops through all the boundaries to determine if the small edge is included in a different sequence of edges (lines 16-19). If the edge is indeed included in a different sequence, then it is not collapsed. The second test (lines 20-31) determines if collapsing a small edge will link two distinct regions. Prior to collapsing, the test loops through all the regions to find two different boundaries that contain the small edge (lines 26-29). If the boundaries are from distinct regions, then the edge is not collapsed (lines 30-31).

The collapse routine simply replaces the end points of the small edge with the midpoint for all edges in the geometry (lines 33-36). It then removes the small edge from every boundary in the geometry (line 37-39). Analysis of the algorithm indicates that it is $O(N^2)$ primarily because of searches in the procedures **CanCollapse** and **Collapse**. If these searches can be eliminated with a double hierarchy, the algorithm will execute in $O(N)$ time.

Algorithm 3.3 Remove Small Edges

```
begin
for each region r in Geometry G do
    for each boundary b of r do
        p := perimeter of b;
        for each edge e of b do
            if length of e < user specified fraction of p then
              e_n := edge after e;
              e_p := edge before e;
              if CanCollapse(e_p, e, e_n, r) then
                 Collapse(e_p, e, e_n);
end

procedure CanCollapse(e_p, e, e_n, r)
begin
collapse := TRUE;

(* check to see if collapsing will distort more than two regions *)
for each region r1 ∈ G do
    if r1 ≠ r then
      for each boundary b1 of r1 do
          for each edge e1 of b1 do
              if e1 = e then
                 if edge after e1 ≠ e_p then
                    collapse := FALSE;
                 if edge before e1 ≠ e_n then
                    collapse := FALSE;
```

```
    (* check to see if collapsing will link two distinct regions *)
preg := ∅;
nreg := ∅;
for each region r1 ∈ G do
    if r1 ≠ r then
        for each boundary b1 of r1 do
            for each edge e1 of b1 do
                if e1 = e_n then
                    nreg := r1;
                else if e1 = e_p then
                        preg := r1;
if (nreg ≠ preg) then
    collapse := FALSE;
return (collapse);
end

procedure Collapse(e)
begin
p := middle point of e;
for each edge e1 ∈ G do
    if e1 shares a point, sp, with e then
        replace sp with p;

for each boundary b1 ∈ G do
    if e ⊂ b1 then
        delete e from b1;
end
```

Collinear and nearly collinear edges are also removed as shown in Figure 3.9. The algorithm for removing collinear edges is very similar to the one for removing small

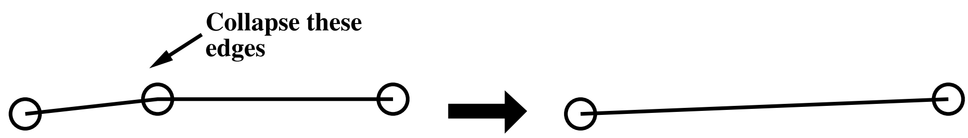


Figure 3.9: Collinear edges in the geometry may be removed to produce a smoother boundary.

edges. One of the collinear edges is deleted from all the boundaries and the end point of the other is replaced with the end point of the deleted edge. The same two tests in Algorithm 3.3 are performed to determine if collapsing will compromise the geometry. These algorithms are essential for full process simulation and are integral to the *Forest* geometry server.

3.4 Summary and Conclusions

This chapter focuses on physically based simulation of etching and deposition. The SPEEDIE program is used to compute material growth and etch rates. The computed velocities are used to move the boundary. Figure 3.10 shows an example simulation of back end processing in an MOS process. Physically based simulation are used for all etching and deposition steps. Planarization is performed using a geometric Boolean operation. First a spacer oxide layer is deposited followed by an etch back. A thick layer of doped glass is then deposited and planarized. Using a photo-resist mask, the glass is then etched back for metallization.

Several algorithms presented in this chapter are utilized in the example. Among these, the most important is delooping as loops always form at concave edges during a deposition simulation. Moreover, hidden layers are accurately represented so that etch simulations maintain accuracy. As soon an etch front reaches a hidden layer, the material parameters of the exposed strings are changed and the etch rates change correspondingly. Thus, the geometry representation and movement algorithm capabilities of the *Forest* geometry server greatly enhance the utility of a topography program like SPEEDIE to simulate several steps in a multi-step simulation.

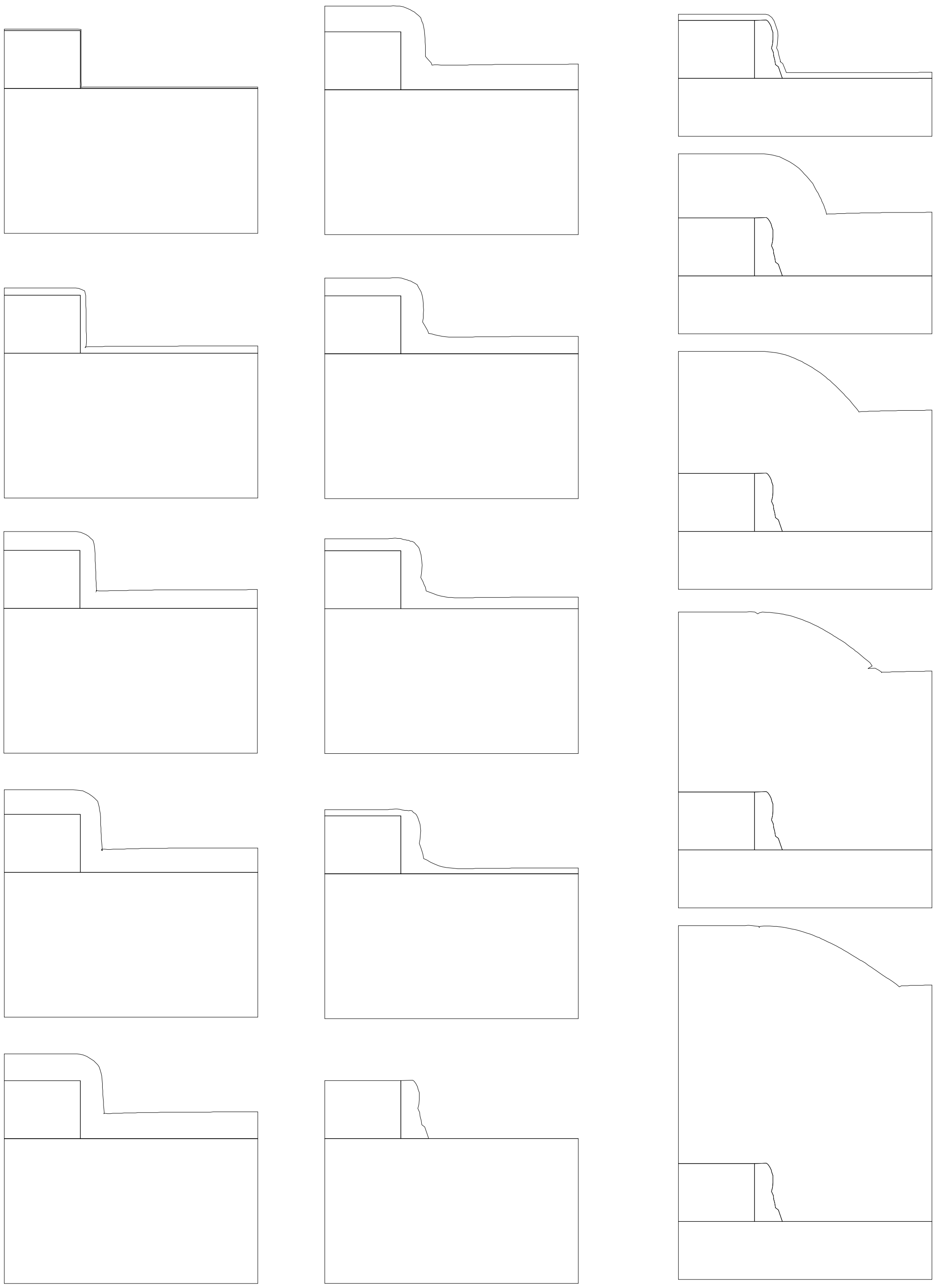


Figure 3.10: Example use of geometry motion algorithms for simulation of back end processing. SPEEDIE was used for etching and deposition simulations.

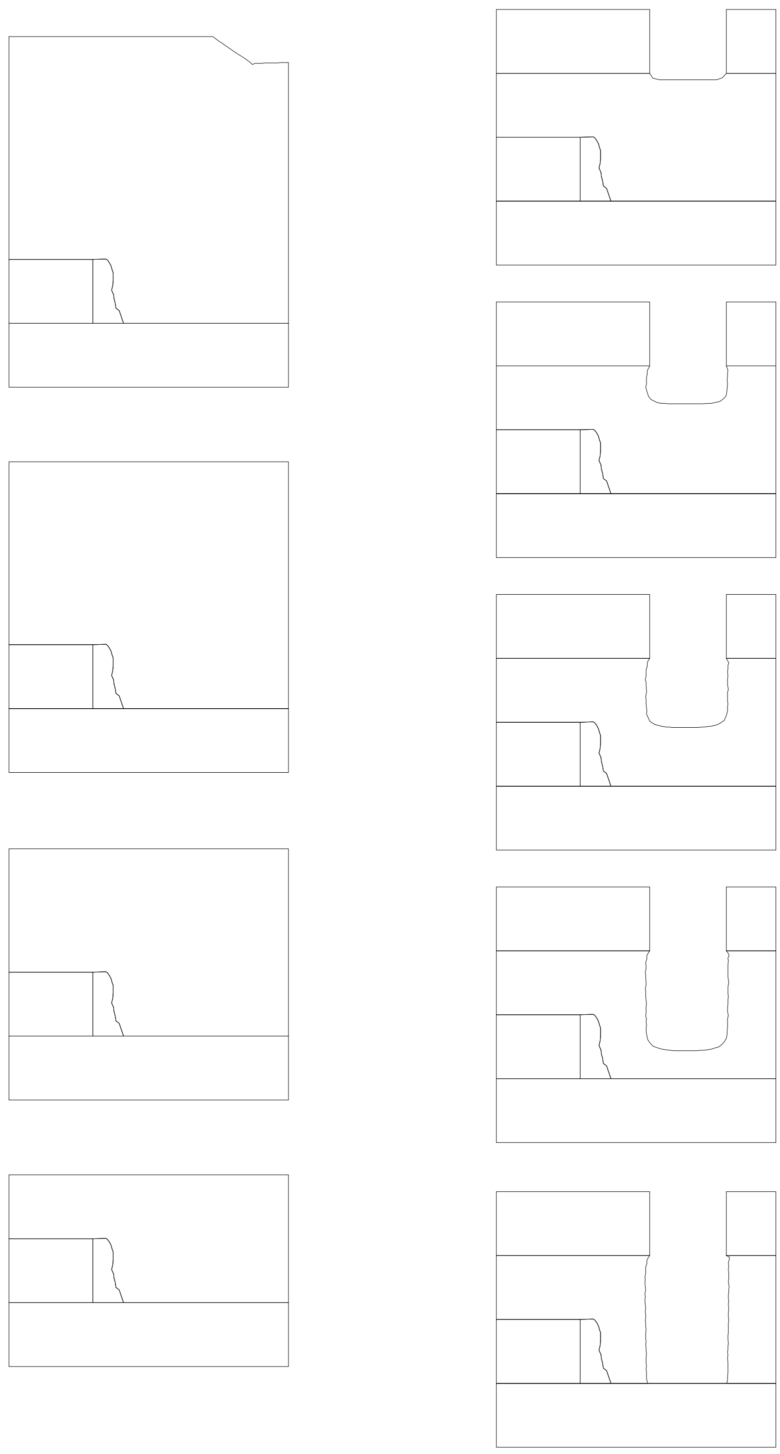


Figure 3.11: Example use of geometry motion algorithms for simulation of back end processing (continued)

Chapter 4

Grid Techniques for Thermal Simulation

4.1 Introduction

The goal of many thermal simulation programs is to numerically solve a set of partial differential equations that describe the physics of the domain of interest. More than one equation may be simultaneously defined and the equations may be non-linear. The equations may involve both spatial and temporal derivatives. The diffusion equations employed in SUPREM IV, for example, solve for diffusion of interstitials and vacancies together with the diffusion of dopants as delineated below [30]:

$$\frac{\partial C_I}{\partial t} = \nabla \cdot (D_I \nabla C_I) + \nabla \cdot (D_I C_I \nabla \psi) - K(C_I C_V - C_I^* C_V^*) \tag{4.1}$$

$$\frac{\partial C_V}{\partial t} = \nabla \cdot (D_V \nabla C_V) + \nabla \cdot (D_V C_V \nabla \psi) - K(C_I C_V - C_I^* C_V^*) \tag{4.2}$$

$$\frac{\partial C}{\partial t} = \nabla \cdot (D(C) \nabla C) \pm \nabla \cdot (D(C) C_A \nabla \psi) \tag{4.3}$$

where C_I, C_V, C, and C_A are interstitial, vacancy, total dopant and active dopant concentrations respectively, ψ is the electrical potential, D_I, D_V and D are the interstitial, vacancy and dopant diffusivities, and K is the recombination constant.

These partial differential equations are defined over a continuum. In order to solve the equations numerically, however, the equations must first be transformed into a set of equations that are defined over a discrete domain that is representative of the continuum. Thus, the equations are solved only at a finite number of points in the continuum. The equations must also be transformed into a set of algebraic equations that can be solved numerically. The finite element and control volume (finite difference) methods are commonly used in process simulation as equation discretization/solution techniques.

The continuum domain is transformed into a discretized domain with discrete elements whose union is the continuum by a grid generation program. The vertices of the elements define points in the domain over which solutions to the partial differential equations will be obtained. This chapter discusses algorithms for automatic grid generation. The dopant diffusion equations are solved using a control volume approach while the oxidation equations are solved using the finite element method. The requirements of the grid arising from the two discretization schemes are discussed in Section 4.2. Various grid generation techniques are discussed in Section 4.3. The grid generation algorithm in the *Forest* program that generates grids using a quadtree technique is discussed in Section 4.4. Grid adaptation is considered next in Section 4.5. Finally, grid movement to support oxidation simulation is discussed in Section 4.6.

4.2 Grid Requirements

Intuitively, it is seen that grid points must be allocated in areas where physical quantities vary. Consider the case of an p-n junction. The concentration of the n-type dopant goes from $10^{18}\,cm^{-3}$ on one side and falls off exponentially to less than $10^{8}\,cm^{-3}$ on the other. If only one point were allocated at the edge of the space charge region on the n-side and the other at the edge of the space charge region on the p-side, then the approximation to the concentration gradient of the n-type dopant based on only those two points will be very coarse. Since the gradient determines the dopant diffusion rate, a coarse approximation will lead to greater errors in the solution. Thus, fine grids should be allocated in areas of large gradients in the solution. On the other hand, the solution procedure solves a set of simultaneous equations with the solutions at the

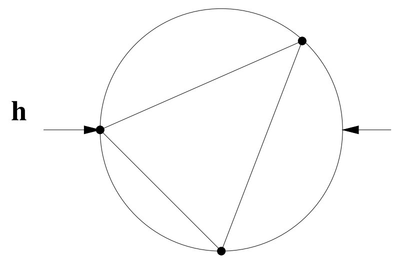


Figure 4.1: The error in the solution depends on the diameter of the circle circumscribing the largest element in the mesh.

grid points as the unknowns. The larger the number of grid points, the greater the matrix inversion time. Thus, the number of elements in the grid must be minimized.

Other requirements on the grid are also imposed by the solution techniques. Since both finite element and finite different methods are used in process simulation, the grid requirements from both are considered. For finite element methods, two different criterion are useful. As shown in the Figure 4.1, the error in a solution is proportional to h, the diameter of the smallest circle that circumscribes the largest element of the grid [26]:

$$e \leq ch^{\alpha}C \tag{4.4}$$

where e is the error difference between the compute solution and the actual solution, c is a constant, h is the grid parameter, and C is the solution to an elliptic equation. For e to be small, h must also be small. Thus, the element sizes should be small. Moreover, since h depends on the circumscribing circle, the elements should be close to equilateral. Thus, slender elements with any large sides are likely to contribute as much error as a large element.

The second criterion applicable to finite element solutions is related to matrix inversion and solution stability. As shown in [20], for non-uniform grids, the condition number, $C(K)$, is given by:

$$C(K) = c(h_{max}/h_{min})^{2m-1}N_e^{2m/n} \tag{4.5}$$

where c is a constant, h_{max} is the size of the largest element in the grid, h_{min} is the size of the smallest element, $2m$ is the order of the differential equation, N_e is the number of elements and n is the dimension. For the smallest condition number, the ratio h_{max}/h_{min} should be minimized. Thus, the elements should be uniformly sized. Note also that the condition number increases as the number of elements increases. Thus, the fewer the elements, the better.

For finite volume discretizations, Pinto [41] analyzes the grid requirements. Using a centered difference approximation, the gradient can be expressed as:

$$\left.\frac{dC}{dx}\right|_{x_i} = \frac{C(x_{i+1}) - C(x_{i-1})}{\Delta x_{i+1} + \Delta x_i} - \frac{\Delta x_{i+1} - \Delta x_i}{2}\frac{d^2C}{dx^2} + \frac{(\Delta x_{i+1})^2 - \Delta x_{i+1}\Delta x_i + (\Delta x_i)^2}{6}\frac{d^3C}{dx^3} \tag{4.6}$$

In order to reduce the truncation error from disregarding the last two terms of the above equation, two conditions must be met. If $\Delta x_{i+1} - \Delta x_i$ is small by using elements of nearly equal size, the second term is minimized. The last term can be made small by making Δx as small as possible. Thus, the grids must be uniformly distributed with fine spacing.

The element shape criteria for finite volume computations arise from using Green's Theorem (Figure 4.2) to integrate the differential equations:

$$\nabla \cdot \nabla C(x, y) = f(x, y) \tag{4.7}$$

$$\int\int_{A_i} \nabla \cdot \nabla C dx\, dy = \int_{L_i} \nabla C \cdot \mathbf{n} dl = \int\int_{A_i} f\, dx\, dy \tag{4.8}$$

where C is a dopant concentration, f is the forcing term, A_i is the area associated with grid point i defined by the contour L_i and $\mathbf{n}$ is the outward unit vector normal to L_i.

Partitioning of an element is done by perpendicular bisectors as shown in Figure 4.2. The use of perpendicular bisectors guarantees that that any area that is closer to grid point i is associated with it - the Voronoi condition. (The dual grid of the perpendicular bisectors is called a Voronoi Diagram.) Integration of the forcing term in Equation 4.8 is performed by assuming constant variation equal to the nodal value over each Voronoi area. Thus, fine grid should be allocated in areas of high variation of the forcing term. Moreover, the Voronoi area associated with a grid point will be inside the triangles surrounding it *if and only if* the triangles are acute. For

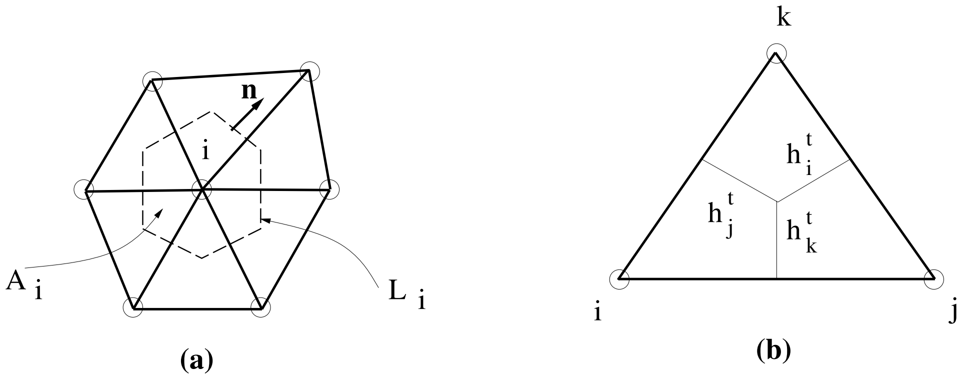


Figure 4.2: (a) Application of Green's Theorem about grid point i. (b) Partitioning of an element is done by perpendicular bisectors.

obtuse triangles, the area extends outside the triangle leading to double counting of some of the area. Additionally, the perpendicular bisector of the side opposite the obtuse angle will point outside the triangle. Thus, when the contour integral is computed, a negative term is added leading to convergence difficulties [41]. Thus, obtuse angles should be avoided.

In summary, the following grid requirements must be met:

- Grids should be allocated to accurately resolve physical quantities of interest.
- Element sizes should be as uniform as possible.
- The number of elements should be as small as possible.
- Slender elements with small or obtuse angles should be avoided.

Note that some of these requirements are conflicting. For example, requiring a uniform fine grid to resolve physical quantities violates the criteria for minimizing the number of elements. Automatic grid generation algorithms strike different compromises to achieve an optimal strategy for a given application domain.

4.3 Grid Generation Techniques

A number of different algorithms have been employed to generate grids over both two and three dimensions. The algorithms are generally classified into three different classes: (1) those that generate structured grids, (2) those that generate unstructured grids and (3) hybrid algorithms that combine the two and form a third class of grids called semi-structured grids.

Structured grids are grids that are in some way related to the regular x-y coordinate system. The intersections of points of the constant x and constant y lines form the grid. For simple rectangular shaped geometries, structured grids are trivial to generate. The original grid generation algorithms in SUPREM-IV and PISCES employed structured grids. The spacing of the constant x or constant y lines can be varied as needed to allocate dense grid points in regions of interest. Thus, the solution can be accurately represented. Moreover, the elements of the grid are all right triangles.

