Chapter 3

Spatial Decomposition And Quadtree Mesh

3.1 Introduction

In the previous chapter, we have discussed mesh requirements and the classification of different mesh schemes. To choose a good mesh generation scheme, a decision should be made based on the requirements imposed by an application. In this chapter, a new quadtree based mesh generation scheme is introduced, which is designed specifically for the semiconductor device simulation and to address various needs of TCAD applications. Mesh generation based on quadtree structure has been used for finite element methods (FEM) to solve PDE problems for more than a decade [YERR84, PERU89]. Recently quadtree based mesh schemes have been incorporated into software for both device and process simulations [CONT91, YANG92, SAHU93]. There are several advantages for quadtree based mesh generation schemes. First, the basic algorithm is simple and the tree data structure is efficient for many topological and geometric operations (such as finding neighbors) [SAME84]. Second, adaptive mesh refinement can easily be incorporated (by dividing a leaf quadrant or pruning a subtree). Third, the scheme can be integrated nicely with solid modelers because the same basic idea has been used to implement many efficient geometric operations.

Several issues must be addressed when adopting the quadtree technique for mesh generation. First, a non-balanced quadtree could produce ill-shaped triangles. Second, if the quadtree structure is strictly followed, some ill-shaped triangles could occur along the boundary of the geometry region [HO88, YANG93]. Various techniques have been proposed to resolve these shortcomings[ALUR91, MULL92]. In this chapter, two new techniques, namely the level control function and the δ-zone triangulation, are introduced. The level control function is used to generate a balanced quadtree. The δ-zone separates the boundary region from the interior region so that a more sophisticated triangulation scheme can be applied to prevent generating ill-shaped triangles. With these two enhancements, a quality mesh for both interior region and regions near the boundary can be achieved. These two techniques have been implemented in our quadtree based mesh generator. Figure 3.1 summarizes the computational flow of the proposed mesh generation process. In this chapter, we also show that the quadtree generated using the level control function is an optimal balanced quadtree with respect to a given mesh density distribution.

The rest of this chapter is organized as follows. Section 3.2 gives an overview of the spatial decomposition technique and the quadtree data structure. Section 3.3 describes the concept of the level control function and its application for generating a balanced quadtree. Section 3.4 details the procedures for interior quadrant triangulation. Section 3.5 provides a thorough discussion on the development of the δ-zone triangulation technique and the final section contains a short summary.

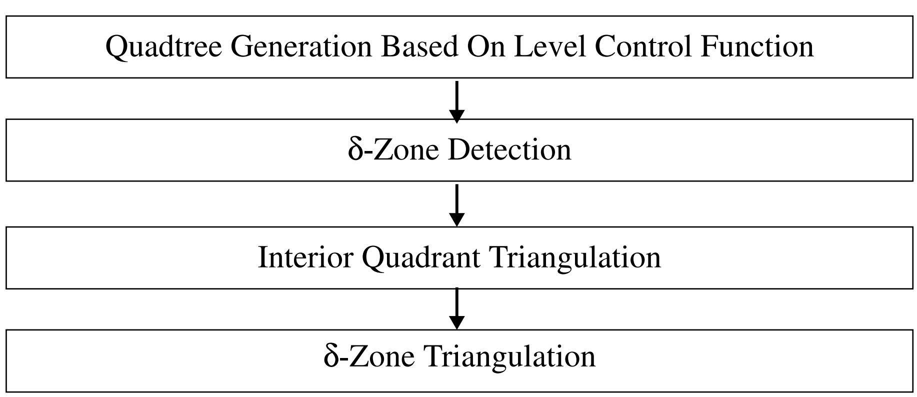


Figure 3.1 Quadtree mesh generation flow.

3.2 2D Spatial Decomposition

3.2.1 Quadtree - A 2D Spatial Decomposition Scheme

Hierarchical data structure coupled with recursive algorithm is a general method widely used in modern computing. Hierarchical data structure enables an elegant description for a complex model. The recursive algorithm originates from a simple but powerful problem solving technique of divide and conquer. The recursive algorithm also maps well onto modern computer architecture and compiler technology, where stack based memory storage is used for implementing function/procedure calls.