If the geometry is more complicated, then a regular box of height 1 and width 1 in the first quadrant of a Cartesian coordinate system is transformed to the shape of the geometry. The points in the Cartesian coordinate system box are mapped onto points on the geometry. Figure 4.3 shows a simple example of a transformation from a square in the Cartesian coordinate system to a concentric set of rings in the cylindrical coordinate system. For more complicated geometries, differential equations are solved to determine the transformation [17].

The greatest advantage of structured grids lies in the fact that the grid is regular. Thus, each grid point has well defined neighbors in all coordinate systems. Thus, connectivity information need not be stored saving memory and processing time. Moreover, since the grid points are simply connected, the matrix bandwidth is guaranteed to be limited. In several fields including computational fluid dynamics, structured grid generation is the technique of choice.

However, one important disadvantage of this approach make it unsuitable for process simulation. Consider the case of geometry deformation and movement in conjunction with oxidation simulation. If the geometry moves, then the transformation changes and the grid needs to be recomputed. Since the transformation is unique for

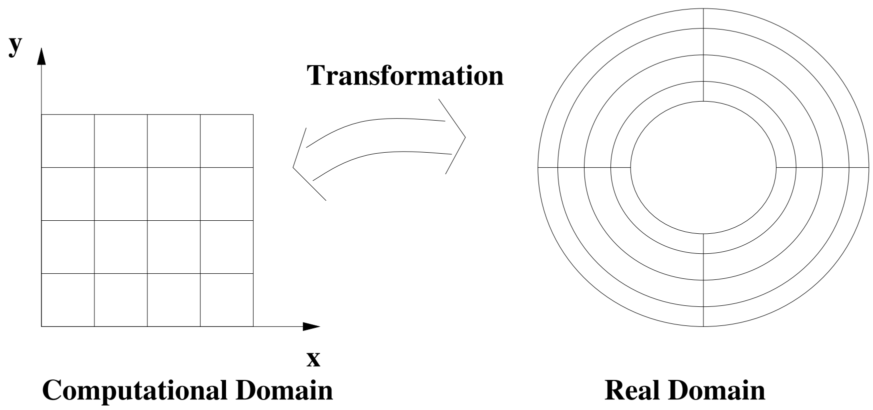


Figure 4.3: Structure grids are generated by transforming the real domain into a computational domain.

each geometry, changing a geometry involves generation of new points. Solution from the old points must now be interpolated onto the new points. Problems associated with dose conservation, interpolation error, etc., then need to addressed.

Unstructured grids are at the other end of the spectrum. Triangles are generated without any regard to a regular Cartesian system. Two sets of different approaches are popular. The first set is based on domain reduction[1]. A single triangle is "chopped off" by connecting three consecutive non-collinear points of the geometry as shown in Figure 4.4. The reduced geometry is then presented for a subsequent chop. This procedure continues until the domain is fully triangulated [5].

Elements generated using the domain reduction method generally need to be post-processed to improve their quality. Since the points of the geometry are used for chopping, it is conceivable that an element with an obtuse angle or with small angles may be chopped off. A popular technique for improving such elements [5] [15], called Laplacian smoothing, moves each point of the grid to the centroid of the polygon formed by all triangles that end at the point. The procedure, illustrated in Figure 4.5, decreases the number of triangles with small angles. However, it does not guarantee a triangulation without obtuse angles.

[1]Several researchers [8] have reported on a technique called "paving." The paving algorithm generates quadrilateral or triangular grids using a process similar to laying tiles on a floor. Fundamentally, however, paving can also be considered a domain reduction technique.

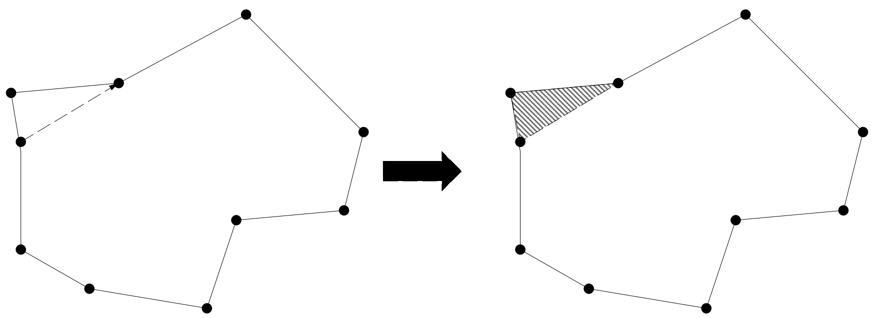


Figure 4.4: Illustration of domain reduction for triangulation.

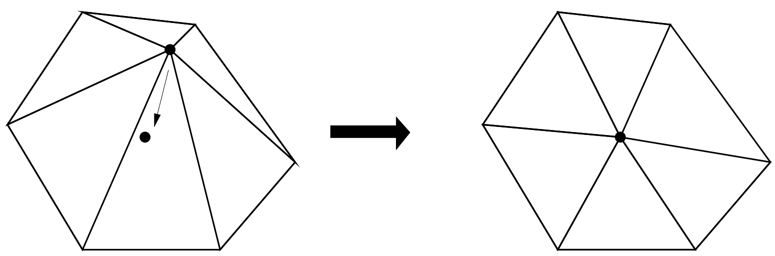


Figure 4.5: Laplacian smoothing reduces the number of triangles with small angles by moving each point in the grid to the centroid of the polygon formed by all triangles at the point.

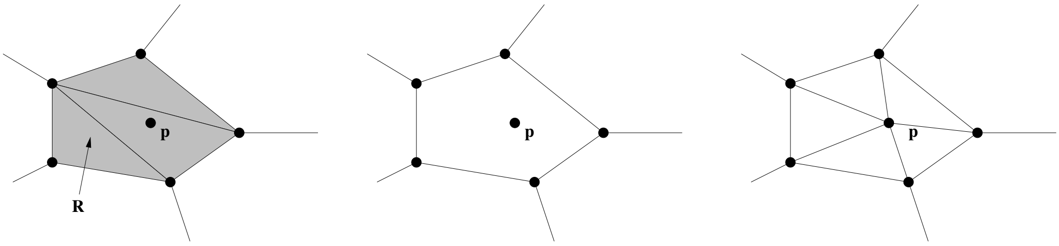


Figure 4.6: Illustration of Delaunay triangulation.

The second unstructured grid generation approach is called Delaunay triangulation. Given a set of points in the plane, a Delaunay triangulation forms triangles that connect these points. Several algorithms exist for generating Delaunay triangulations [11]. Only one of them is reviewed herein in Algorithm 4.1 and illustrated in Figure 4.6.

Given a point set, P, a "super triangle" is drawn enclosing all the points (line 1 in Algorithm 4.1). Each point p is then introduced into the triangulation and the connectivity of the triangles is changed as follows. All elements whose circumcircle contains p are found and deleted. This creates a subregion, R, surrounding p (lines 2-7). Lines to p are drawn from each of the vertex points of the subregion (line 8). The connectivity is then changed to satisfy the Delaunay criteria by a technique called diagonal flipping as illustrated in Figure 4.7 (line 9). The diagonal flip recursively removes all obtuse angles in a triangulation by choosing the best diagonal in a quadrilateral defined by two triangles. The next point is then introduced into the current Delaunay triangulation and the procedure is repeated. Finally, elements that are not in the convex hull of P are discarded (line 10). Judicious choice of data structures and searching algorithms reduce the algorithm execution time to $O(N \log(N))$ where N is the number of points.

Algorithm 4.1 Delaunay Triangulation

procedure Delaunay
Input: set of points P in the plane
Output: triangulation of the points
begin

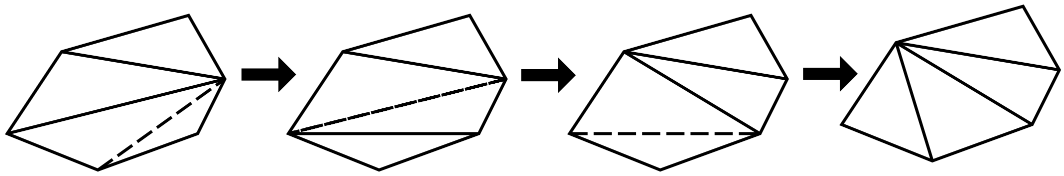


Figure 4.7: The best diagonal of the quadrilateral containing two triangles is chosen iteratively to generate a Delaunay triangulation.

1. $T :=$ super triangle enclosing P
2. **for** each $p \in P$ **do**
3.   insert a point p into T;
4.   **for** $e \in T$ **do**
5.   **if** circumcircle of e contains p
6.   add vertices of e to form a subregion R
7.   delete e;

8.   Connect each point in R to p forming new elements in T
9.   diagonal flip elements in T to enforce Delaunay condition

10. Remove elements not in convex hull of P
end

In most cases, one needs to triangulate a set of polygons and not simply a set of points. Delaunay triangulation then proceeds as follows. The N vertices of the polygon are used for an initial triangulation using Algorithm 4.1. Next, the center of the largest circumcircle in the triangulation is inserted. The procedure outlined above is applied to the new point. Insertion of new points continues until all the elements are of the desired size.

Delaunay triangulations have many desirable properties. The triangles are all acute or right triangles in the interior. However, along the boundary, one cannot perform a diagonal flip and hence, the Delaunay condition must be relaxed. Triangulations from this relaxed Delaunay condition are often termed *constrained* Delaunay triangulations. Moreover, all the area around the adjoining Voronoi dual diagram is closest to the appropriate grid point. Thus, assumptions of constant variations of values on an element introduce less error in the final solution of a partial differential equation.

4.4 Tree Based Grid Generation

Tree based grid generation takes an intermediate approach between a fully structured grid and a fully unstructured grid. The basic idea behind tree based gridding is to reduce the general problem of triangulating a set of polygons to a simpler problem of triangulating a quadrilateral given a boundary constraint. Regularity is obtained in one sense as a basic unit cell (square) is duplicated throughout. The grid is unstructured in the sense that not all grid points are connected by orthogonal grid lines to other points in all directions. It guarantees a constrained Delaunay grid - all interior elements will be right or acute while those on the boundary may be obtuse.

Several authors have used this approach for different application domains [35] [4] [7]. The approach of Muller et. al. [35] generates a quadtree by drawing horizontal and vertical lines from each point of the geometry. These lines divide the geometry into a set of quadrilaterals that are further divided as needed and then triangulated. This approach, while ideal for device simulation, is not suitable for moving boundary process simulation. Since the quadtree depends on the points of the geometry, geometry change during motion will change the entire quadtree and the subsequent grid. The points of the new grid will not coincide with those of the old grid necessitating interpolation. Moreover, since the quadtree is generated based on the geometry points, a complicated geometry with many points describing the geometry will lead to excessively fine initial quadrilaterals and consequently to large number of grid points.

The *Forest* program generates grids using an approach similar to that of Bern et. al. [7]. The algorithm is as shown in Algorithm 4.2. It consists of five major steps.

Algorithm 4.2 Tree Based Triangular Grid Generation

procedure GenerateGrid
Input: Geometry
Output: Triangular Grid

begin

1. QuadDecimation(); *(* generate quadtree from geometry *)*

2. BalanceTree();

3. Warp();

4. Triangulate(); *(* triangulate quads using templates *)*

5. PostProcess(); *(* generate neighbor lists, etc. *)*

end

The geometry is first surrounded with a root quadrilateral which is recursively decimated to produce a terminated lines quadtree quadrilateral grid. The tree is then balanced so that there is a smooth variation in the sizes of neighboring elements. Quadrilaterals along the boundary are then *warped* to avoid elements of poor quality. The elements themselves are generated next using templates. Post processing of the elements generates neighbor information and finally, solution values are mapped onto the grid points. Each of these sub-steps is discussed in greater detail in the following subsections.

4.4.1 Generating the Quadtree

The algorithm for generating the quadtree is as shown in Algorithm 4.3. The root quadrilateral is first determined from the largest bounding rectangular box of the geometry. Since squares are used for the quadtree decimation procedure, the smallest square that encloses the bounding box is used. The root square is then recursively decimated using the procedure **QuadDecimation** as illustrated in Figure 4.8.

Each parent square is divided into four smaller squares (lines 1-3 in Algorithm 4.3. Each of the "child" squares, $q[i]$, is intersected with each of the regions of the geometry (lines 4-5) using the region intersection algorithm described in Chapter 2. The intersection, I, is stored as attributes of the square. If the square lies outside the geometry, then it is not divided any further. Squares within the geometry are recursively divided as required using the same procedure (lines 6-14).

The recursive decimation procedure builds a tree data structure as shown in Figure 4.8. Each quadrilateral holds a pointer to its parent and pointers to all its children.

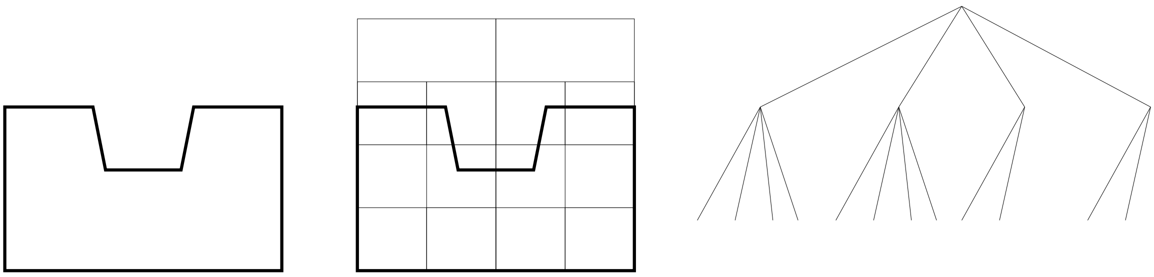


Figure 4.8: Decimation of the geometry into a quadtree quadrilateral grid simplifies the geometry and generates a tree data structure.

Thus, each quadrilateral object on the tree is equivalent to every other quadrilateral object on the tree. This tree data structure is exploited in a variety of ways as described below.

The level of decimation is set by the user. Prior knowledge of areas of high solution variation can be used to make the grids finer. Typically, grid sizes of at least as small as a few microns are desirable. Thus, the active region of the micron long transistor will have approximately 100 elements in the length direction. Each quadrilateral size is compared to the user set threshold and divided if it is bigger (lines 8-9).

For quadrilaterals along the boundary, characteristics of the geometry determine the decimation level. As each quadrilateral is intersected with each of the regions of the geometry, the intersection points are stored as attributes at the quadrilaterals. Note that the intersection is computed with each region of the geometry. Since the same intersection points may be found for intersections with different regions, the attribution mechanism searches through the current list and does not duplicate any entries. Quadrilaterals that enclose more than one point of the geometry are decimated (lines 11-12) since it is extremely difficult to triangulate such quadrilaterals as shown in Figure 4.9. Thus, the number of points used to define a boundary determines the depth of the quadtree. Moreover, if there are more than two intersections of the edges of the quadrilateral with the geometry, the rules of triangulation also require decimation as shown in Figure 4.9b (lines 13-14). Thus, the number of regions in the geometry also determines the depth of the quadtree.

Algorithm 4.3 Generate Quad Tree

```
procedure QuadDecimation
Input: Parent quadrilateral P and geometry G
Output: Sub-tree of quadrilaterals underneath
begin
    allocate memory for four new quadrilateral leaves;
    for i := 0 until i = 3 do
        form quadrilateral q[i] from P

        I := ∅;
        for each region r in  G do
            I := I + q[i] ∩ r; (* compute intersection *)

        if q[i] ⊂ G then
            (* determine if quad must be divided again *)
            if size of q[i] > threshold then
              QuadDecimation(q[i], G);
            if I ≠ ∅ then
              if I contains more than one vertex point then
                QuadDecimation(q[i], G);
              if I contains more than two edge intersections then
                QuadDecimation(q[i], G);
end
```

Analysis of algorithm complexity is somewhat difficult. Consider separating N randomly placed vertices of a geometry into a quadtree structure with one vertex per leaf quadrilateral. The first decimation divides N into 4. The problem has been reduced to four sub-problems of smaller size and the algorithm is of $O(N \log(N))$ complexity. Algorithm 4.3 is similar except that the complexity of the quadrilateral-geometry intersection (I) is continually reduced as the decimation proceeds.

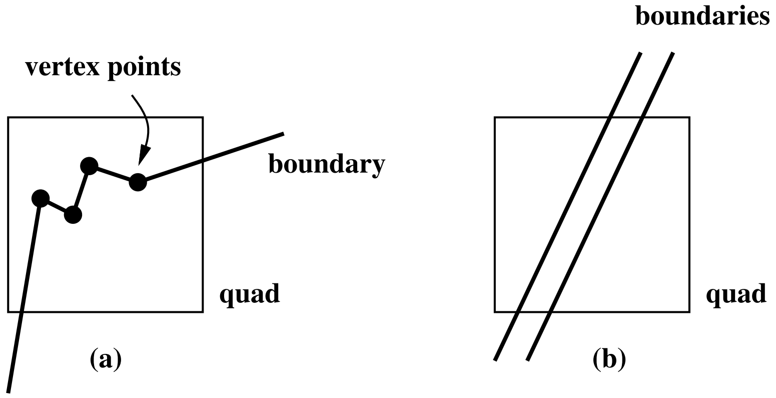


Figure 4.9: Boundary complexity determines the level of decimation as the quadrilateral is not simple enough to triangulate. (a) Quadrilaterals with more than one interior point are divided. (b) Quadrilaterals with more than two intersection points are divided.

4.4.2 Balancing the Tree

The quadtree decimation procedure will in general produce a tree that is unbalanced. Quadrilaterals of varying sizes will be next to one another. Two types of imbalance are possible as shown in Figure 4.10. Quadrilaterals of different sizes whose edges coincide could be next to each other. Triangulation of such quadrilaterals is difficult. Even if appropriate templates can be defined, triangulations of such quadrilaterals will lead to poor quality triangles. Thus, the larger quadrilaterals are subdivided. Quadrilaterals that only share a vertex may also be next to each other. Triangulation of such quadrilaterals will not produce poor quality triangles. However, there will be a large variation in element sizes. As shown in Section 4.2, a smooth variation in the element size is highly desirable. Thus, the larger quadrilateral should be divided. The quadtree, therefore, needs to be balanced. However, the tree is not *fully* balanced. Otherwise, all the elements in the grid will be equally sized leading to potentially large grids. One level size variation is used to gradually increase the grid density. Triangulation of quadrilaterals from such a tree yields elements of high quality while at the same time allowing gradual variation in the element size. Thus, large numbers of elements can be allocated locally in different parts of the device geometry.

The essence of the tree balance algorithm is to compare the size (or depth in the

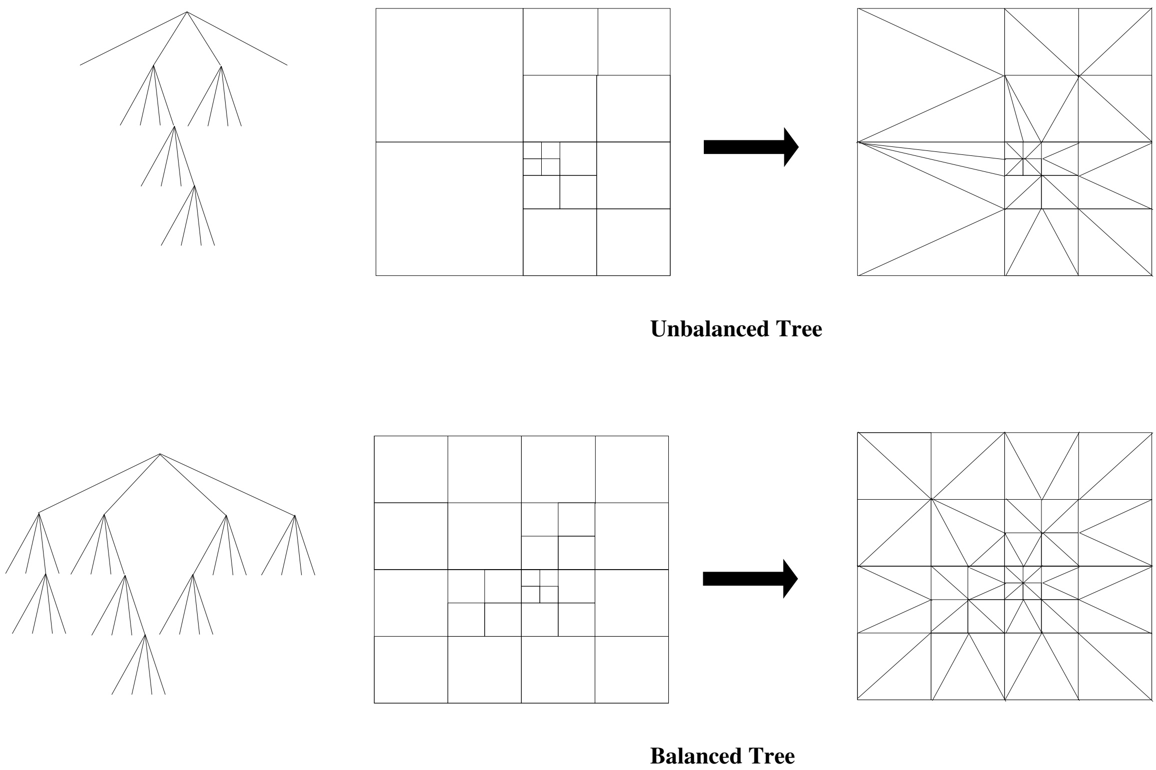


Figure 4.10: Tree balancing insures that there is a smooth variation in element size and preserves element quality.

tree) of two neighboring quadrilaterals and to decimate the larger quadrilateral. The quadtree data structure can be used to greatly simplify the search process needed to find neighbors. In the *Forest* grid server, a different approach is used. Each corner point holds pointers to all quadrilaterals that end on it. Thus, the problem of searching for neighbors is greatly simplified. The procedure is as shown in Algorithm 4.4.