One type of hierarchical data structure is the region quadtree (or simply quadtree) for handling geometry related computational problems in 2D space. Since its introduction in the late sixties [WARN68], quadtree has become an increasingly important representation technique in the domains of computer graphics, CAD and computer vision. In this section, the basic concept of quadtree is presented, followed by a discussion of terminologies and related properties.

Given a bounded geometry region, we can enclose it with a square. If the square contains area outside the region, a subdivision is performed to create 4 quadrants. This process is recursively applied to each quadrant until the quadrant is either entirely contained in the region, completely disjointed from the region, or non-divisible. A quadrant is non-divisible if its dimension has reached a predefined lower bound.

In the quadtree representation, each quadrant is called a *quadrant node* (or simply a *node*). The square enclosing the entire region is also called the *root* node. The *degree* of a node is the number of its children. A child quadrant of a node is also called the *subquadrant* of the node. In this section, we always assume a quadrant is either divided into 4 subquadrants or it has no subdivision at all. Hence, each node is either of degree 4 or 0[1].

A *leaf* quadrant is a quadrant with none or less than 4 child quadrants. A leaf quadrant is called an *I-node* if the quadrant is entirely contained in the region and an *O-node* if the quadrant is completely disjointed from the region. For figures shown in this thesis, we use

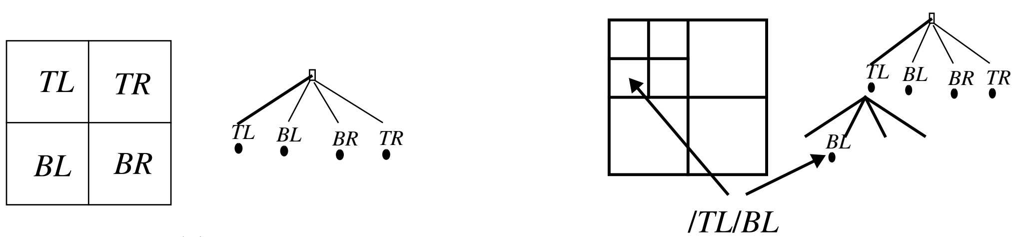
1. In Subsection 3.2.3, quadrants of degree other than 0 or 4 will be discussed.

a circle to denote an O-node, a dot for an I-node and a non-filled square for a non-leaf node.

The children of a node are arranged in a geographical order and, as illustrated in Figure 3.2(a), they are labeled as *TL* (top-left), *BL* (bottom-left), *BR* (bottom-right) and *TR* (top-right). This ordering enables us to address any quadrant node using a relative path mechanism similar to the Unix file system. For example, */TL/BL* refers to the quadrant which is the *BL* subquadrant of the *TL* subquadrant of the root (see Figure 3.2(b)).

Figure 3.3 shows an example of a geometry region, its spatial decomposition and the quadtree data structure. The root node corresponds to the square (p_0, p_1, p_2, p_3), which encloses the entire region. The square is divided into 4 quadrants, *TL* = (p_0, p_4, p_8, p_7), *BL* = (p_4, p_1, p_6, p_8), *BR* = (p_8, p_6, p_2, p_5) and *TR* = (p_7, p_8, p_5, p_3). The quadrants */BL* and */TR* are further divided since neither of these two nodes is an I-node nor an O-node. Similarly, the quadrant */BL/TR* is subdivided. The final quadtree has 4 non-leaf nodes, (*ROOT*, */BL*, */TR* and */BL/TR*), and 13 leaf nodes, of which 7 are I-nodes and 6 O-nodes.

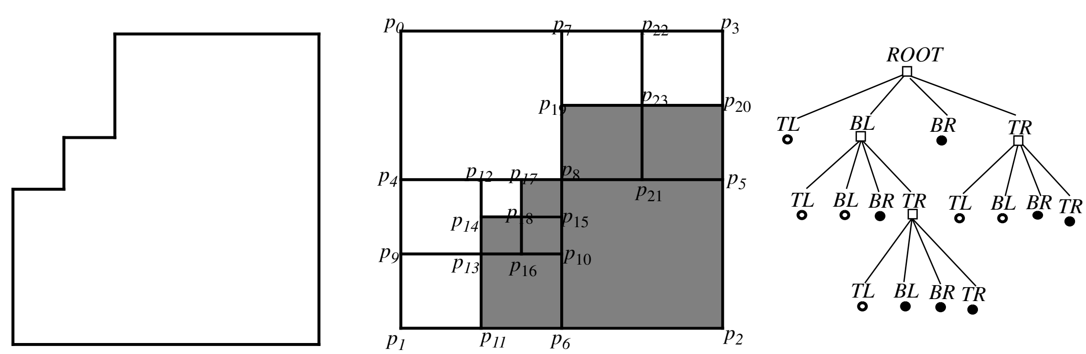
Since a quadtree representation is a tree structure, all conventional notations related to a graph or a tree can be applied. A *path* between a root and the quadrant Q is a contiguous sequence of edges and nodes $\{ROOT=Q_0,\ Q_1,...Q_i,\ Q_{i+1},\ Q_k=Q\}$ where Q_{i+1} is a subquadrant of Q_i. Each of Q_i, $i \neq k$, is called an ancestor of Q and Q is a descendant of Q_i. A *relative path* between a quadrant Q and its ancestor Q_i is the subsequence of the path



(a) (b)

Figure 3.2 Address of quadrants: (a) Address scheme.
(b) An example of quadtree.

without the first i quadrants, i.e. $\{Q_i, Q_{i+1},..., Q_k = Q\}$. In the example of Figure 3.3, the path for the node $(p_{14}, p_{13}, p_{16}, p_{18})$ is $\{ROOT, (p_4, p_1, p_6, p_8), (p_{12}, p_{13}, p_{10}, p_8), (p_{14}, p_{13}, p_{16}, p_{18})\}$ and the relative path between the node $(p_{14}, p_{13}, p_{16}, p_{18})$ and the node (p_4, p_1, p_6, p_8) is $\{(p_4, p_1, p_6, p_8), (p_{12}, p_{13}, p_{10}, p_8), (p_{14}, p_{13}, p_{16}, p_{18})\}$. Using the addressing scheme for denoting the relative location of the quadrants as discussed earlier, there is an one-on-one map between the path of a node and the address of the node. Within the context of this thesis, the path of a node and the address of a node are used interchangeably. In the example of Figure 3.3, the path for the quadrant node $(p_{14}, p_{13}, p_{16}, p_{18})$ can be written as */BL/TR/BL* and the relative path between $(p_{14}, p_{13}, p_{16}, p_{18})$ and (p_4, p_1, p_6, p_8) is *TR/BL*.



(a) (b) (c)

Figure 3.3 Quadtree representation of a geometry region: (a) Region. (b) Decomposition. (c) Quadtree.

For a given path $\{ROOT=Q_0, Q_1,..., Q_k=Q\}$, the length of the path is k and Q is said to be of level k. For a quadtree QT, the *level* of a vertex p in the quadtree is the maximum number of levels of all the quadrants containing p, i.e.

$$Level_{QT}(p) = max\ \{Level(Q) \mid p \in Q, Q \in QT\}.$$

A *full* quadtree is a tree with all its leaf quadrants having the same level, as shown in Figure 3.4.

3.2.2 Finding an Neighbor Quadrant

In this subsection, we discuss an algorithm for finding neighbor quadrants, which is employed extensively in generating unstructured mesh. The algorithm also demonstrates how the hierarchical data structure can facilitate an efficient and robust implementation of some complex computational operations.

The problem of finding a neighbor quadrant can be stated as follows:

> *Given a quadrant node and a direction (i.e. left, right, top and bottom), find the neighbor quadrant which is of size equal to or greater than the given node.*

The problem is not well posed if the size limitation is not specified. For the example in Figure 3.3, the quadrant */BR* has three left neighbor quadrants, */BL/BR*, */BL/TR/BR* and */BL/TR/TR*. To find the left neighbor for */BR* is an ambiguous task if the size constraint is ignored.