At each corner point, pairs of leaf quadrilaterals are tested for size variation. Each "large" quadrilateral is first marked for division - a pointer to that quadrilateral is stored in an array (lines 1-7). After all such "large" quadrilaterals have been found, the quadrilaterals are then divided (lines 8-9). The procedure is repeated until no quadrilaterals need to be divided (lines 10-11). Each loop executes in $O(M^2N)$ where N is the number of corner points and M is the largest number of quadrilaterals at a corner point. However, the number of times that the loop executes could be large resulting in a super-linear execution time. To ameliorate the situation, corner points are marked as the leaf quadrilaterals are tested. Only the points that have new leaf quadrilaterals are tested at the next loop. Thus, the number of points tested at the next iteration is far less than N. This reduction in N continues until the tree is balanced.

Algorithm 4.4 procedure *BalanceTree*

begin
1. *done := 0*
2. **while** *done = 0* **do**
3. **for** each corner point **do**
4. **if** new quadrilaterals have been added at point **then**
5. **for** each pair of unmarked leaf quadrilaterals at point **do**
6. **if** there is more than one level difference between the pair **then**
7. mark larger leaf quadrilateral for division

8. **for** each marked quadrilateral **do**
9. divide quadrilateral by calling **QuadDecimation**;

10. **if** none were marked for division **then**
11. *done := 1;*
end

4.4.3 Warping Quadrilaterals

After a balanced tree has been obtained, a procedure called warping is performed to improve the quality of the triangulation along the boundary. Consider the case of making triangles from a boundary quadrilateral as shown in Figure 4.11a. The triangles must include the corner points of the quadrilateral and the intersection points of the geometry with the quadrilateral's edges. Figure 4.11b shows the case where the triangles are of very high quality if the intersection points are nearly centered on the quadrilateral edges. If the intersection points are very near the corner points however, the triangles will be of poor quality with small angles (Figure 4.11c and d) or the triangles will be extremely small (Figure 4.11e and f). A similar situation arises when an interior vertex point of the geometry is close to the corner point as shown in Figure 4.11g and h. A slightly different situation arises if the vertex point of the geometry is close to an edge of the geometry as shown in Figure 4.11i and j. The triangles from the quadrilateral, in addition to having small angles will also have a large obtuse angle. Baehmann et. al. [4] removed such triangles using a post processing approach. They detected small and poorly shaped triangles and deleted them.

In the *Forest* program, warping [7] *prevents* the initial formation of all poor triangles. The warping procedure essentially moves the corner points of the quadrilateral onto its intersection points or to the interior vertex point as shown in Figure 4.12. For the case of a vertex point being close to an edge, the edge itself is warped onto the vertex point. The area lost by a quadrilateral is gained by its neighbors.

The criteria for warping are developed from the expression for the quality, Q, of a triangle [5]:

$$Q = \frac{4\sqrt{3}A}{\sum s_i^2} \tag{4.9}$$

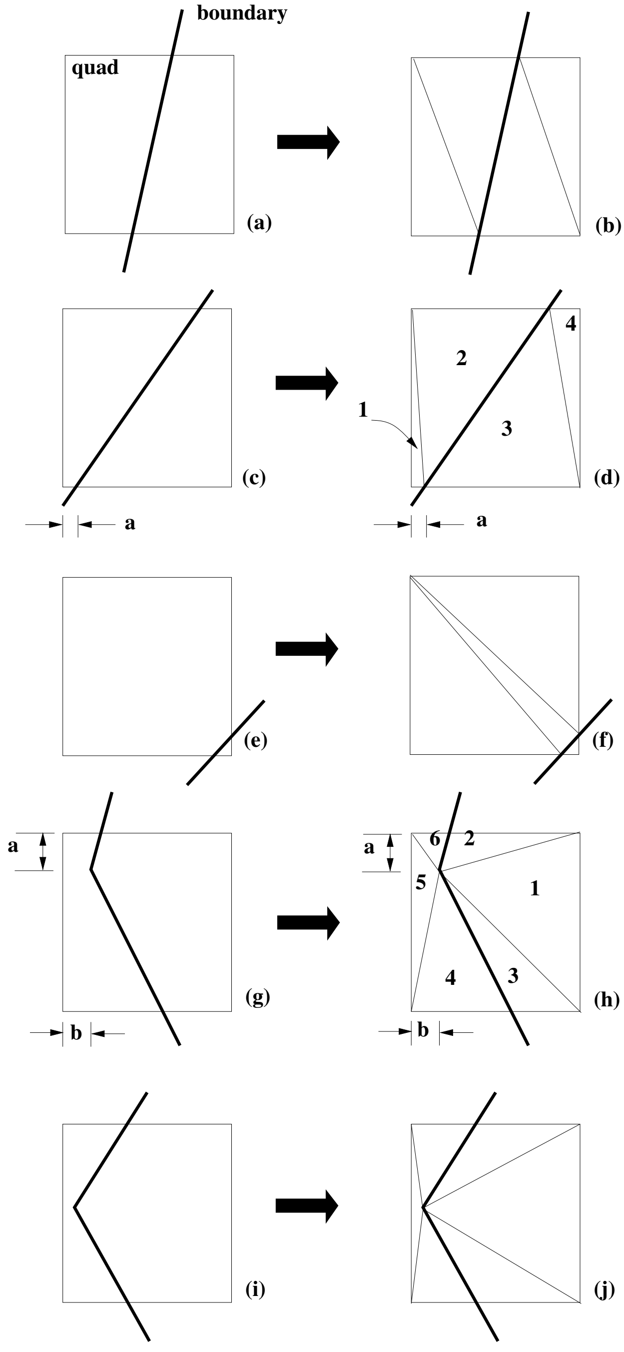


Figure 4.11: Triangles along the boundary may be of poor quality as the boundary may pass through a quadrilateral at any point.

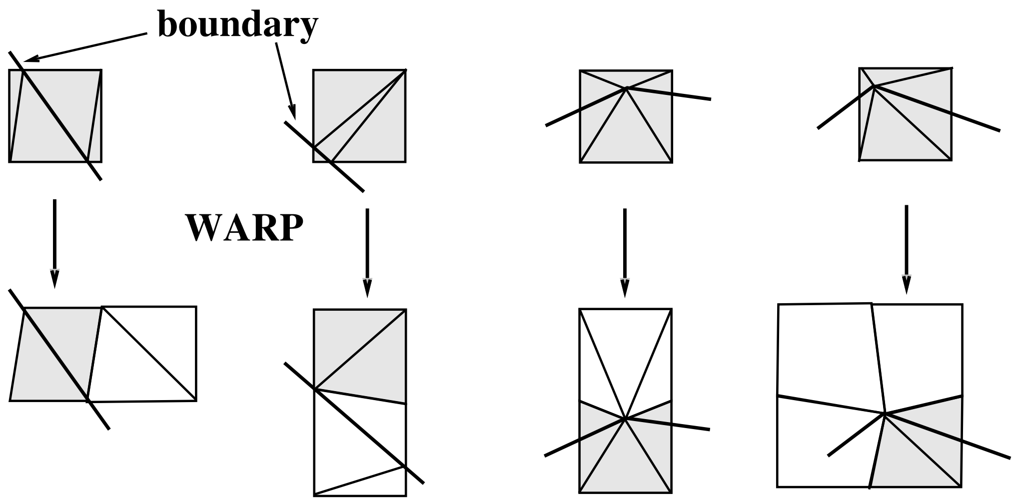


Figure 4.12: Warping is used to improve the quality of triangles along the boundary.

where A is the area of the triangle and s_i is the length of side i. Q is equal to 1 for an equilateral triangle and approaches zero for triangles with small angles. Thus, it is desirable to obtain a triangulation with triangles of as high a quality as possible. Traditionally triangulations with triangles of $Q > 0.5$ have been considered "optimal."

Consider the case of triangle 1 in Figure 4.11d. Applying the above equation for a quadrilateral of side length = 1, Q reduces to:

$$Q_1 = \frac{2\sqrt{3}a}{2(1+a^2)} \tag{4.10}$$

where a is the distance of the intersection point from the nearest corner. For Q_1 to be greater than 0.5, a must be greater than $\sqrt{3} - \sqrt{2} = 0.31$. Thus, if the intersection point is any closer than approximately one third of the quadrilateral edge length, the corner point should be warped onto the intersection point.

Analysis for triangle 5 in Figure 4.11h yields the following expression for Q:

$$Q_5 = \frac{\sqrt{3}a}{a^2 + b^2 - b + 1} \tag{4.11}$$

where a and b are as shown in the figure. Plotting this equation as a function of a and b as in Figure 4.13 shows that for Q to be larger than 0.5, a must be greater

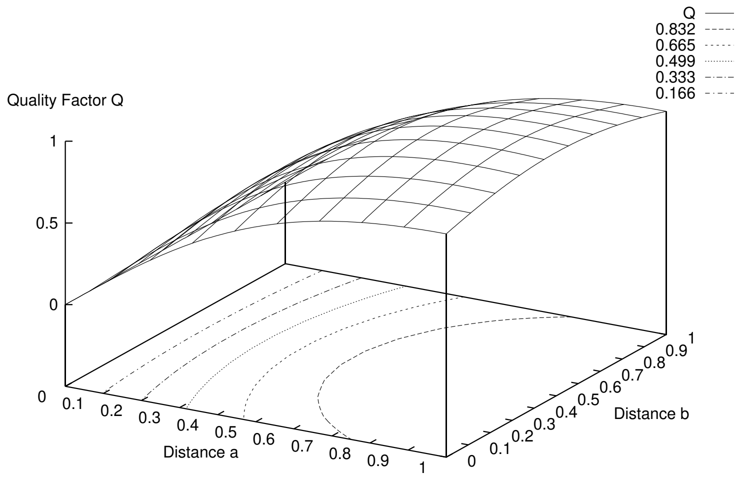


Figure 4.13: Quality factor of triangle 5 of Figure 4.11g as a function of distances a and b of the vertex point. Contour lines show that a must be greater than approximately 0.3 for $Q > 0.5$.

than approximately 0.3 irrespective of b. For triangle 1, a and b are interchanged. Hence, both a and b must be greater than 0.3. Similar analysis of other triangles in Figure 4.11 yields the same warp condition - if the vertex or intersection point is closer than one third the quadrilateral edge length, the corner point should be warped onto the vertex or intersection point.

The warping algorithm shown in Algorithm 4.5 entails a straight forward traversal through all the leaf quadrilaterals, warping each one. Note that warping changes all quadrilaterals attached to the corner point. Thus, the intersection with the geometry that was computed during tree generation must be recomputed for all these quadrilaterals.

Algorithm 4.5 Warp Quadrilaterals

procedure Warp
begin
1. **for** each leaf quadrilateral q in tree T **do**
2.   **if** q contains a vertex point **then**

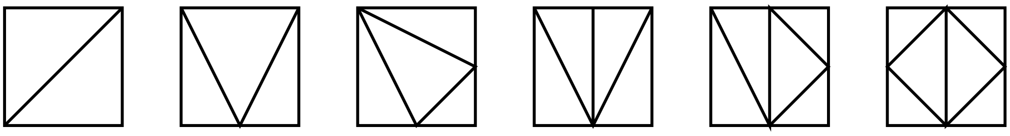


Figure 4.14: Templates used for generating triangles from interior quadrilaterals.

    if vertex point is close to a corner **then**
      warp corner point to vertex point*;*
      for each leaf quadrilateral attached to corner point **do**
          re-compute intersection with geometry*;*
  elseif q is intersected by the geometry **then**
        if intersection point is close to a corner **then**
          warp corner point to intersection point*;*
          for each leaf quadrilateral attached to corner point **do**
              re-compute intersection with geometry*;* **end**

4.4.4 Triangulation

Once the quadtree has been generated, balanced and warped, the leaf quadrilaterals are triangulated using templates. The triangulation algorithm thus consists of a large decision tree that matches a given quadrilateral with the appropriate template. Figure 4.14 shows the templates used for triangulating quadrilaterals that are not on a boundary. Figure 4.14 also lists the lowest quality factor Q of a triangle in that template. It is readily seen that all interior triangles are of high quality and that none of them are obtuse. Figure 4.15 shows the templates used for triangulating quadrilaterals on the boundary. All the triangles are once again of high quality. Elements that are within the geometry domain are kept while those that are not inside the geometry are freed and deleted. The region intersection routine from Chapter 2 is used again for this purpose.

Despite the fact that the tree should have been decimated sufficiently, on occasion, due to round off error or due to problems with the region intersection algorithm

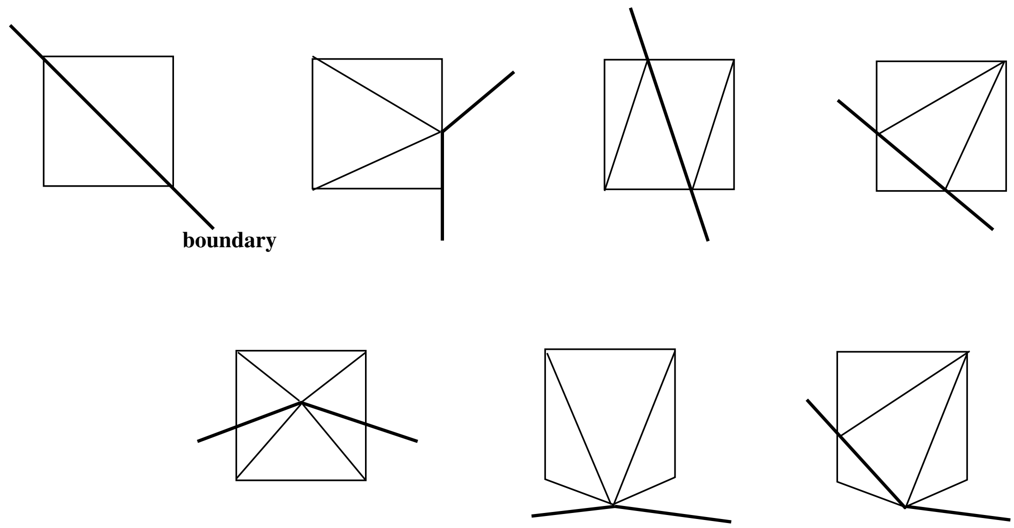


Figure 4.15: Templates used for generating triangles from quadrilaterals along boundaries. Dark lines represent geometry boundaries.

in degenerate cases, a particular quadrilateral may not fit one of the triangulation templates. The triangulation algorithm generates an error flag in such cases that is used to further decimate the quadrilateral. Moreover, elements of quality less than 0.5 may be generated along the boundary despite warping as shown in Figure 4.16. The cause of such triangles is due to neighboring (smaller) quadrilaterals whose end points must also be part of the triangulation or due to conflicting warps. The triangulation algorithm also marks such quadrilaterals for further decimation. Thus, triangulation is done iteratively within the **TreeBalance** procedure. The threshold for acceptable quality is set by the user and is less than 0.5 and is usually between 0.4 and 0.45. This allows a few triangles whose quality is less than 0.5, but greatly reduces the number of triangles.

Figure 4.17 shows examples illustrating the use of the above algorithm for grid generation. As can be seen, even very complicated geometries can be gridded satisfactorily. The figure also lists the quality of the triangulation. Roughly 5% of the elements are obtuse. Except for the last structure, the triangulations produced elements of quality greater than 0.5. For the last structure, the minimum quality was greater than 0.475.

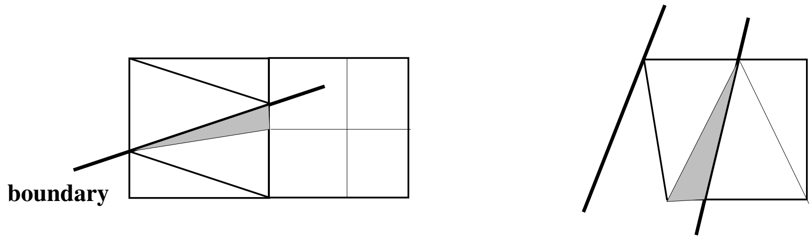
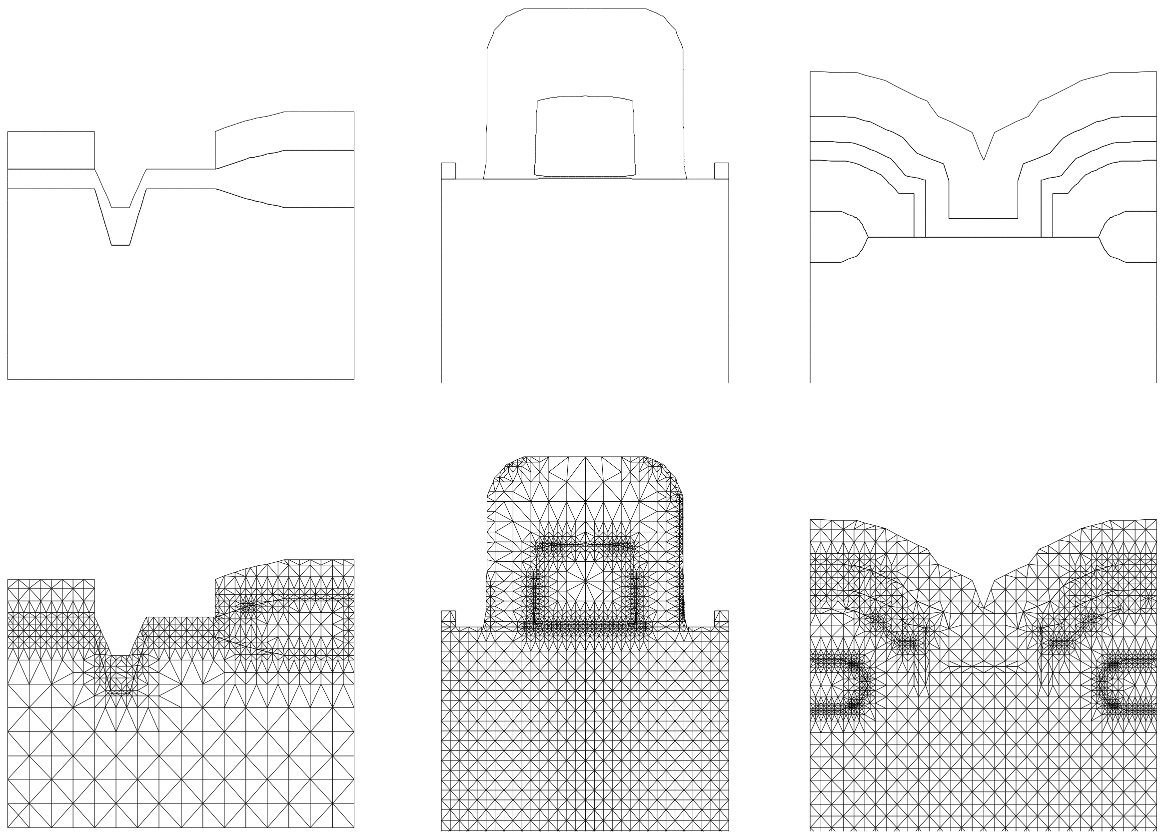


Figure 4.16: Smaller neighboring quadrilaterals or conflicting warps may produce poor triangles.

Once the triangles have been generated, post-processing of the elements generates auxiliary connectivity information. For example, many solvers require neighbor information - which element abuts which element? For a fully unstructured grid, this neighbor search problem can be expensive. In *Forest*, all grid points hold pointers to their elements. Thus, neighbors for each element are found easily by searching through the elements at the points of the element. Furthermore, the grid points are stored in an x- and y-coordinate sorted hash table. As each element is made, its vertex points are added to the hash table if the points do not already exist. This prevents duplication of points in the data structures. The hash table greatly reduces the search time to determine if a point already exists. As a new point is added, its bin is determined. The new point is compared to the existing points within the bin and added to a linked list of points in the bin if necessary.

4.5 Grid Adaptation

The automatically generated grid, however, may not be ideal for computation. More grid may need to allocated in some areas to reduce the error in the solution. Moreover, in process simulation, dopant atoms redistribute, changing grid requirements as the solution proceeds. After an implant, fine grid points are required near the surface to resolve the implant profile. As the implant is driven in, small elements are no longer needed at the surface as the profile flattens out. However, finer grid is needed at the tail end of the profile as gradients in the dopant concentrations are high. The grid should, thus, move along with the diffusion front. A method for grid adaptation is



Number of triangles = 1562
Number of obtuse triangles = 49
Best quality = 0.997516
Worst quality = 0.519615
Average quality = 0.826004
Quality < 0.5 = 0 out of 1562
Quality < 0.6 = 71 out of 1562
Quality < 0.7 = 316 out of 1562
Quality < 0.8 = 411 out of 1562
Quality < 0.9 = 1417 out of 1562

Number of triangles = 5800
Number of obtuse triangles = 279
Best quality = 0.999049
Worst quality = 0.503792
Average quality = 0.843018
Quality < 0.5 = 0 out of 5800
Quality < 0.6 = 61 out of 5800
Quality < 0.7 = 728 out of 5800
Quality < 0.8 = 941 out of 5800
Quality < 0.9 = 5456 out of 5800

Number of triangles = 5498
Number of obtuse triangles = 246
Best quality = 0.997047
Worst quality = 0.476711
Average quality = 0.835038
Quality < 0.5 = 10 out of 5498
Quality < 0.6 = 94 out of 5498
Quality < 0.7 = 853 out of 5498
Quality < 0.8 = 1171 out of 5498
Quality < 0.9 = 5046 out of 5498

Figure 4.17: Example grids generated by the *Forest* grid generation program.

therefore needed.