Definition 3.1 The *local address* of a quadrant is the relative path between the quadrant and its parent. Each local address can be represented by two coordinates X and Y where X

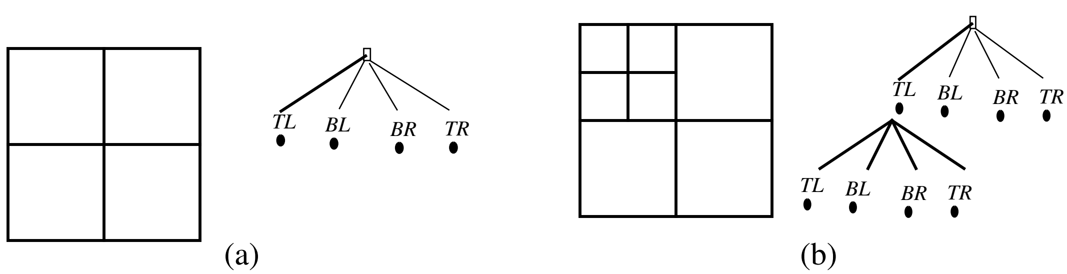


Figure 3.4 Examples of full and non-full quadtrees: (a) full quadtree. (b) non-full quadtree.

takes the value L (left) or R (right) and Y the value T (top) or B (bottom). For a relative path P, the *x-symmetry* is a new relative path P' which results from replacing all the X coordinates of P by its opposite values. Similarly, the *y-symmetry* is the relative path which results from replacing all the Y coordinates of P by its opposite values.

In the example of Figure 3.3, the local address for the quadrant (p_9, p_1, p_{11}, p_{13}) is *BL*. For the relative path *TR*/*BL* between the quadrant (p_4, p_1, p_6, p_8) and the quadrant (p_{14}, p_{13}, p_{16}, p_{18}), its x-symmetry is *TL*/*BR* and the y-symmetry *BR*/*TL*.

Now we are ready to present the algorithm for finding the neighbor quadrants. The idea is based on a 2-step tracing technique. First we perform a bottom-up ascending trace by following the parent links and then a top-down descending trace by following the child links [SAME88]. This procedure is illustrated by the example shown in Figure 3.5, where we want to find the right neighbor of the quadrant *A*. We first follow the ascending path to find the closest ancestor which contains *A* as one of its *L* (left) subquadrants. In this case, the closest ancestor is the quadrant *ROOT* and the ascending path is *P = /TL/TR/TR*. Next we find the x-symmetry of *P*, *P' = /TR/TL/TL*. From the ancestor quadrant *ROOT*, we follow the symmetry path *P'* and reach the node *B*, which is the right neighbor of *A*. This process is shown graphically by the directed path in Figure 3.5(b). The algorithm for finding neighbor quadrants can be summarized in the following three basic procedures: FindNeighborQuadrant, Ascending and Descending. The procedure Ascending finds the closest ancestor and the procedure Descending finds the neighbor node if it exists.

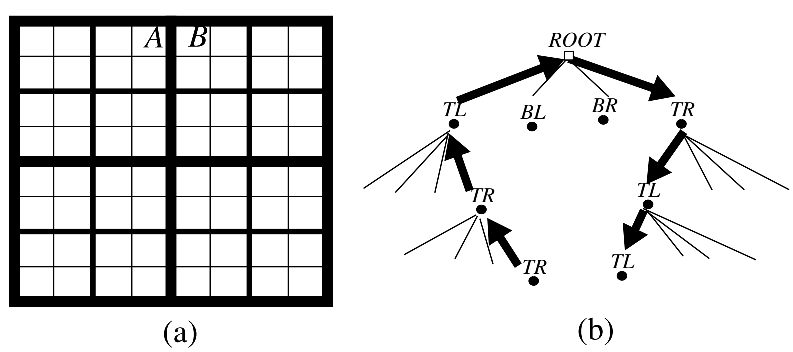


Figure 3.5 Procedure for finding neighbor node: (a) Spatial decomposition.
(b) Quadtree representation and direct paths.

```
PROCEDURE   FindNeighborQuadrant(q: Quadrant, d: Direction): Quadrant

    (* d takes value of L (left), R (right), T (top) or B (bottom) *)
    (* pathStack is a stack for storing local addresses along the path *)
    Initialize(pathStack);
    (* find the closest ancestor *)
    ancestor = Ascending(q, d, pathStack);
    (* trace the path to find the neighbor *)
    neighbor = Descending(ancestor, d, pathStack);
    RETURN neighbor;

END         FindNeighborQuadrant

PROCEDURE   Ascending(q: Quadrant, d: Direction, pathStack: Stack): Quadrant

    p = Parent(q);
    WHILE p ≠ NULL AND q's local address contains d DO
        a = LocalAddress(q);
        Push(a,  s);
        q = p;
        p = Parent(q);
    END;
    RETURN p;

END         Ascending
```