Several different adaptation criteria have been developed for the diffusion equations [27] [3] and error estimation is an active area of research. In this work, a simple solution variation criterion is used. The grid is refined if the variation in the dopant concentration is greater than one order of magnitude within an element[2]. Furthermore, refinement is performed only if the concentration on at least one of the element's points is greater that $10^{12}\,cm^{-3}$ for dopants and greater than $10^{8}\,cm^{-3}$ for point defects. Coarsening is performed if the variation in the solution at each of the grid points is less than $10^{0.2}\,cm^{-3}$.

Several different adaptation techniques are also popular as illustrated in Figure 4.18. Once a grid has been generated, the elements may be divided by adding their mid-points as points in the triangulation [27] [5]. This straight forward approach creates a triangular tree that may be pruned at various levels for dynamic gridding. However, some triangles may be of low quality if their parent was divided into two. Diagonal flipping and Laplacian smoothing will remove all such triangles. A different approach used by Baccus et. al. [3] creates a new point set which is triangulated akin to the Delaunay triangulation algorithm discussed previously. The approach used in *Forest* employs the quadtree for adaptation. If finer elements are needed, the quadtree is grown. If coarser elements are needed, the quadtree is pruned. Subsequent triangulation of the quadrilaterals generates the desired adapted grid.

All these approaches have their advantages and disadvantages. The first approach of dividing elements is very simple to implement and justify. Moreover, since points are added along the edges of existing triangles, interpolating a new solution onto them is extremely efficient as one need not search for an element containing the new point. However, edge flipping and Laplacian smoothing needed to improve the grid quality ruin the element hierarchy making it difficult to coarsen the grid. Laplacian smoothing also creates artificial diffusion as the points are moved. Interpolation strategies need to compensate for the artificial diffusion. The second approach using the Delaunay approach guarantees a high quality grid. However, new points may be allocated

[2]This heuristic adaptation criteria may be changed by the user as appropriate.

anywhere on an element leading to more interpolation error. Moreover, neither of the first two approaches allow anisotropic adaptation - refinement or coarsening in one direction only. The quadtree approach uses an element independent hierarchy allowing coarsening and refinement regardless of any post processing steps applied to the elements. As discussed below, it allows anisotropic adaptation. All points in the interior of the device are along the edges of existing elements reducing interpolation error in the interior. Due to warping, new points along the boundary may not be added anywhere leading to both high interpolation error along the boundary and to reduced adaptation efficiency as one must now search for elements containing a newly added point.

The algorithm for grid adaptation is as shown in Algorithm 4.6 and illustrated in Figure 4.18. The procedure first iterates through all the elements testing for refinement or coarsening (lines 1-8) and marking them appropriately. The leaf quadrilaterals are then traversed. If any element contained within them is marked for refinement, then the quadrilateral is marked (lines 9-11) and refined by calling procedure QuadDecimation (lines 15-16). The quadrilaterals are traversed again, this time checking for coarsening (lines 12-14). If all the elements within a quadrilateral are marked for coarsening, then the quadrilateral is removed from the tree (lines 17-18). The quadtree is re-balanced, re-warped and re-triangulated. Finally, solutions from the old point set are interpolated onto the new point set.

Algorithm 4.6 Grid Adaptation

procedure AdaptGrid
begin
1. $R := \emptyset$; $C := \emptyset$;
2. **for** each solution sol **do**
3.   **for** each element e **do**
4.   **for** $i := 0$ **until** $i = 2$ **do**
5.   **if** $\|sol[i] - sol[(i+1)\%3]\| >$ refine threshold **then**
6.   INSERT(e, R);
7.   **if** $\|sol[i] - sol[(i+1)\%3]\| <$ coarsen threshold **then**

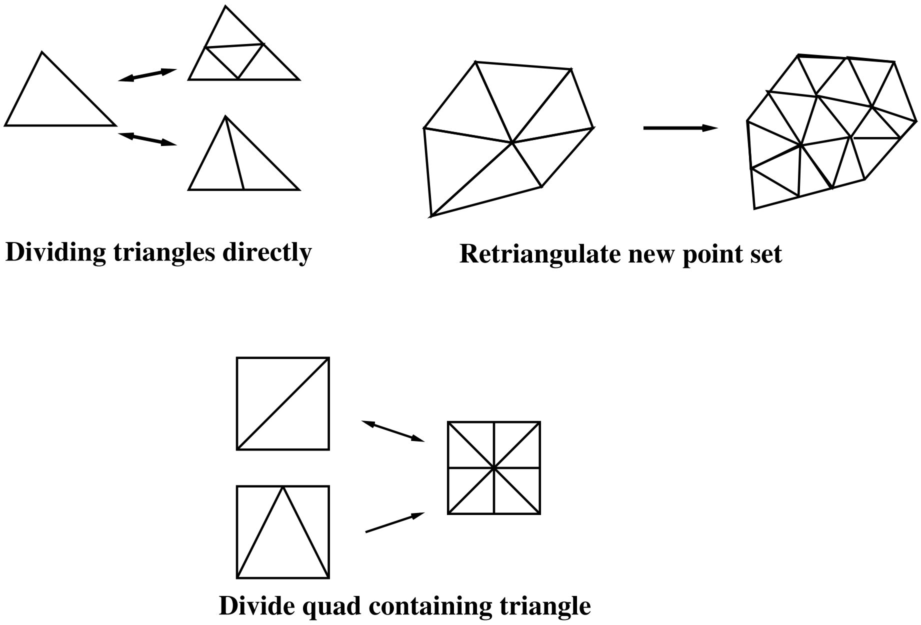


Figure 4.18: Grid adaptation using the tree. The tree is grown or pruned as necessary and the elements are regenerated.

8. INSERT*(e, C);*

QR := ∅; QC := ∅;
9. **for** each leaf quadrilateral q in quadtree **do**
for each element $e \subset q$ **do**
10. **if** $e \in R$ **then**
11. INSERT*(q, QR);*

12. **for** each leaf quadrilateral q in quadtree **do**
13. **if** *all elements* $e \in C$ **then**
14. INSERT*(q, QC)*

15. **for** each $q \in QR$ **do**
16. QuadDecimation*(q)*

for each $q \in QC$ **do**
    Delete*(q)*

BalanceTree( )*;*
Warp()*;*
Triangulate( )*;*
(interpolate solutions on new points *)*

end

Interpolation on new points can become a very computationally expensive if one is not careful. First, the element from the old grid that encloses needs to be found. Solutions on the point are then interpolated by solving a simple 3x3 matrix. If there are N points and M elements, a linear search matching points with elements will take $O(NM)$ time. Since both N and M are of the same order, the algorithm is essentially $O(N^2)$. If M can be limited or made equal to 1 in the ideal case, the search process will be reduced to linear complexity $O(N)$. For adaptation schemes that divide triangles, M is 1 as one knows the exact triangle that begets a new point. Due to warping in the tree based adaptation procedure, however, the exact location of new points cannot be determined beforehand and different techniques are needed to limit M. The *Forest* program uses a hash table for this purpose. As noted previously, all points in the grid are stored in a coordinate sorted hash table. Moreover, all points hold pointers to their corresponding elements. For each new point that is added, its hash table bin is determined. Elements at all the points within that hash table bin are first tested to determine if any of them encloses the new point. If an enclosing element is not found, then elements from points in neighboring bins are searched. This procedure is repeated with ever increasing circles of neighboring bins until the enclosing element is found. While this procedure does not limit M to 1, it reduces M significantly from N.

Figure 4.19 illustrates the use of this technique for implantation calculations. An implant is performed on a woefully coarse grid where the profile is fully contained

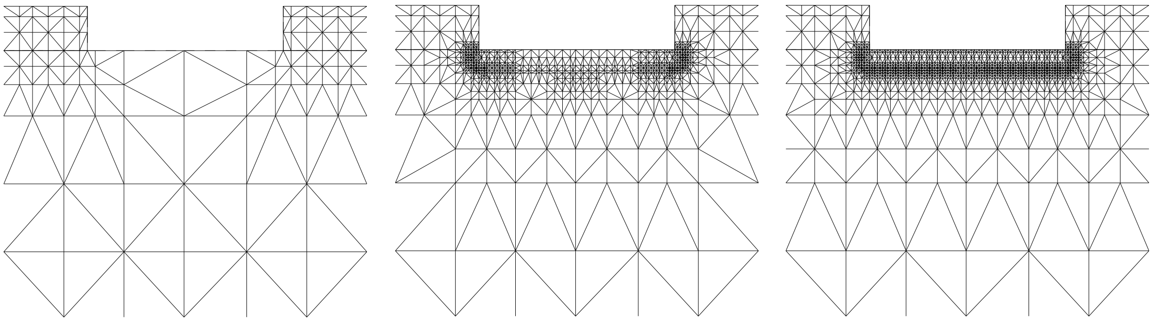


Figure 4.19: Example illustrating grid adaptation for implantation.

within two grid points. Grid adaptation is used to achieve better resolution. Since grid coarsening does not serve any purpose, only refinement is performed during implantation calculations. However, each time the grid is changed, the implant has to be recomputed for dose conservation. Thus, grid adaptation during implantation can be computationally expensive.

Figure 4.20 illustrates grid adaptation during a diffusion simulation. The as-implanted profile and grid from Figure 4.19 is subjected to a drive-in. As the profile moves and flattens out, the grid is refined in areas of high concentration gradients and coarsened in areas of low concentration gradients. As expected, the dense grid front moves down into the device during the drive-in.

4.5.1 Anisotropic Adaptation

The tree based adaptation mechanism has a unique advantage that can be exploited to produce anisotropic grids. The direction of solution variation can be measured on an element and if the variation is only in one direction, then the quadrilateral may be divided into two in that direction only rather than isotropically into four. This procedure, illustrated in Figure 4.21, greatly reduces the number of grid points. Figure 4.22 shows the grid ion implantation adaptation example from the previous section using anisotropic grid adaptation. The number of grid points in the anisotropic grid is approximately 60% of those used in the isotropic grid.

However, there are several disadvantages to anisotropic grid adaptation that need to be considered. Neither the boundary quadrilaterals nor the interior quadrilaterals

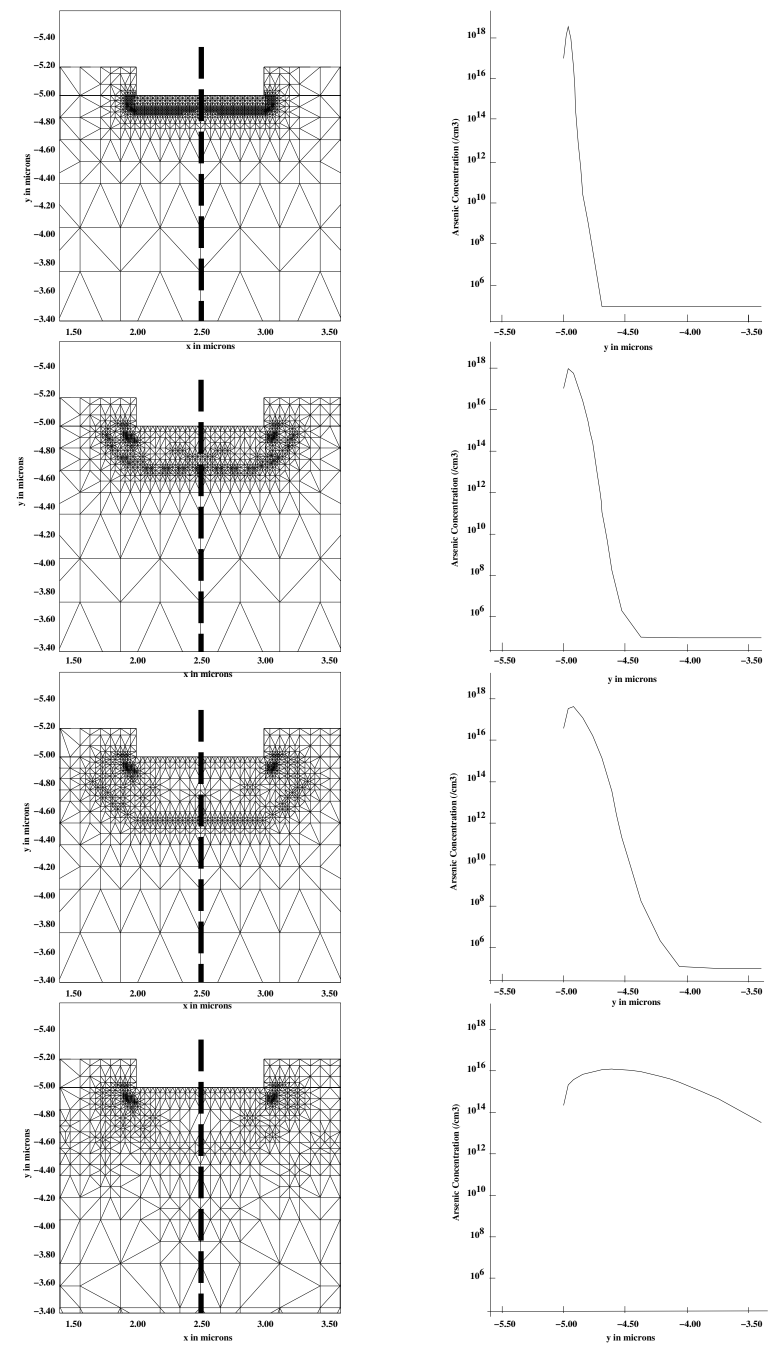


Figure 4.20: Example illustrating grid adaptation for diffusion. The grid tracks the diffusion front - reducing grid points where the profile is flat and adding points where the profile is steep. The doping profile plotted in the right is along the dotted line shown on the grids.

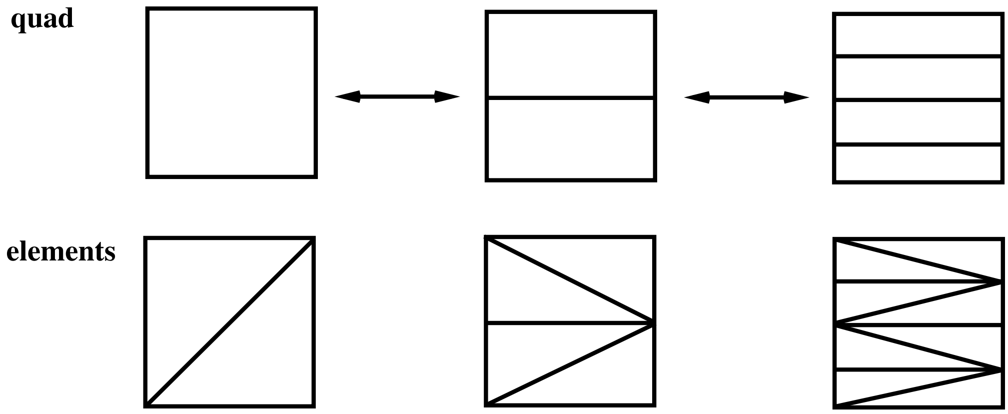


Figure 4.21: Anisotropic grid adaptation technique using the quadtree data structure. The quad is refined or coarsened in the direction of solution variation only.

Figure 4.22: Example illustrating anisotropic grid adaptation for implantation. The grids are shown prior to Delaunay flipping.

will produce triangles of high aspect ratio even with warping. The optimization criteria, however, should not be the aspect ratio. It should be the Delaunay condition - no obtuse angles. For interior quadrilaterals, the Delaunay condition is met if the quadtree is sufficiently divided. If a non-Delaunay triangle is found in a quadrilateral, that quadrilateral is divided in the direction that removes the offending triangle. The Delaunay diagonal flip may also be used to remove the offending triangles.

For boundary quadrilaterals, however, the warping algorithm guarantees good aspect ratio elements only when the leaf quadrilaterals are squares. If they are rectangular, then the algorithm will produce both obtuse and small angled triangles. Moreover, since one cannot perform a Delaunay flip for obtuse angles at the boundary, only a constrained Delaunay triangulation is possible. Thus, anisotropic adaptation should not be used at boundaries.

4.6 Grid Movement

Modeling oxide growth is the most difficult aspect of process simulation. The geometry changes as the oxide grows. However, the velocities are computed by solving a transport and reaction equation using a finite element method on a grid. The grid points themselves also move to simulate transport within the oxide. In addition to the problems associated with a moving geometry (regions crashing into one another, loop formation, etc) that were discussed in Chapter 3, several new problems associated with grid motion arise. In this section, these problems are discussed and algorithms to overcome them are presented.

As the oxide grows, the velocity of the grid points at the silicon/oxide interface directly point into the silicon. The velocity of the points immediately below the interface in the silicon is zero. Motion of the interface points into the silicon increases the volume of the oxide at the expense of the volume of the silicon. Thus, as the interface points move into the silicon, the triangle elements in the silicon will be compressed (their height will decrease while their base remains the same). Thus, the quality of the elements in the silicon decreases. On the other side of the interface, velocity of the first layer of points in the oxide is away from the interface representing

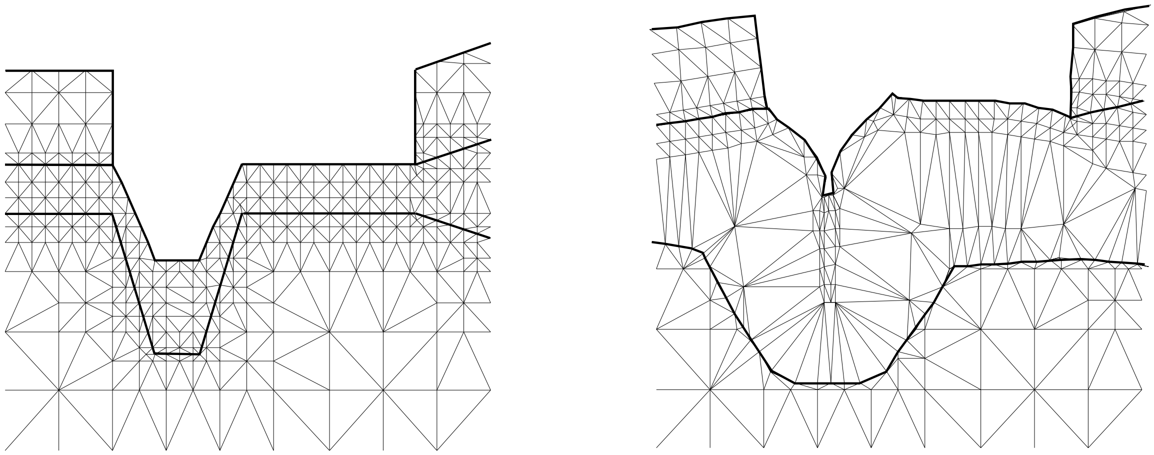


Figure 4.23: Grid motion stretches elements in the oxide reducing their quality and compresses elements in the silicon also reducing their quality.

volume expansion of the oxide. Thus, motion of these points stretches the layer of elements, once again reducing their aspect ratio (Figure 4.23).

Different approaches have been used to improve the quality of the elements after grid motion. The original approach in SUPREM-IV [45] attempted to remove the extremely small elements in the silicon. Moreover, additional points were added in the oxide. Another approach [42] moves all the points in the silicon by solving a simple elastic deformation equation in the silicon subject to the boundary condition of the interface motion. Since material is not transported (physically) in the silicon, one has to compensate for the artificial dopant diffusion by adding upwinding terms in the dopant diffusion equations.

Yet another problem associated with grid motion is one of points overtaking one another. Consider, for example, the case of the interface point moving into the silicon while the point in the silicon immediately underneath it being stationary. If a large time step is taken, then the moving point will overtake the stationary point. Thus, the time step during oxidation must be controlled.

The algorithms in Chapter 3 relied on delooping to remove geometry anomalies associated with geometry motion. In the case of oxidation, however, those algorithms present additional difficulties. As noted in Chapter 3, delooping adds extra material to the geometry. When triangles and points are allocated to cover this extra material,

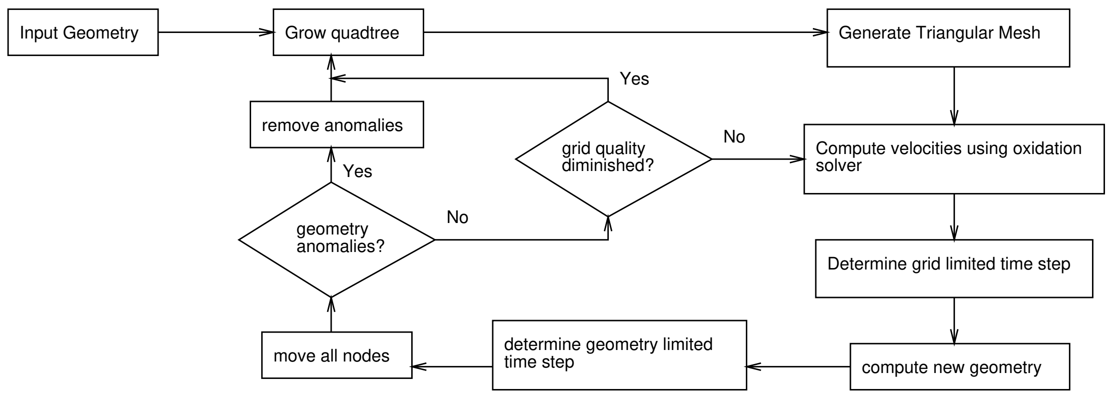


Figure 4.24: Flowchart for grid evolution during oxidation simulation.

one is faced with the arduous task of assigning solution values at these points. Since material did not exist previously at the spatial locations, no element can be found in the old grid enclosing those points. Solutions values must thus be interpolated based on the "closest" element. This entails searching through all the elements to find the closest element - a computationally expensive operation.

The grid movement algorithms in *Forest* address all these problems. The flowchart for the entire algorithm is shown in Figure 4.24. First, a grid is generated using the quadtree approach discussed above. Point velocities are then computed using SUPREM-IV's oxidation solver [45] on the triangular grid. The maximum motion limited time step is determined. All the points in the triangular grid are moved and a new geometry is computed. If geometry anomalies are encountered, the geometry is repaired. The triangular grid is regenerated if elements become highly deformed or if the geometry is repaired.

The maximum time step is determined as follows. The algorithm first takes a time step, t_0, moving all the grid points according to their velocities. Then the deformed grid and the geometry are tested to determine if the motion has caused anomalies. If an anomaly is detected, a smaller time step t_1 that is half as long as t_0 is taken. If an anomaly is not detected with the smaller time step, the time step is increased to the average of the smaller and larger time steps - $(t_1 + t_0)/2.0$. The time bisection procedure is repeated until the largest possible time step is obtained.

Three tests are performed to detect anomalies in the grid and the geometry as

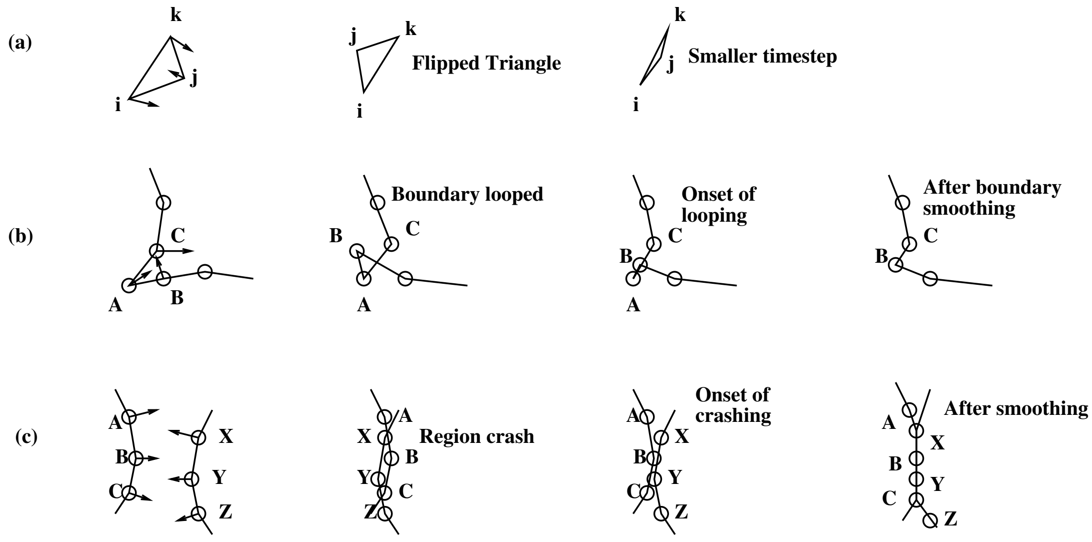


Figure 4.25: Grid and geometry criteria used to limit time step. (a) Nodes should not overtake one another. (b) Boundaries should not form self loops. (c) Regions should not crash into one another.

illustrated in Figure 4.25. In each case, the point velocity vectors are shown along with point movement with too large a time step and with the optimal time step. First, grid point overtake conditions are detected. Points on elements are ordered in a counter clockwise manner. If a triangle flips by one point overtaking another, the ordering of the points switches and the computed area will be negative. The next two tests attempt to minimize the effect of delooping. The idea is to take as large a time-step as possible *until* a loop forms. Thus, the geometry is checked for loops using the tests discussed in Chapter 3 at each time-step.

In order to check for geometry anomalies, the geometry must be moved. The oxidation solver computes velocities on the grid and not on the geometry. Thus, before the geometry can be moved, the server must build a geometry representation from the grid after moving the grid points. The following procedure is used.

Algorithm 4.7 Build Geometry From Grid

procedure GeometryFromGrid
begin

```
    (* find an element on the boundary *)
1.  E := set of elements in grid G;
2.  while (E ≠ ∅) do
3.      for each e ∈ E do
4.          if e has only two neighbors or
              if one of its neighbors is in another region then
5.            g := e;
6.            break;
7.          else
8.            DELETE(E, e);

9.      boundary B := ∅;
10.     f := neighbor of g also on the same boundary
11.     while (f ≠ e) do
12.         INSERT(B, boundary edge of f);
13.         DELETE(E, f);
14.         f := neighbor of f also on the same boundary
15.     DELETE(E, g);

16.     add B as a boundary and form corresponding region in the geometry
end
```

The first element that is on a boundary is found. Boundary elements will either have less than three neighbors or one of their neighbors will be in a different region. (lines 3-6). Starting from this first element, neighboring elements that have the same neighbor attributes are found. The edges of these elements that face the boundary are collated into a list (lines 9-14). This list forms a closed boundary which is added to the geometry (line 16). The procedure is repeated to find another element along another boundary. The loop is stopped when no such element is found (line 2). Note that the element list is continually shortened as each element is tested. Thus, no element is visited more than once. Thus, the algorithm runs in $O(N)$ time where N

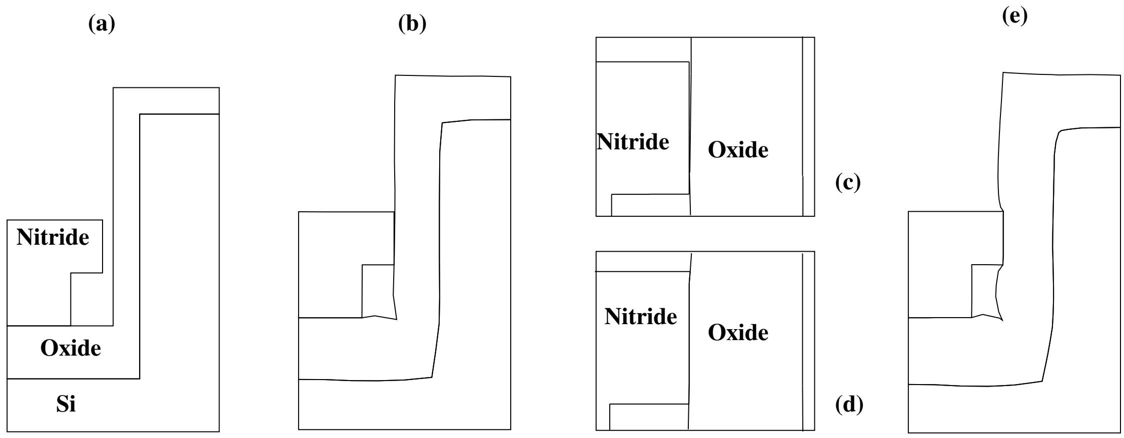


Figure 4.26: Oxidation simulation with colliding regions. (a) Initial geometry. (b) Geometry at the onset of crashing during oxidation. (c) Close-up view of regions before crashing. (d) Boundary smoothing to allow proper geometric coupling. (e) Subsequent oxidation.

is the number of elements.

As discussed above, using a small enough time step before a geometry anomaly occurs prevents formation of loops and crashes. However, the loop will happen at the next time step. Preventing the loop will entail taking an extremely small time step and the simulation will end. To continue the simulation after the onset of region crashing or boundary self-looping, a procedure akin to delooping is performed. The boundary is smoothed by collapsing nearby edges as illustrated in Figure 4.25. Essentially, the geometry is non-physically changed **before** a loop or a crash occurs. Since the collapse changes the geometry only minimally, the geometry is not compromised to as great a degree as it would if a full scale delooping were performed. The grid is updated to conform to the new geometry and the simulation is continued.

Figure 4.26 illustrates the utility of this approach in oxidizing complicated structures. The sidewall oxide crashes into the abutting nitride region. Time steps are chosen until the onset of the crash and the geometry is smoothed as shown in Figures 4.26c and 4.26d. The grid is updated and simulation continued (Figure 4.26e).

The grid is regenerated if either the grid quality diminishes or if the geometry is changed. The regeneration process illustrated in Figure 4.27 proceeds by unwarping

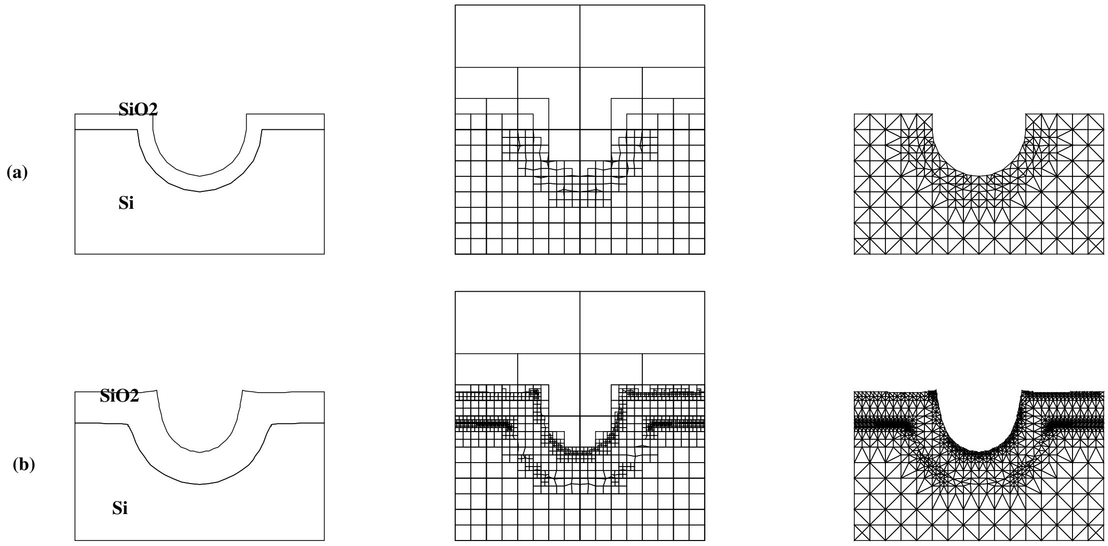


Figure 4.27: Use of Forest in oxidation simulation. Geometry, quadtree grid and triangular grids before (a) and after several oxidation steps (b).

all the leaf quadrilaterals. The intersection of each of the leaf quadrilaterals with the geometry is then recomputed. If the geometry extends outside of the old root quadrilateral, a new root quadrilateral is formed by recursively doubling the old root's size. All of the leaf quadrilaterals in the quadtree are then visited and pruned or grown. If a quadrilateral that previously enclosed an area of the geometry now does not enclose a material area, it is pruned - the subtree under it is discarded. Conversely, if an old quadrilateral encloses an area when it did not previously, it is grown. The new quadrilaterals are then warped and triangulated. This strategy guarantees a high quality grid at all times. Figure 4.28 shows another example of the use of this approach for oxidation simulation.

Solutions on the new points are interpolated from the old triangle grid and the old triangular grid is destroyed. The advantage of this approach is that the quadtree structure changes *only* along the moving oxide interface. The triangulation points in areas of no grid movement are the same - resulting in fewer interpolations between grids. Moreover, the level of recursive division of the quadtree can be controlled to make boundary quadrilaterals or quadrilaterals in areas of high solution variation finer, allowing simultaneous grid movement and adaptation.

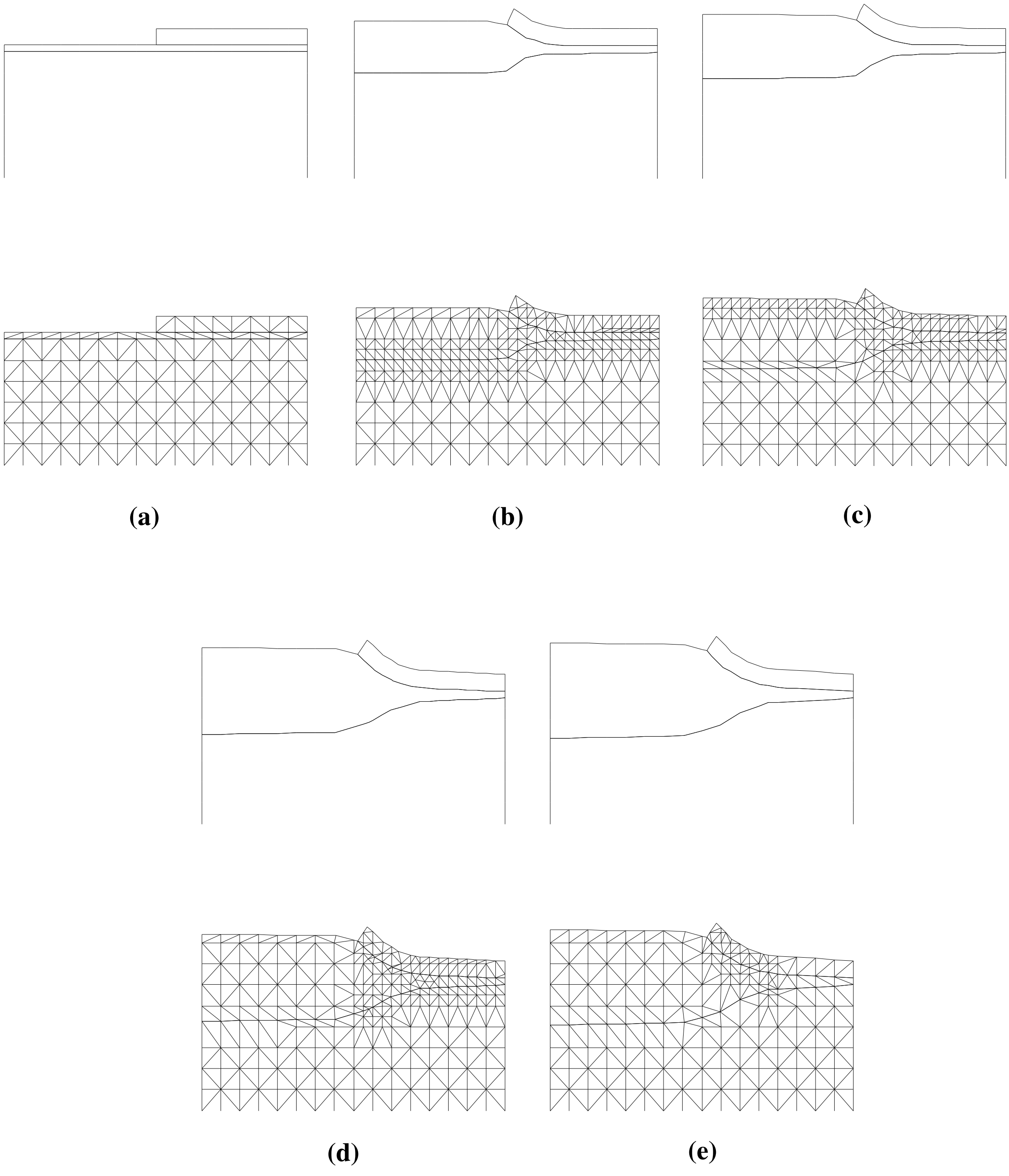


Figure 4.28: Oxidation example using *Forest*.

4.7 Summary and Conclusions

A quadtree based technique for generate grids in 2D is presented. Algorithms to service all grid requirements in process simulation - generation, adaptation and movement are presented. The quadtree grid generation technique divides the problem of generating a grid over a complex geometry into a large set of sub-problems of generating grids over quadrilaterals. Algorithms for the reduction problem and for triangulating quadrilaterals are discussed. Grid adaptation based on pruning and growth of the tree is also considered. Anisotropic adaptation illustrates the power of the tree based approach. Problems encountered during grid motion are illustrated and the techniques developed as part of this work to overcome these problems are presented. Oxidation simulation utilizes all aspects of the *Forest* program - geometry representation, geometry motion, loop detection, delooping, grid generation, grid adaptation and grid movement.

Chapter 5

Heterogeneous Process Simulation Tool Integration

5.1 Introduction

As discussed in the previous chapters, full process simulation involves the numerical solution of equations describing the physics of diffusion, oxidation, lithography, ion implantation, etching and deposition steps, resulting in geometry and doping profiles that define a device. Traditionally, different tools have focused on the solution of each of the different process steps. SUPREM-IV [30] and FLOOPS [31] solve diffusion and oxidation equations; CREEP [50] solves only the oxidation equations; SPEEDIE [33] and SAMPLE [36] solve boundary motion equations for etching and deposition. For greatest flexibility, it is desirable to use the best available tool for simulating a given process step. Moreover, several different geometry and grid programs similar in functionality to *Forest* exist. ACIS [1] and the IBM Geometry Engine [29] are full 3D geometry modelers. Meshbuild [35], Omega [13], Trigen [5] and many other grid generation programs have been used and adapted for TCAD. Ideally, one would like to interface with any and all of these programs choosing the best program based on functionality. Thus, integration of different tools (programs) is a compelling need.

Three different levels for achieving this integration are considered: (1) file formats, (2) information models and (3) server models. If the output data file from any tool can be used as the input data file for any other tool, all tools can be seamlessly used together. A standard file format provides this capability and obviates the need for format conversion programs. If all tools use a standard internal data structure or information model, each tool and its subroutines can be used as sub-functions within a larger simulation environment. The tools can be developed independently and used interchangeably. However, while an information model is useful for new tool development, existing tools can be integrated through a file in lieu of a total program rewrite. Conversely, a file format alone, while ideal for integrating existing tools, does not provide the benefits of code-reuse (assuming no supporting library) or standard information models for writing new tools.

Moreover, data file formats and information models are not sufficient for tool integration. The problem arises from incompatible data representations and manipulation techniques of the different tools. Previous chapters have considered each tool and it's data representation and manipulation techniques without regard to other tools. Consider now the integration of a grid-based diffusion tool and a surface-mesh based etching tool. The diffusion tool output consists of a grid with dopant concentration solutions. A surface mesh can be easily constructed from the grid. This surface mesh can be transformed by an etching solver to create a new device geometry. For further diffusion simulation, the new geometry must be gridded, and the dopant solution values interpolated from the old grid - services that etching solvers typically do not provide, due to lack of data representation and algorithm capability. Clearly, a "server" to reconcile the new geometry with the old grid without any data loss is needed.

In this chapter, a new heterogeneous TCAD tool integration strategy focusing on server functionality is presented. Data needed by the various TCAD programs is identified. The geometry, surface mesh and grid modules of the *Forest* system discussed in previous chapters are formalized into servers that are carefully limited in scope and defined with functional abstraction to contain and operate on the data. Both existing and new tools benefit from the clearly delineated roles and interactions of the three servers[1]. A minimal standard server API to aid tool integration is specified.

[1]Note that the term "server" refers to a combined function library and data repository that

Results from integrating existing tools, such as SUPREM-IV and SPEEDIE as well as other tools described in previous chapters, are used to illustrate the strategy.

After reviewing previous work in Section 5.2, the integration architecture and servers are presented in Section 5.3. Implementation results are presented in Section 5.4. Server functions that define a new tool integration standard are presented in Appendix A.

5.2 Previous Work

Earliest work in tool integration involved data conversion programs interfacing process and device simulators. The multi-organization European DAMSEL [14] effort introduced common FORTRAN-77 data structures defining geometries, grids, solution values and matrices for integrated process and device simulation. While several integrated process and device simulation tools were written with this data structure, integration of other independently developed tools was not considered.

The pioneering work of Duvall [16] at Intel defined a "Profile Interchange Format" (PIF) enabling broad inter-operability among TCAD applications. PIF specified a self documenting, dimensionally independent, data file format for storing all simulation data. The highly successful VISTA system [18] from TU-Vienna overcame minor ambiguities in PIF by further standardization and provided a toolkit library. The integration framework with a VISTA-like system using standard data file formats is illustrated in Figure 5.1. As can be seen, independent applications can easily be interfaced into the system. Modified PIF formats were also used in the Philips design environments IDDE [22] and PRIDE [49].

Working with PIF objects (coordinates, lines, regions, elements, grids, etc), the in-memory information models PIF/Gestalt [9] at MIT and BPIF [53] at Berkeley were developed. PIF/Gestalt differed from BPIF only in the degree of definition formality and conceptual data hiding. Object manipulation operations were mirrored in the PIF

provides data and services for client TCAD applications. While similar to the traditional client-server concept, in the current implementation, the server is not a separate process; instead it is linked with the client applications during compilation.

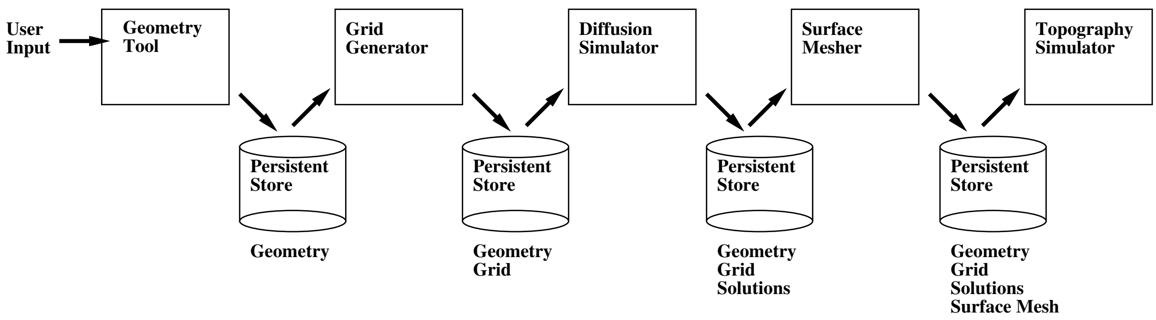


Figure 5.1: Tool integration using a standard data file format. The VISTA system uses PIF as the common file format. All applications read and write standard PIF objects using a toolkit library.

database. As an information model, however, the use of PIF/Gestalt for integrating existing tools was limited and tools like SUPREM-IV were integrated using a file interface. Both these techniques were also very closely linked to and constrained by the original PIF model.

Berkeley's SIMPL-IPX system [52] provided utilities to integrate 2D topography and impurity profile simulators. Geometries were updated from SAMPLE generated etch profiles and SUPREM-IV grids were updated from geometry profiles.

Recent work towards tool integration has been based on an object-oriented client-server architecture called the Semiconductor Wafer Representation (SWR) [21] [54]. The multi-organization American SWR effort defined an information model consisting of a hierarchy of C++ classes and utilities for geometry and field (grid) servers. However, the complete specification of the class hierarchy and utility methods discouraged alternative implementations. The inconsistent separation between geometry and field servers also led to complex server-server and server-application interactions. Moreover, while the geometry server was defined to have all the functionalities of a complete geometry modeler, the grid servers were defined to have *representation* capability only. Grid generators were defined as clients.

The technique presented in this chapter builds upon the experiences of these previous approaches. Applications are presented with a data repository-like interface to servers that also provide a functionally abstracted and simplified information model. Server utilities provide all the necessary functions for integrating new and existing tools. Grid servers are defined to include a grid generator; geometry servers are defined to include several functions to move, validate and repair geometries.

5.3 Integration Architecture

As illustrated in Figure 5.2, three different types of tools are typically used for TCAD simulation: (1) Grid based tools (eg. diffusion solvers) that solve for quantities inside the device body, (2) Surface mesh based tools (eg. etching and deposition solvers) that solve for quantities on device boundaries and (3) Geometry based tools that "simulate" topography steps by performing Boolean geometric operations. Other hybrid simulators (eg. oxidation programs) solve for quantities in the interior and on the device boundaries, typically using gridded structures. Each of these simulators and the data representations and required utilities were discussed in previous chapters. Since the different types of tools each use different device and attribute representations, any integration environment must support all these data representations and thereby adjudicate incompatibilities.

The integration architecture of the *Forest* system is shown in Figure 5.3. It consists of three core modules - a geometry modeler, a surface mesh generator, and a grid generator. The core modules together with access functions comprise a "server[2]." The access and utility functions therein specify a minimum server capability. The data within these modules is accessed and manipulated only via an application procedural interface (API). The interface is kept as simple as possible without any explicit hierarchy to allow easy translation, inline function, or #define access to the internal data of the core modules. Applications use the API to obtain data from the servers, perform computations on local representations and update the servers as needed. The process can be viewed as simple input/output (I/O) between an application and the server even though the server performs many functions that are hidden from the application as discussed below. The advantage of the data repository I/O view of the server is that existing applications can be easily integrated without extensive source-code changes.

[2]The collection of the three servers is also called "server." The context is used to determine the appropriate definition.

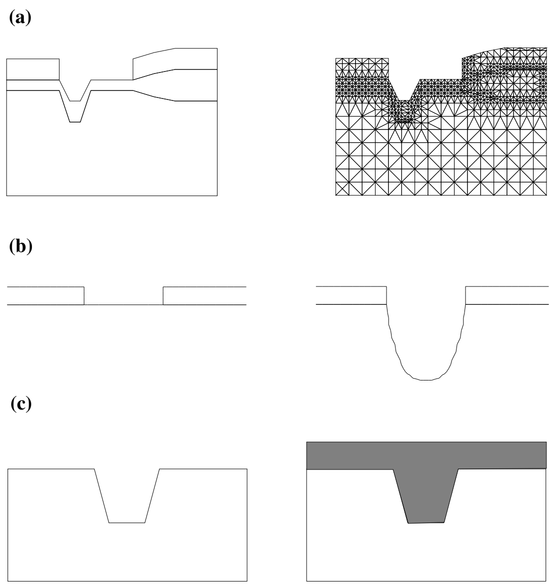


Figure 5.2: Different tools used in modern day TCAD require different data types. (a) Tools that solve for interior quantities require grids. (b) Tools that solve for boundary quantities require surface meshes. (c) Geometry manipulation tools require a wire geometry.

Figure 5.3 shows schematically a restructured version of SUPREM-IV that supports the integration architecture. The code example shown in Figure 5.4 represents information interactions between *any* pair of oxidation and diffusion modules. Considering only the diffusion step in Figure 5.4, the current solution values and grid are obtained from the server and the diffusion module's data structures are constructed (line 3 in the figure) at each time-step. The diffusion module operates on this data and computes a new solution for each time-step (line 4). Upon completion of the time-step, the new solutions are used to update the server's data areas and the diffusion module's data areas are deleted (line 5). This procedure is continued throughout the solution process. The scheme adds a minor data conversion overhead at each time step, while providing the benefit of allowing specialized grid techniques to be applied and changed seamlessly.

Note that an error estimate is used for grid adaption (lines 8 and 9). The error estimator's data representation is different from that of the diffusion module. Hence, the error estimator obtains grid and fields data directly from the server (line 10 in

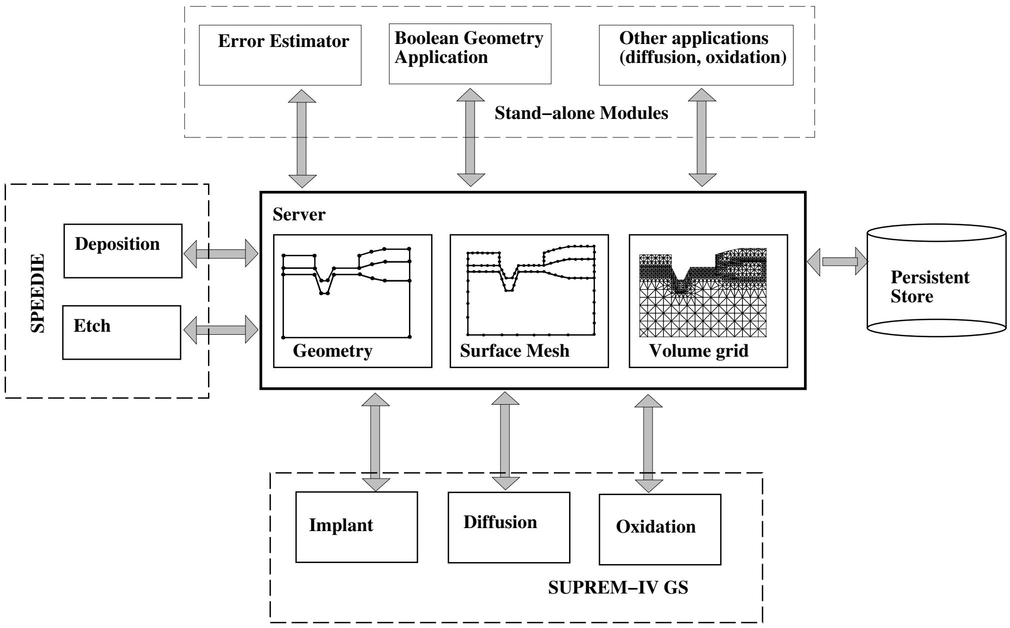


Figure 5.3: Integration architecture with server and several application tools. Three core modules - geometry modeler, surface mesh generator and grid generator - comprise a server that applications programs can use for data and functionality.

Figure 5.5) and sends updates to the server (line 12).

Integration of a stand alone oxidation program[3] also called from the diffusion solver is illustrated. The oxidation solver is called from within the diffusion loop (line 1) only to compute *velocities* on the grid points. Grid motion is performed by the server. Moreover, since the oxidation solver uses a different grid than that used by diffusion, it calls the grid server separately (line 14 in Figure 5.5) and updates velocity solutions (line 16) independently from the diffusion solver. Upon completion of each diffusion step, dopant data is updated, error estimations are computed and grid adaption is performed. However, the diffusion solver must be able to adjust the grid points to properly account for dopant flux across moving interfaces. Since the responsibility of boundary motion is relegated to the server, a potential conflict arises. This difficulty is resolved by requiring the diffusion solver to update the new dopant concentration solutions, including grid motion, before the server actually moves the grid points.

[3]one that is independent of the diffusion solver.

```
#include ``Grid.h´´    // server header file

diffuse(double time, double temperature)
{
 Grid *vgrid;                // Server´s grid
 MyGrid local_grid;          // structure that will hold local diffusion
 MyField local_fields;       // grid and concentrations data

 vgrid = GridGenerate(ServerGeometry, Tri3 );  // server generate a grid

 // do the time loop
 delta_t = inital_time_step;

 while ( t < time ) {
    if (oxidizing) {
(1)    do_oxide( vgrid, temperature );
(2)    oxide_t = GridMaxDt(vgrid, delta_t );
      }

    // Translate data from Server and get all fields
(3) ServerToDiffuse(&local_grid, &local_field, vgrid);

                                   // solve diffusion step
(4) delta_t = do_diffuse(local_grid, &local_field, oxide_t, temperature);

                                   // update solutions in server
(5) DiffuseToServer(local_grid, local_field, vgrid);

                                   // Move the grid
(6) rc = GridMove(vgrid, delta_t, ``xvel´´, ``yvel´´, NULL );

(7) t += delta_t;

(8) error_estimate(adaptname, vgrid);      // compute error estimates

(9) rc = GridAdapt(vgrid, 1, adaptname);     // server adapt the grid
  } // end of while loop
} // end of function diffuse()
```

Figure 5.4: Code fragment illustrating information exchange between server and multiple applications for a coupled oxidation/diffusion simulation.

```
void error_estimate( char **adaptname, Grid *vgrid )
{
 EEGrid ee_grid;  EEField ee_field;

                        /* first translate server data to estimator data */
(10) ServerToErrorEstimator(&ee_grid, &ee_field, vgrid);

    /* do the actual computation and mark elements for adaptation */
(11) do_estimate(&ee_grid, &ee_field, adaptname);

                         /* back to server */
(12) ErrorEstimatorToServer(ee_grid, ee_field, vgrid);
}

void do_oxide( double temperature, Grid *vgrid )
{
 OxideGrid  oxgrid;  OxideField oxfield;

     // translate to oxide info model
(14) ServerToOxide(&oxgrid, &oxfield, vgrid);

     // acutal computation done here
(15) ComputeVelocities(oxgrid, &oxfield);

     // send velocities back to server
(16) OxideToServer(oxgrid, oxfield, vgrid);

} /* end of function do_oxide */
```

Figure 5.5: Continuation of code fragment from Figure 5.4.

When the server moves the grid points (line 6), concentrations are then assigned to the correct spatial points. The actual boundary motion performed by the grid server also invokes the geometry server as described in Section 5.3.4. Yet, these details are hidden from the diffusion and oxidation applications.

This overall strategy allows large, complicated applications and servers to be developed separately and linked with any set of other applications and servers. Data structure or programming language constraints are not imposed on either application or server. The oxidation and error estimation modules, for example, are in fact separate applications. In this manner, a number of different diffusion, oxidation and error estimator tools can be used collectively and interchangeably without requiring custom interfacing. Only the translation routines that create each tool's data structures and update server data need to be created and they are the responsibility of application programs. Applications can access data from any or all of the three servers as appropriate.

The functionality of three - geometry, surface mesh, and grid - servers in *Forest* and their use in an integrated TCAD simulation environment are described in the following subsections. Any server implementation that provides the same capability using the API described in Section A can be used in the integration framework. The specific server implementations and associated algorithms of this work have been discussed in previous chapters and are only briefly delineated in Section 5.4.

5.3.1 Geometry Server

The geometry server stores and manipulates the gross structural features of a device by tracking material boundaries. Geometric capabilities support both physical simulation and rapid structural prototyping. The geometry server has the following functionality:

- Read/store a geometry from persistent storage,
- Update geometry with a new geometry,
- Determine if a geometry is valid,
- Move the surfaces of the geometry,

- Repair the geometry if it becomes invalid,
- Determine the maximum surface displacement,

Geometry applications typically compute a change in the geometry and request the server to update the current global geometry. Each application is given the freedom to destroy a geometry and to construct a new one. In-place geometry modification is provided to support simulation steps such as oxidation and etching/deposition even though applications might corrupt the geometrical connectivity. While detecting anomalies and repairing them is extremely difficult in the most general case, even in 2D, a few operations are useful for full process simulation. Thus, the geometry server provides three types of geometry checking and repair functions. First, since material growth (topography and oxidation) simulations generally create loops particularly at concave corners, loop detection and removal are provided. Second, as some applications use only a subset of the geometrical structure to perform computations, an advancing front might crash into another region. Crash detection and boundary collapsing operations are provided by the server. Third, since the geometry is sometimes built from a dense grid or surface mesh, the geometry data is often complicated. Smoothing operations are used to reduce the complexity.

For many processing steps, most notably lithography and topography steps, a geometry operation is used instead of a physically based simulation [51]. The types of operations which fall into this category include Boolean set operations (e.g. intersection), geometric constructions (e.g. sweeps), translations, and rotations. The operations required for such manipulations have been relegated to applications.

5.3.2 Grid Server

Many TCAD applications require gridded structures - a collection of elements whose union forms the device cross section and solution values that are defined over grid points or elements. In addition, a set of utilities is required for grid manipulation as the structure changes physically due to processing. The following functions are provided by the grid server to support these requirements:

- Generate area/volume grid on the geometry,
- Define solutions over the grid points and elements,
- Adapt grid for solution efficiency and accuracy,
- Move grid to simulate material transport and structural change,
- Determine if grid connectivity is valid,
- Repair corrupted grid,
- Determine maximum nodal displacements to preserve grid validity.

The grid server within this framework includes a grid generation program that can generate grids for arbitrary multi-layer and multi-material geometries. Higher order elements are supported to allow grid points anywhere within an element. The server also maintains solution values defined over the grid points and over the elements. Applications add and delete solution variables and values. The grid server interpolates solutions from one grid to another as the grid changes. The type of interpolation used for different solutions is specified by the application (eg. linear, log, dose conserving, etc.)

Grid adaptation is performed by the grid server, thus decoupling adaptation from the solver. The criteria for adaptation is computed by a separate error estimate application. The grid server refines or coarsens the elements and interpolates solutions between the original grid and the adapted grid. Upwinding solvers that compensate for grid adaptation induced artificial diffusion [42] need to instruct the grid server to hold the displacement and previous node positions as extra solutions to be used at the next time-step.

The grid server moves the mesh to simulate material transport; applications only compute nodal velocities or displacements. Since grid motion may result in an invalid grid, the grid server determines the maximum displacement or time-step that can be used before the grid becomes invalid. Even if the grid is corrupted or its quality diminishes, the grid server makes the necessary repairs. The grid server also interacts with the geometry server continuously to avoid geometrical anomalies arising from grid motion.

5.3.3 Surface Mesh Server

Programs such as SPEEDIE solve for quantities on device boundaries using a surface mesh. For moving boundary problems, the surface is advanced based on a velocity solution at the mesh points. Since there is no fundamental difference between grids and meshes, the functions defined for the surface mesh server are analogous and equivalent to those defined for the grid server. Differences resulting from application-specific use are illustrated in the following subsection and sections.

5.3.4 Interaction between Geometry and Grid/Mesh Servers

For moving boundary problems such as oxidation, etching and deposition, the geometry, surface mesh and grid should be kept consistent between processing steps. The required interactions between the respective servers are discussed in this section. Note that the responsibilities of the different servers is unambiguously divided.

After a geometry application has finished its operations, the grid and surface mesh need to be computed. As all grid based applications call the grid generation function before they access the grid, re-synchronization between the geometry and grid is performed by the grid generation functions.

During surface mesh based etching and deposition simulation, the surface mesh server moves the surface mesh and calls the geometry server to update and repair the geometry. If the geometry changes due to repair, the surface mesh is regenerated. While the surface mesh and geometry are continuously updated, the grid is updated only upon completion of the unit process simulation step and only when a grid based application calls the grid server to generate a grid.

For oxidation simulation, the grid server moves the grid points based on the computed displacement field. The grid server calls the geometry server to update the geometry; in turn, the geometry server performs geometry validity tests and repairs the geometry as needed. The grid is then regenerated with this new geometry and solutions are interpolated. In this case, the geometry and grid servers are continuously re-synchronized. The surface mesh is updated only upon completion of the entire oxidation simulation and only when a surface mesh based application calls the surface mesh server to generate a mesh.

Some etching and deposition tools use a cell based approach [39] - adding or deleting cells of materials and thereby simulating material growth. Since cells are elements in a grid, the grid server is used to generate the cells. In this case, however, an extra "gas" region above the device cross section needs to be defined and gridded by the grid server. Cell deletion and addition is performed by changing the material attribute of interface elements. Upon completion of the growth simulation, the new geometry is extracted and updated. The surface mesh is updated upon a subsequent call to the surface mesh generator.

Implicit in their interaction is the ability of the three servers to update their respective view of the device based on the view of another server. The geometry server has to update the geometry based on a (changed) grid or mesh. The grid and mesh servers need to update the grid or surface mesh based on a new geometry. Since the grid and mesh servers encapsulate a grid or mesh generator, grid and mesh updates are performed when an application specifies a (re-)generation. On the other hand, the geometry server is updated when the grid or mesh server *computes* a new geometry based on its view and sends the new geometry to the geometry server. The geometry server incorporates this new geometry with its current view.

Note that consistency does not imply congruency between the different data representations. For example, the grid only approximates the geometry of the device. Similarly, surface mesh nodes and volume grid nodes do not necessarily coincide. Applications that require congruency should use one representation to build the other representations, using the provided utilities. The grid representation can be used to build congruent surface mesh and geometry representations while the surface mesh representation can be used to build a congruent geometry representation.

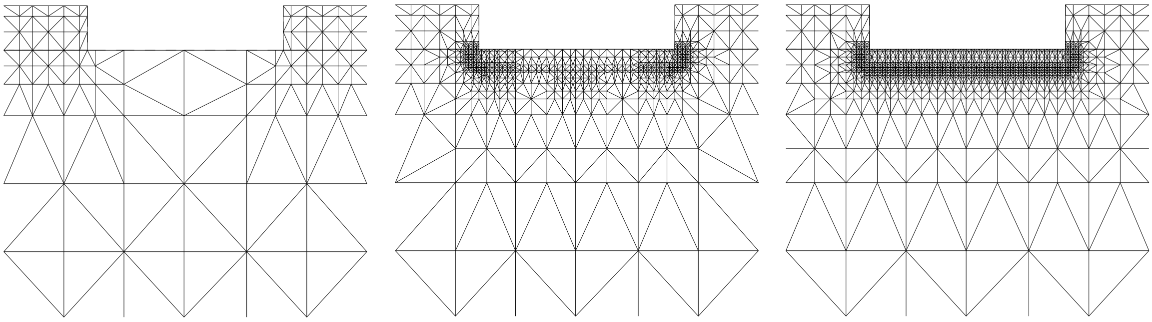


Figure 5.6: Example use of ion implantation with grid adaption in the new integration framework. The implantation module is from SUPREM IV. The error indicator is a new application. Grid adaptation is performed by the grid server.

5.4 Integration Results

All the examples of using *Forest* with SUPREM-IV, and SPEEDIE presented in previous chapters are performed with the server-based integration approach presented in this chapter. SUPREM-IV provides ion implantation, diffusion, and oxidation simulation capabilities; SPEEDIE models topography changes due to deposition and etching. Both these programs use the specified server APIs. A new error indicator tool supports grid adaptation and also uses the grid server API. In this section, the results from previous chapters are summarized and additional results are presented, illustrating the power of the integration approach.

Results of the interaction of two different programs, each with its own internal grid representation, through the server is illustrated in Figure 5.6. Sequential grid frames are shown during grid adaption for an implant simulation. SUPREM-IV provides ion implantation capability while a new error indicator tool supports adaptation. *Forest* performs the actual grid adaptation. In this case, the implantation application is the controlling application, calling both *Forest* and the error indicator. The error indicator interacts independently with *Forest* to update the server with its results (a flag indicating whether an element should be refined, coarsened or left alone). To minimize interpolation error, the implant application recomputes the profile and updates the server after each adaption until no more adaptations are performed.

The diffusion application is, from the server's perspective, the same as the ion implantation application. Adaptation during diffusion simulation is illustrated in Chapter 4. However, since the diffusion application in SUPREM-IV computes a symbolic factorization of the global stiffness matrix, based on the current grid, a new grid at every time step results in repeated symbolic factorizations. But the grid connectivity is changed only by regridding during oxidation or a grid adaption step; hence, no symbolic factorization is performed unless these conditions occur.

When the server is updated within an adaption or time-stepping loop, the number of updates can be large and efficiency is of the utmost importance. The penalty for using the server, defined as the total server access time, is the sum of the times taken to: (1) obtain grid and solution data from the server, (2) to build the local representation (including building all derived connectivity information and calculation of coupling coefficients, etc.) and (3) to update server solution values at each time step.

Figure 5.7 compares the *Forest* server access time with the total CPU time for several SUPREM-IV diffusion simulations. Breakdown of the CPU time between the three operations is also shown. For the simplest diffusion simulation of a boron implant drive-in using the "fermi" model, server access time can be as high as 20% of the total CPU time and is nearly independent of the number of nodes. The biggest part of the total server access time for SUPREM-IV, however, is spent building a local congruent geometrical representation from the grid representation that is *not* used for diffusion calculations. Simulation times for more advanced diffusion models are higher and consequently, the fractional cost of server access is reduced.

Interaction between the geometry and grid servers is illustrated by the new Boolean applications which provide geometric etching and deposition. The applications compute a new geometry and update the geometry server directly. Figure 5.8 illustrates the functionality of the deposition application and the interaction of geometry and grid servers. In Figures 5.8(a) and 5.8(b), the data of the geometry and grid server before the Boolean geometry operation is shown. The Boolean operation is performed as illustrated in Figure 5.8c. The grid is subsequently regenerated as shown in Figure 5.8d.

Chapter 3 showed several examples illustrating the interaction between SPEEDIE and *Forest* for physically based topography simulation. SPEEDIE uses the *Forest*

			Server Access Time				
No of grid nodes	Defect Model	No of time steps	Data get time (sec)	Local data build time (sec)	Server update time (sec)	Total simulation time (sec)	% Server access time
736	Fermi	12	5.9	15.9	4.3	132.5	19.7
894	Fermi	13	8.2	22.1	5.6	160.4	22.4
1631	Fermi	17	20.5	58.5	22.2	461.5	21.9
894	Transient	15	9.2	25.3	6.5	342.2	12.0
894	Fully Coupled	15	9.2	25.3	6.6	370.2	11.1

Figure 5.7: Comparison of server access time with total 2D diffusion simulation time using SUPREM-IV and *Forest*. Boron was implanted through an oxide window (dose = 1e14, energy = 40keV) and driven in at 1100C for 30 minutes.

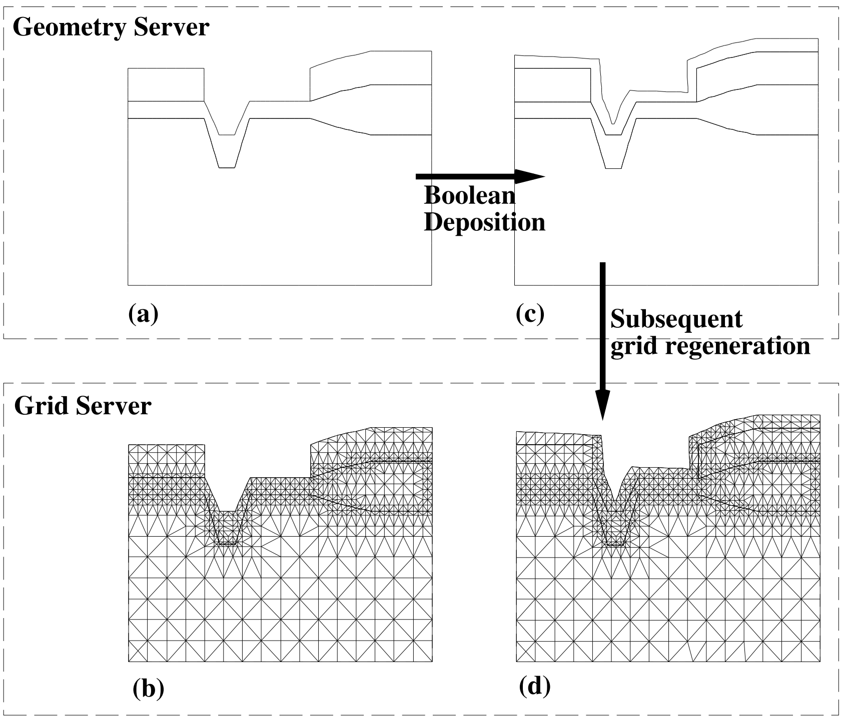


Figure 5.8: Illustration of Boolean applications using the new integration framework. Operations are performed with the geometry server. The grid is generated subsequently as needed.

Operation	CPU time (s)
Get Data from Server	1.4
SPEEDIE Velocity Calculation	208.1
Server Update	10.7

Figure 5.9: CPU usage for a SPEEDIE oxide deposition. 0.04 micron long surface mesh elements ($\simeq$ 100 mesh points) were used to deposit a 0.25 micron thick layer. 28 time-steps were used to complete the simulation.

server for all surface mesh and geometry operations. It obtains a surface mesh from the server, computes velocity vectors for the mesh points and then instructs the server to move the mesh. The server, after moving the mesh, computes a new geometry from the surface mesh. The geometry server checks for loop formation and performs delooping. The surface mesh is then regenerated for further simulation. CPU usage for a typical simulation is listed in Figure 5.9. Time spent in the server performing surface movement, loop detection and removal takes only 6% of the total CPU time.

Use of the coupled diffusion/oxidation modules shown in Figure 5.4 is illustrated in Chapter 4. In this case, SUPREM-IV provides the functionality of both diffusion and oxidation applications; the server brokers sub-sets of the data for the separate application modules. The diffusion solver uses the entire domain (all material regions) for computing dopant concentrations while the oxidation solver uses only the oxide and nitride regions to compute grid point velocities. Data translation routines access the necessary elements in the grid and build a separate representation for the oxidation application. Such an interaction will be extremely cumbersome in a file based implementation.

The oxidation application obtains an initial grid from the grid server and computes velocity fields over the grid points. The grid server is then called to obtain the maximum allowable time step to maintain grid validity. The geometry server is then called to obtain the geometry limited time step to avoid boundary loops and region crashings. The application then calls the server to move the nodes for a given time-step which may be greater than the maximum time steps discussed above. The geometry server moves the geometry and repairs anomalies - loops and region crashes.

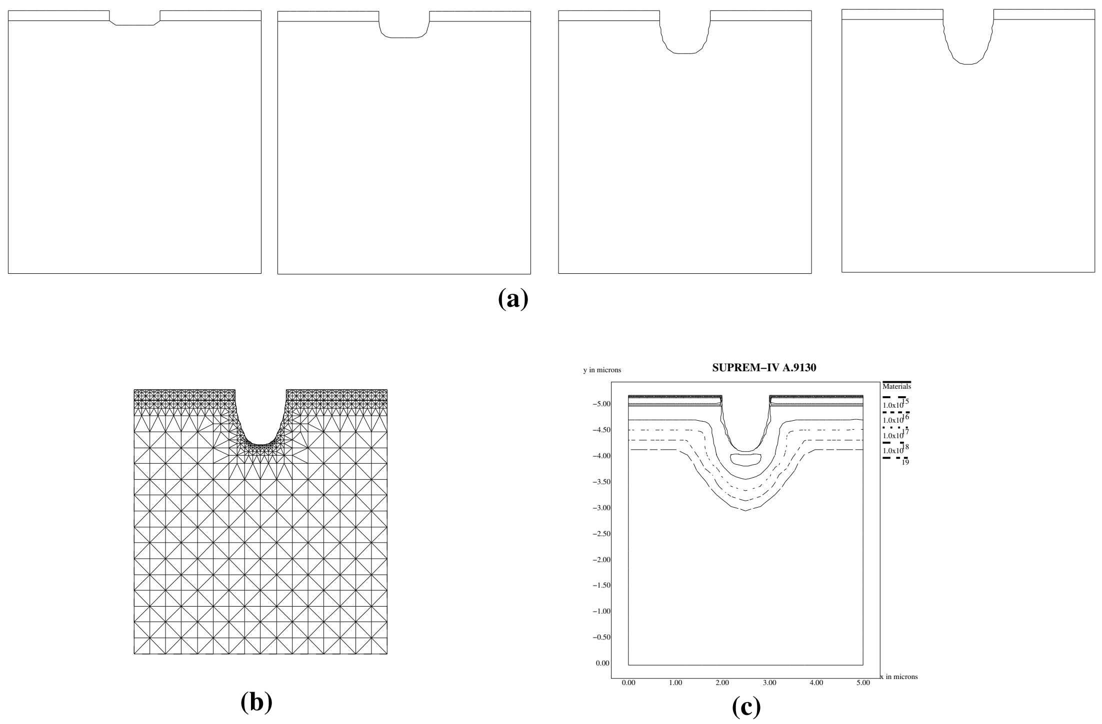


Figure 5.10: Use of SPEEDIE and SUPREM with *Forest* for etching, implantation and diffusion simulations. (a) Geometry frames during etching simulation. (b) Regeneration of volume grid after etch simulation. (c) Contour plots from boron implantation and drive-in simulations using SUPREM-IV.

The grid server moves all the grid nodes and repairs the grid by regeneration. The simulation then continues with the next time-step.

Figure 5.10 shows the linking of SPEEDIE and SUPREM-IV based on common use of geometry/mesh/grid services. After the geometry is initialized by the server, SPEEDIE uses the surface mesh server to compute a rate-based etch profile. A grid is then generated by the grid server and used by SUPREM-IV for subsequent ion implantation and diffusion steps. These heterogeneous tools, each with totally different representations and internal data structures, have been successfully integrated using the server methodology discussed above.

5.5 Future Extensions to 3D

The tool integration effort presented in this chapter has mainly concentrated on 2D applications and all server capabilities as described in Sections 5.3.1 through 5.3.4 and Section A have been implemented in 2D. In this section, 3D extensions are considered.

The data representation in all three *Forest* servers is fully 3D. The geometry representation can be used to manipulate 3D volumes as well as 2D regions. The mesh and grid information models are also dimension independent. Elements of arbitrary dimension linked to an arbitrary number of grid points are supported. However, unlike the SWR work, the servers in this integration strategy are defined *not* merely in their ability to represent data in various forms or dimensions but in terms of their functionality and interactions with one another and with TCAD applications.

While server-server and server-application interactions delineated in Section 5.3 are directly applicable in 3D, the limiting factor in 3D is the required server functionality. Several functions need to be implemented. For the geometry server, validity and repair operations are the most critical. Despite the complexity, however, an efficient $O(N \log N)$ loop detection and removal algorithm in 3D has been reported [24]. For the grid and mesh servers, generation and adaptation is the most critical. 3D grid generation algorithms specialized for TCAD applications have also been published [13]. Thus, the integration strategy discussed here can be and is being implemented in 3D. Moreover, the server API listed in Appendix A can be used *without* modification.

5.6 Summary and Conclusion

A new TCAD tool integration strategy focusing on server models is presented. Three servers - geometry, mesh and grid - capable of representing and providing data for the various TCAD tools are defined. Minimum server functionality is specified for tool integration. The servers consist of core modules and support utilities. The grid server, for example, is built around a core grid generator. Functional abstraction is used to provide access to the data within a server. Different core modules (grid generators, geometry modelers, etc) can thus be easily used within the integration framework owing to the simplified procedural interface.

For grid based applications, the onus of grid generation, adaption, and movement is relegated to the grid server. Applications are presented with function calls for grid and solution data access and update. For surface mesh based solvers, the functions of mesh generation and movement are encapsulated within the surface mesh server. Functions that mediate between geometric, mesh and grid operations are also defined as part of the integration framework. Successful implementation of this integration strategy is demonstrated with the integration of SUPREM-IV, SPEEDIE, a new error estimator, and Boolean geometric applications, each supported with the grid, mesh and geometry modules of *Forest*.

Chapter 6

Conclusion

This dissertation covers techniques to represent and manipulate three different views of the semiconductor wafer - geometry, surface mesh and grid. Manipulation algorithms in 2D, as implemented in the *Forest* program, are also discussed. The use of these algorithms and information models in an integrated heterogeneous simulation environment is presented. Different simulation programs like SUPREM-IV, SPEEDIE, and other new applications link with *Forest* for full process simulation.

The geometry is represented using a wire frame boundary representation model. Algorithms to perform Boolean etching and deposition operations are presented and analyzed. Utility algorithms to manipulate the geometry are also presented and analyzed. The complexity of all the algorithms is linear, provided that a double hierarchy is implemented in the boundary representation model. If only a single hierarchy is used, then some of the algorithms are shown to be of quadratic complexity. This trade-off between computational efficiency and memory usage is discussed. These algorithms are shown to be sufficient to build an approximate representation of the wafer after processing.

Several algorithms to support boundary motion are presented. These algorithms enhance the capability of topography programs like SPEEDIE. Hidden layers are accurately represented so that etching simulations maintain accuracy. Several difficulties encountered during geometry are also discussed. Among these, loop formation is the most important. Algorithms to detect and excise such loops are presented and analyzed. Boundary loop formation during topography simulation is the biggest drawback of using the boundary representation model. Alternative schemes need to explored.

For thermal simulation, one of the most difficult tasks for the user is the specification of a simulation grid. The problem is compounded by dopant redistribution and geometry evolution - the grid requirements change as the simulation proceeds. Thus, automatic 2D grid generation and adaptation algorithms are needed. Algorithms developed as part of this work to provide such functionality are presented. A quadtree approach is used to divide the problem of generating a grid over a complex geometry into a large set of sub-problems of generating grids over quadrilaterals. The algorithm is shown to generate grids of high quality. Grid adaptation during diffusion, implantation and oxidation simulations illustrate the power of the quadtree approach. These algorithms greatly increase the class of structures that can be simulated.

Since full process simulation requires the use of many different programs, each with its own wafer state representation, a method for using all these programs seamlessly is needed. A new tool integration strategy that fulfills this need by focusing on minimal server models is presented. Results of integrating SUPREM-IV, SPEEDIE, a new error estimator and new Boolean geometry applications show that the approach is versatile. Integration does not impose any implementation details on applications or servers. Thus, both new and existing programs can be integrated easily.

6.1 Future Work

6.1.1 3D

Several topics presented in this dissertation warrant further research. The most obvious of these is the extension to 3D. Representation of geometry, surface mesh and grid in 3D is relatively straight forward. However, geometry manipulation, grid generation, grid adaptation and grid movement are extremely difficult. Recent efforts have yielded very promising results. Successful geometry representation and manipulation has been reported for virtual processing [51]. Robust surface movement and repair for

3D physically based topography simulation has been achieved [24]. Grid generation has also been demonstrated using the octree method [13]. However, the integrated use of all these algorithms, in concert, to facilitate full process simulation has not been accomplished (yet).

6.1.2 Geometry Representation

As discussed in Chapter 3 in relation to topography simulation, and again in Chapter 4 in relation to oxidation simulation, the boundary representation geometry model is inadequate for moving boundary simulation. Operations of delooping, and crash removal non-physically change the geometry, thereby compromising mathematical rigor in the simulation. Furthermore, extension of many of these operations to 3D will be extremely difficult. Thus, alternative techniques for representing the geometry should be explored. Some work has already been published on using a "boundary free" representation for topography simulation [25]. Applying similar techniques to oxidation simulation will be very useful and will obviate the need for many of the algorithms presented in this dissertation.

However, incorporation of a boundary free representation into a heterogeneous simulation environment with several programs requiring different representations poses several questions. Which representation is the native representation from which all the others are derived? Further work and experimentation with different native representation strategies is required to elucidate the trade-offs involved.

Moreover, is the boundary free representation part of the geometry server or should it be part of the grid server? Since the boundary free representation requires a grid, one should incorporate it into the grid server. The division between grid and geometry servers discussed in this dissertation will then need to be removed. It is not clear how the integration of the two servers will affect the minimal server interactions discussed in Chapter 5.

6.1.3 Grid and Surface Mesh

Several aspects of grid generation also warrant further research. One of the most critical is the grid discretization of small dimension regions. In semiconductor devices, thin structures are commonplace - gate oxides are 100Å thick but are roughly 1 micron long. If equilateral triangles are used in the gate oxide, then a large number of triangles are needed to fill the length direction. In 2D, the large number of triangles in one direction is not a big concern. In 3D, however, with two dimensions that are large, the number of elements increases dramatically. Anisotropic grids need to be used to reduce the number of elements. However, as discussed in Chapter 4, anisotropic grids using the tree approach may contain both small angles and will not satisfy the *strict* Delaunay condition. Further processing to satisfy the Delaunay condition will be required.

Error estimation for surface mesh and grid adaptation is another rich area of research. For topography simulation, the mesh needs to be adapted to insure accuracy in boundary motion. As discussed in Chapter 3, a fine mesh generally produces more realistic profiles. However, the threshold for mesh density should be based on error estimates from the solution of the governing differential equations. Since both the species flux calculations and the velocity calculations contribute to the error, an error estimate for both calculations is required. Surface mesh elements with large error can then be refined and the total error in the simulation can be determined.

For thermal simulation, an ad hoc grid adaptation criterion is used as discussed in Chapter 4. A rigorous mathematically based criterion should be used instead. Error estimators need to be developed for both the diffusion and oxidation equations. For the diffusion equations, adaptation could based on a number of different criteria - error in the potential, error in the dopant concentrations, or error in the point defect concentrations. While some work has been reported for the diffusion equations, similar results have not been reported for oxidation simulation.

6.1.4 Tool Integration

Several challenges still exist in tool integration. In this dissertation, tool integration is discussed from the point of view of manipulating a common semiconductor wafer state (back end integration). However, an equally important integration question is in user interaction with each of the tools (front end integration). Historically, different tools have had different user interfaces, each requiring a set of keywords and commands. It is not clear how one tool can execute commands from another tool. Fortunately, the problem of front end integration has been solved by the Tk/Tcl scripting language [38]. Tk/Tcl or a similar scripting language should be used to implement all TCAD applications and servers.

Yet another interesting aspect of tool integration is in implementing a true client-server architecture for the servers. Using the X-windows server as an analogy, the geometry, grid, and surface mesh servers should run in memory as independent processes that are queried and updated as needed by the applications. Since it is impossible to compile all applications programs together into one executable, such an implementation is required for true tool integration.

Appendix A

Minimal Server Procedural Interface

In Chapter 5, minimal servers are defined to facilitate tool integration among the various TCAD applications. It is also shown that by using the minimal server methodology, existing and new tools can be integrated rapidly. In this Appendix, the procedural interface to the minimal servers is fully documented[1].

One of the most critical aspects of an integration environment is its procedural interface (API). The interface must be powerful enough to accommodate the needs of many application programs. Yet it must be simple enough to allow different geometry and grid programs to comply to the interface. The poorly designed procedural interface of the original SWR 1.5 system was one of its major failings. The original SWR C++ interface specifies the full hierarchy of objects within both the geometry and field (grid) servers. Moreover, the objects are dereferenced and very specific methods are required for each object. Thus, each object and corresponding class has to exist in the geometry and grid programs. Interfacing a server with a different internal information model is extremely difficult.

The functional interface of the three servers of the *Forest* program avoids all these pitfalls. The interface grew out of the experience of an ill-fated attempt at providing an SWR 1.5 compliant interface to the functions and data within *Forest*. The minimal *Forest* interface is general. It is also dimension independent - equally applicable to

[1] #defines and numbering conventions are found in header files and are not documented here due to space considerations. They can be found in [55].

3D as well as 2D servers. The underlying information model and implementation of the servers is hidden as much as possible by functional abstractions that use and/or return *handles* to objects.

While *handles* in the API are conceptualized as pointers, the pointers are never dereferenced allowing bindings for different computer languages. If the servers are implemented in FORTRAN for example, the "iterator" functions return integers that represent array indices as *handles* to objects. An application can then query the server for properties of the object by passing the integer handle to a server query function. The query function looks up the object in the data array and returns the desired properties. Thus, the interface does not require a C++ implementation of the servers. Since objects are never dereferenced, the objects need to exist in the server in only a virtual sense - a strict object-oriented design. Morever, the functions that operate on the objects are described at a very high level. Implementation details and utility methods for the objects are not specified.

The interface specification in this Appendix is divided into the logical geometry, grid, and surface mesh sections. Unless otherwise noted, functions return TRUE on successful completion and FAIL when an error occurs. However, functions that return *handles* return NULL on error conditions.

A.1 Attribution

A common string attribution mechanism is provided for all objects in all servers. Applications specify the name of an attribute, and the attribute itself, which is stored as a character string. Three functions are provided to "set," "get" and "remove" attributes:

- void *object***SetAttribute**(*object* *, char *attrName, char *attrVal)
- char **object***GetAttribute**(*object* *, char *attrName)
- void *object***RemoveAttribute**(*object* *, char *attrName)

The *italicized* object name is replaced by one of the server objects: **Point**, **Edge**, **Boundary**, **Region**, **Surface**, **Volume**, **Geometry**, **GridPoint**, **Element**, **Grid**,

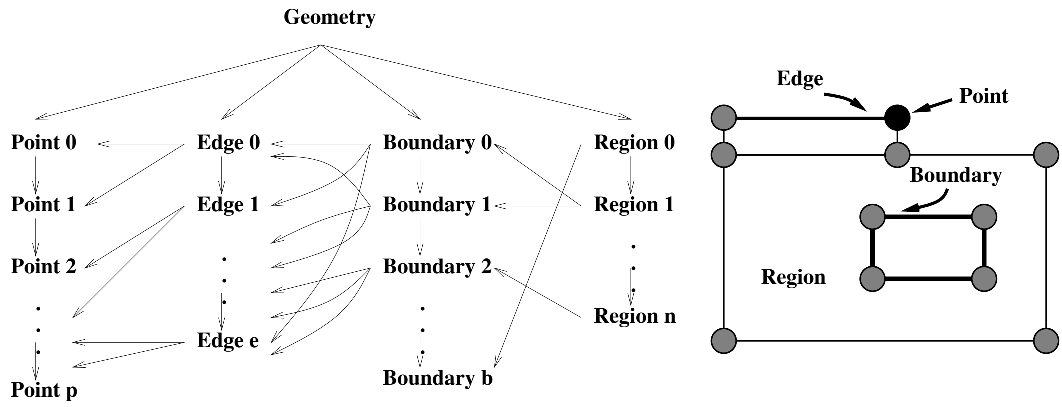


Figure A.1: Data structures required to represent a geometry using the boundary representation model in 2D. The geometry is a collection of regions that are bounded by boundaries. Boundaries are closed loops consisting of edges. Edges are directed line segments defined by points.

MeshPoint, or **SurfaceMesh**. Attributes are persistent - an attribute set by one application is available to all applications. Similarly, an object numbering scheme is also provided to set and get integer attributes.

- void *object***SetNumber**(*object* *, int number)
- int *object***Number**(*object* *)

A.2 Geometry Server

In this section, functions are defined to access and manipulate the boundary representation geometry model discussed in Chapter 2. Figure A.1 repeats the geometry information model. Since most geometry applications change the geometry, the geometry server has all the functions needed to construct a geometry.

A.2.1 Geometry Functions

Objects may be added and deleted using functions of the form shown below:

- *object* ***GeometryAdd***object*(Geometry *, *object* *)

- void **GeometryDelete***object*(Geometry *, *object* *)

The *italicized* object name is replaced with one of: **Point**, **Edge**, **Boundary**, **Region**, **Surface**, or **Volume**.

Several functions are used to manipulate the geometry. **GeometryRead** and **GeometryStore** facilitate persistent storage of geometries. **GeometryReplace** replaces the server's geometry with a new geometry. **GeometryMove** is used to simulate material movement and serves as an in-place operation. The geometry server provides the **GeometryMaxDt** function to return a maximum time step so that an application can anticipate the geometry becoming invalid. **GeometryValid** and **GeometryRepair** are used to validate and to repair an invalid geometry. The interfaces for these functions are as listed below:

1. Geometry ***GeometryRead**(int *numFilenames*, char ***filenames*)
2. int **GeometryStore**(Geometry *, int *numFilenames*, char ***filenames*)
3. int **ReplaceGeometry**(Geometry **oldg*, Geometry **newg*)
4. int **GeometryValid**(Geometry *)
5. int **GeometryMove**(Geometry *, double *time*)
6. double **GeometryMaxDt**(Geometry *, double *timeLimit*)
7. int **GeometryRepair**(Geometry *)

The following functions represent sequencers over all the points, edges, boundaries, regions, surfaces or volumes in the geometry:

1. **ForEachPointInGeometry**(Geometry *, Point *)
2. **ForEachEdgeInGeometry**(Geometry *, Edge *)
3. **ForEachBoundaryInGeometry**(Geometry *, Boundary *)
4. **ForEachRegionInGeometry**(Geometry *, Region *)

```
#include ``Point.h´´
#include ``Geometry.h´´

void MyFunction(Geometry *mygeom)
{
 Point *mypoint;
 double xcoord_sum = 0.0;

 ForEachPointInGeometry(mygeom, mypoint)
   {
    xcoord_sum += PointCoord(mypoint, X);
   }

 cout << ``total = ´´ << xcoord_sum << ``\n´´;
}
```

Figure A.2: Code fragment illustrating use of iterators.

5. **ForEachSurfaceInGeometry**(Geometry *, Surface *)

6. **ForEachVolumeInGeometry**(Geometry *, Volume *)

The application passes a geometry handle argument and the function returns a handle to the desired object. Figure A.2 illustrates the use of these functions.

A.2.2 Component Functions

Standard functions that operate on geometry objects are documented below. For points, functions to set and get the coordinates, velocities and displacements are provided. For edges, access (set and get) to the end points is provided. For boundaries, access functions are provided for the boundary edges. Similar functions are provided for regions, surfaces and volumes. The nomenclature used is self-explanatory. Utility functions to support geometric point objects:

1. double **PointCoord**(Point *, int *axis*) // return the coordinate

2. double **PointVelocity**(Point *, int *axis*) // return the velocity

3. double **PointDisplacement**(Point *, int *axis*) // return the displacement
4. void **PointSetCoord**(Point *, int *axis*, double *val*) // set coordinate
5. void **PointSetVelocity**(Point *, int *axis*, double *val*) // set velocity
6. void **PointSetDisplacement**(Point *, int *axis*, double *val*) // set displacement

Utility functions to support geometric edge objects:

1. Point ***EdgeStartPoint**(Edge *) // return the two points
2. Point ***EdgeEndPoint**(Edge *) // that make the edge
3. void **EdgeForm**(Edge *, Point **startp*, Point **endp*) // construct edge from 2 points

Utility functions to support geometric boundary objects:

1. int **BoundaryNoofEdges**(Boundary *) // how many edges on the boundary?
2. Edge ***BoundaryEdge**(Boundary *, int)
3. void **BoundaryForm**(Boundary *, int *numEdges*, Edge ***edgeptrs*) // construct boundary from edges

Utility functions to support geometric region objects:

1. int **RegionNoofBoundaries**(Region *) // how many boundaries?
2. Boundary ***RegionBoundary**(Region *, int)
3. void **RegionForm**(Region *, int *numBd*, Boundary ***bdPtrs*) // construct regions from boundaries

Utility functions to support geometric surface objects:

1. int **SurfaceNoofBoundaries**(Surface *) // how many boundaries?
2. Boundary ***SurfaceBoundary**(Surface *, int)

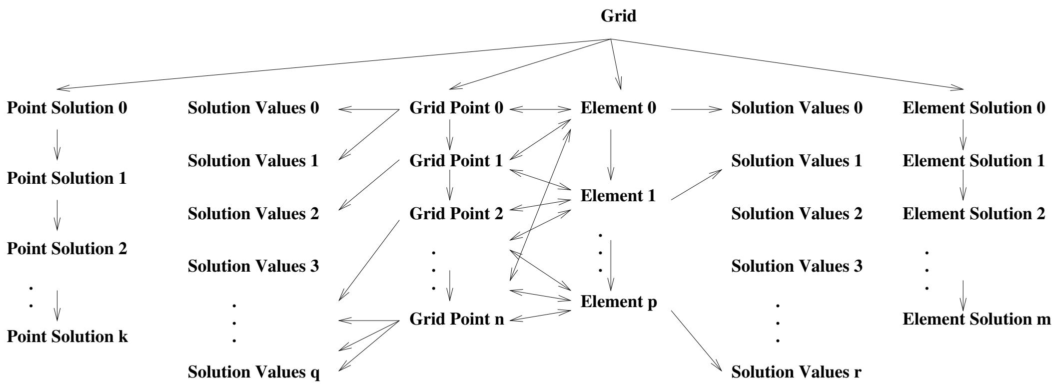


Figure A.3: Information model of the *Forest* grid server. Grids are a collection of elements and grid points with solutions and solution values defined over them. Note that solution values can be accessed only via its parent element or grid point.

3. void **SurfaceForm**(Surface *, int *numBd*, Boundary ***bdPtrs*) // construct surfaces from boundaries

Utility functions to support geometric volume objects:

1. int **VolumeNoofSurfaces**(Volume *)
2. Surface ***VolumeSurface**(Volume *, int)
3. void **VolumeForm**(Volume *, int numSf, Surface **sfPtrs) // construct volume from surfaces

A.3 Grid Server

Grids are defined as objects with lists of elements, grid points, solutions, and connectivity between grid points and elements. Solutions can be defined on the elements or on the grid points. The grid server information model is shown in Figure A.3. The mesh server information model is similar.

A.3.1 Grid Level Functions

Several functions are defined to operate on grids. **GridGenerate** generates a grid of type *gridtype* over a geometry. All applications must call **GridGenerate** before attempting to access a grid. Grid adaption is performed based on solution names representing adaption criteria passed to **GridAdapt**. Grid motion is supported by **GridMove** whose arguments are velocity solution names. **GridMaxDt** determines the maximum time step that can be taken before the grid becomes invalid. The arguments specify the solution names of the velocity field and the maximum allowed time step. **GridRepair** is used to re-validate an invalid grid. **GridConstruct** is used to construct a grid. The function arguments consist of coordinate values, element connectivity, element regions, and boundary conditions. **GridAddSolution** and **GridRemoveSolution** add and remove solution sets from the grid server. All solutions are defined to be of type double precision. *Solutiontype* specifies whether the solution is defined over grid points or over elements. Interpolation types are specified by the argument *interpolationtype*. The interfaces to these grid functions are as follows:

1. Grid ***GridGenerate**(int *gridType*, Geometry *)
2. int **GridStore**(Grid *, int *numFilenames*, char ***filenames*)
3. int **GridValid**(Grid *)
4. int **GridAdapt**(Grid *, int *numNames*, char ***solutionName*)
5. int **GridMove**(Grid *, double *time*, char **xvelName*, char **yvelName*, char **zvelName*)
6. double **GridMaxDt**(Grid *, char **xvelName*, char **yvelName*, char **zvelName*, double *maxTime*)
7. int **GridRepair**(Grid *)

8. int **GridConstruct**(Grid *, int *numPoints*, double **xcoord*, double **ycoord*, double **zcoord*, int *numElements*, int **connectivity*, int **numNodesPerElement*, char ***elementRegion*, char ***boundaryCondition*)

9. int **GridAddSolution**(Grid *, char **solutionName*, int *solutionType*, char **interpolationType*)

10. int **GridRemoveSolution** (Grid *, char **solutionName*)

The following functions represent methods to sequence over all the points, elements, point solutions, or element solutions in the grid:

1. **ForEachGridPoint**(Grid *, GridPoint *)

2. **ForEachElement**(Grid *, Element *)

3. **ForEachPointSolution**(Grid *, char *)

4. **ForEachElementSolution**(Grid *, char *)

They are similar to the sequence functions in the geometry server.

A.3.2 Grid Point Functions

Grid points are defined to have the functions listed below:

1. double **GridPointCoord**(GridPoint *, int *axis*)

2. **ForEachRegionAtGridPoint**(GridPoint *, char **regName*)

3. void **GridPointSetSolution**(GridPoint *, char **regName*, char **solName*, double *solValue*)

4. double **GridPointGetSolution**(GridPoint *, char **regName*, char **solName*)

GridPointCoord returns the desired coordinate value. The second function sequences through the regions at a point, similar to the ones described above. **GridPointSetSolution** and **GridPointGetSolution** set and get a solution value at the grid point associated with the specified region.

A.3.3 Element Functions

The following are the functions defined over elements:

1. GridPoint ***ElementGridPoint**(Element *, int *gridPointNumber*)
2. Region ***ElementRegion**(Element *)
3. Element ***ElementNeighbor**(Element *, int *neighborNumber*)
4. int **ElementType**(Element *)
5. char ***ElementFaceBoundaryCondition**(Element *, int *face*)
6. void **ElementSetSolution**(Element *, char **solName*, double *value*)
7. double **ElementGetSolution**(Element *, char **solName*)

ElementPoint returns a handle to a grid point for the specified element. The *gridPointNumber* argument is the local point number. **ElementRegion** returns the element's region. **ElementNeighbor** returns a handle to a neighbor element. **ElementType** returns the element's type, for example a 3 noded triangle or a 20 noded hexahedron. The boundary condition along the specified face of the element is returned by **ElementFaceBoundaryCondition**. Finally, **ElementSetSolution** and **ElementGetSolution** set and get specified solutions on the element.

A.4 Surface Mesher Interface

The interface for a surface mesh server is the same as for the grid server. The functions have a one-to-one correspondence. The naming convention is different however. **Grid** in all the names is replaced with **SurfaceMesh**; **GridPoint** is replaced with **MeshPoint**.

Bibliography

[1] ACIS Spatial Technology Inc., Boulder, CO, 1992.

[2] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA. 1974.

[3] B. Baccus, D. Collard and E. Dubois. "Adaptive Mesh Refinement for Multilayer Process Simulation Using the Finite Element Method." *IEEE Trans CAD*, vol 11, no. 3, pp 396-403, March 1992.

[4] P. L. Baehmann, S. L. Wittchen, M. S. Shephard, K. R. Grice and M. A. Yerry. "Robust, Geometrically based, Automatic Two Dimensional Mesh Generation." *Intl J Num Meth Eng.* vol.24, pp 1043-1078 (1987).

[5] R. Bank, *PLTMG User's Guide.* 5th. ed. Technical Report. Department of Mathematics, Univ. of Calif. San Diego, 1988.

[6] D. J. Benson, "Computational methods in Langrangian and Eulerian hydrocodes." *Computer Methods in Applied Mechanics and Engineering.* vol.99, no.2-3, pp 235-94. Sept. 1992.

[7] M. Bern, D. Eppstein and J.R. Gilbert. "Provably Good Mesh Generation." *Proc. 31st IEEE Symp. Foundations of Computer Science* (1990) 231-241.

[8] T. D. Blacker and M. B. Stephenson. "Paving: A new approach to automated quadrilateral mesh generation." *Intl. J. Num. Meth. in Eng.* Vol. 32, pp 811-847. (1991)

[9] D. S. Boning, M. L. Heytens, and A. S. Wong, "The Intertool Profile Interchange Format: An Object-Oriented Approach." *IEEE Trans CAD*, vol 10, no. 9, pp. 1150-1156, Sept. 1991.

[10] D. Borrione, R. Piloty, D. Hill, K.J. Lieberherr, and P. Moorby. "Three decades of HDLs. II. Conlan through Verilog." *IEEE Design and Test Of Computers*, vol.9, no.3, pp. 54-63, Sept. 1992.

[11] G. F. Carey, "Grid Generation, Refinement and Redistribution." The Univ. of Texas at Austin Press. Austin, TX. 1992.

[12] G. R. Chin, "Circuit Oriented IC Technology Simulation - Information Models and Integration Frameworks." Ph.D. Thesis. Stanford University, Department of Electrical Engineering. June 1992.

[13] P. Conti, N. Hitschfeld, and W. Fichtner, "Omega -an octree-based mixed element grid allocator for the simulation of complex 3-D device structures." *IEEE Trans CAD*, vol. 10, no. 10, pp. 1231-41, Oct. 1991.

[14] C. H. Corbex, A. F. Gerodolle, S. P. Martin and A. R. Poncet, "Data Structuring for Process and Device Simulations." *IEEE Trans CAD*, vol. 7, no. 4, pp. 489-500, Apr. 1988.

[15] K. Deljouie-Rakhshandeh, "An Approach to the generation of triangular grids possessing few obtuse triangles." *Intl. J. Num. Meth. in Eng.* Vol. 29, pp 1299-1321. (1990)

[16] S. G. Duvall, "An Interchange Format for Process and Device Simulation." *IEEE Trans CAD*, vol. 7, no. 7, pp. 741-754, Jul 1988.

[17] P. R. Eiseman, "Adaptive Grid Generation." *Computer Methods in Applied Mechanics and Engineering.* 64(1987) pp. 321-376.

[18] F. Fasching, C. Fischer, S. Selberherr, H. Stippel, W. Tuppa and H. Read, "A PIF Implementation for TCAD Purposes." *Proc. 4th Intl Conf on Simulation of Semiconductor Devices and Processes.* Sept. 1991, pp. 477-482.

[19] M. Fischetti and S. E. Laux, "Monte Carlo study of electron transport in silicon inversion layers." *IEDM'92 Technical Digest.* Dec. 1992. pp. 721-724.

[20] I. Fried, "Condition of Finite Element Matrices Generated from Nonuniform Meshes." *AIAA Journal.* vol. 10, no. 2, pp. 219-221. Feb 1972.

[21] M. D. Giles, D. S. Boning, G. R. Chin, W. C. Dietrich Jr., M. S. Karasick, M. E. Law, P. K. Mozumder, L. R. Nackman, V. T. Rajan, D. M. H. Walker, R. H. Wang, and A. S. Wong, "Semiconductor Wafer Representation for TCAD." *IEEE Trans CAD*, vol. 13, no. 1, pp. 82-95, Jan 1994.

[22] P. A. Gough, M. K. Johnson, P. Walker, and H. Hermans, "An Integrated Device Design Environment for Semiconductors." *IEEE Trans CAD*, vol. 10, no. 6, pp. 808-821, Jun 1991.

[23] Hamaguchi S. et al., "A shock Tracking Algorithm for Surface Evolution Under Reactive Ion Etching" *IBM RC* 18283 (80168) 8/28/92.

[24] J. J. Helmsen, E. W. Scheckler, A. R. Neureuther, and C. H. Sequin, "An efficient loop detection and removal algorithm for 3D surface-based lithography simulation." *Proc. of Intl. Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits.* June 1992, pp. 3-8.

[25] Z.-K. Hsiau, E. C. Kan, J. P. McVittie, R. W. Dutton. "Physical Etching/Depositon Simulation with Collision-Free Boundary Movement." IEDM Tech Digest. Dec. 1995.

[26] T. J. R. Hughes. *The Finite Element Method.* Prentice-Hall. Englewood Clifs, NJ. 1987.

[27] R. Ismail and G. Amaratunga. "Adaptive Meshing Schemes for Simulating Dopant Diffusion." *IEEE Trans. on CAD.* Vol. 9. No 3. March 1990. 276-289.

[28] R. C. Jaeger. *Introduction to Microelectronic Fabrication. Volume V. Modular Series on Solid State Devices.* Addison Wesley. Reading, MA. 1988.

[29] M. S. Karasick and D. Lieber, "Schemata for interrogating solid boundaries," *ACM 1991 Symposium of Solid Modeling Foundations and CAD/CAM Applications*, Austin, TX. 1991.

[30] M. E. Law, and R. W. Dutton, "Verification of Analytic Point Defect Models Using SUPREM-IV." *IEEE Trans on CAD*, vol. 2, no. 2, pp. 181-190, Feb 1988.

[31] M. E. Law, "FLorida Object Oriented Process Simulator Manual." Dept of Elect Eng, Univ. of Florida. 1994.
(WWW document: http://www.eel.ufl.edu/~law/manual/Intro.html)

[32] R. Malladi, J. Sethian, and B.C. Vemuri. "Shape modeling with front propagation: a level set approach." *IEEE Trans Patt Anal Mach Intel* vol. 17. Feb 1995. pp. 158-174.

[33] J. P. McVittie, J. C. Rey, and K. C. Saraswat, "Speedie User's Manual. Version 0.3." Stanford Univ., Stanford, CA. Mar 1991.

[34] J. P. McVittie, J. C. Rey, A. J. Bariya, M. M. IslamRaja, L. Y. Cheng, S. Ravi, and K. C. Saraswat, "SPEEDIE: A Profile Simulator for Etching and Deposition." *SPIE* Vol. 1392. pp. 126-138. 1990.

[35] S. Muller, K. Kells and W. Fitchner. "Automatic Rectangle-Based Adaptive Mesh Generation Without Obtuse Angles." *IEEE Trans CAD.* vol.11, no.7, pp 855-863. Jul 1992.

[36] W. G. Oldham, A. R. Neureuther, C. Sung, J. L. Reynolds, and S. N. Nandgaonakar, "A general simulator for VLSI lithography and etching processes: Part II-Application to deposition and etching." *IEEE Trans Electron Devices*, vol. ED-27, pp. 1455-1459, Aug 1980.

[37] J. O'Rourke, C.-B. Chien, T. Olson, and D. Naddor, "A new linear algorithm for intersection convex polygons," *Computer Graphics and Image Processing* **19** pp. 384-391 (1982).

[38] J. Ousterhout, *TCL and the TK Toolkit.* Addison Wesley. Reading, MA. 1994.

[39] J. Pelka, "Simulation of ion-enhanced dry-etch processes." *Proc. of SPIE - Advanced Techniques for Integrated Circuit Processing.* vol. 1392. Oct. 1990, pp. 55-66.

[40] L. Piegl and W. Tiller, "Curve and surface constructions using rational B-splines." *Computer Aided Design*, vol. 19, no. 9, pp. 485-98, Nov. 1987.

[41] M. R. Pinto, "Comprehensive Semiconductor Device Simulation for Silicon ULSI." Ph.D. Thesis. Stanford University Department of Electrical Engineering. Aug 1990.

[42] M. R. Pinto, D. M. Boulin, C. S. Rafferty, R. K. Smith, W. M. Coughran, Jr., I. C. Kizilyalli, and M. J. Thoma, "Three-dimensional characterization of bipolar transistors in a submicron BiCMOS technology using integrated process and device simulation." *IEDM'92 Tech Digest.* Dec. 1992, pp 923-926.

[43] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York. 1985. pp 286.

[44] A. Pruijmboom, C. E. Timmering and J. J. E. M. Hageraats. "18ps ECL-Gate Delay in Laterally Scaled 30GHz Bipolar Transistors." *IEDM'94 Tech Digest.* Dec. 1994, pp 825-828.

[45] C. Rafferty. "Stress Effects in Silicon Oxidation - Simulation and Experiments." Ph.D. Thesis. Dept. of Elect. Eng. Stanford University. 1989.

[46] Z. H. Sahul, R. W. Dutton, and M. Noell, "Grid and Geometry Techniques for Multi-Layer Process Simulation." *Proc. of 5th Intl Conf on Simulation of Semiconductor Devices and Processes*, Sept. 1993, pp. 417-420.

[47] Z. H. Sahul, E. W. McKenna, and R. W. Dutton. "Grid Evolution for Oxidation Simulation Using a Quadtree Based Grid Generator." *Proc. of Intl Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits.* June 1994, pp. 155-158

[48] Z. H. Sahul, K. C. Wang, Z.-K. Hsiau, E. W. McKenna, and R. W. Dutton. "Heterogeneous Process Simulation Tool Integration." *IEEE Trans Semi Manuf.* vol.9, no.1 pp. 35-58. Feb 1996.

[49] M. R. Simpson, "PRIDE: An Integrated Design Environment for Semiconductor Device Simulation." *IEEE Trans CAD.* vol. 10, no. 9, pp. 1163-1174, Sep 1991.

[50] P. Sutardja, "Finite-Element Methods for Process Simulation Application to Silicon Oxidation". PhD Thesis. U.C. Berkeley. May 1988.

[51] K. Wang, F. Rotella, T. Chen, D. Yang, A. Lee, Z. Yu, R. W. Knepper, J. Watt and R. W. Dutton, "Layout-based Extraction of IC Electrical Behavior Models". *IEDM'94 Technical Digest*, Dec 1994, pp. 209-212.

[52] R. H. Wang, A. Gabara and A. R. Neureuther, "BTU - Berkeley Topography Utilities for Linking Topography and Impurity Profile Simulations." *Proc. of Intl Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits.* Jun. 1992, pp. 225-230.

[53] A. S. Wong, and A. R. Neureuther, "The Intertool Profile Interchange Format: A Technology CAD Environment Approach." *IEEE Trans CAD*, vol. 10, no. 9, pp. 1157-1162, Sep 1991.

[54] C. Yang and M. D. Giles, "Architecture and Implementation of 3D field support for Semiconductor Wafer Representation." *Proc. of Intl Workshop on Numerical Modeling of Processes and Devices for Integrated Circuits.* June 1994, pp. 81-84.

[55] D. Yergeau, Z. Sahul, E. McKenna and R. Dutton, "HDF-Vset File Format for Persistent Storage of Semiconductor Simulation Data." Technical Report. Integrated Circuits Lab. Stanford University. Dec 1994. (WWW document: http://www-ee.stanford.edu/ee/tcad/programs/hdfvset.html)