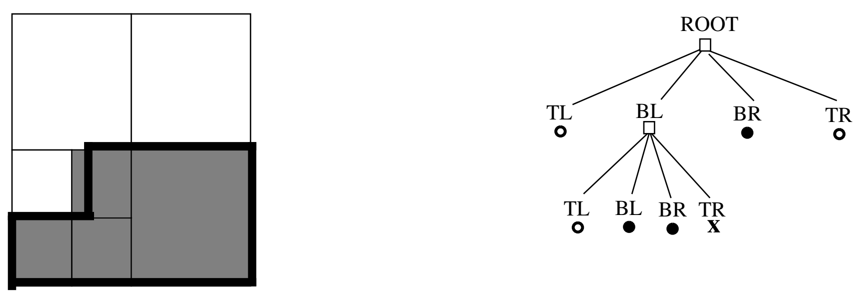
```
PROCEDURE   Descending(p: Quadrant, q: Direction, pathStack: Stack): Quadrant

    (* compute the symmetry for pathStack *)
    IF (d = L) OR (d = R) THEN
        pathStack = X-Symmetrize(pathStack);
    ELSE
        pathStack = Y-Symmetrize(pathStack);
    (* trace the path to find the neighbor quadrant *)
    WHILE NotEmpty(pathStack) AND NotLeaf(p) DO
        localAddress = Pop(pathStack);
        p = ChildNode(p, localAddress);
    END
    RETURN p;

END         Descending
```

3.2.3 Generalized Quadtree

In the basic quadtree representation, we always assume that a leaf quadrant is either an I-node or an O-node. This representation, however, is not sufficient to fully characterize the boundary of a geometry region. In the example of Figure 3.6, the leaf quadrant */BL/TR* is neither an I-node nor an O-node. Instead, a third type of leaf node, called B-node, is required. A *B-node* is neither an I-node nor an O-node, i.e. part of a B-node is inside of the geometry region and part outside. Any B-node must contain a section of the region's boundary. To distinguish it from the I-nodes and the O-nodes, a B-node is denoted by a cross in the figures.

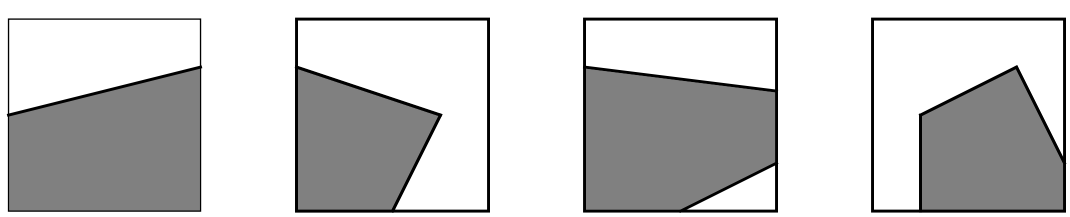
With B-nodes, any bounded geometry region can be described in an exact manner provided that we have a way to store the boundary information for each of the B-nodes. It is evident that the region contained in a B-node can be of any shape and requires a general data structure if no conditions are imposed. A more practical approach is to assume that the intersection of the geometry region and a B-node follows certain patterns [PERU89, CONT91]. In our approach, the patterns depicted in Figure 3.7 are assumed. If a B-node is of a shape other than these 4 patterns, the node will be further divided until the B-node can be represented using one of the patterns or a predefined minimum resolution is reached. In the latter case, a clipping onto one of these 4 patterns will be performed.

In addition to allowing the B-nodes in a quadtree, another generalization is made by relaxing the constraint on the degree of a quadrant. Instead of only permitting quadrants of degree 0 or 4, a quadrant can also have 1, 2, or 3 subquadrants. The implication of this generalization is that the geometry region is covered not only by leaf quadrants of degree



(a) (b)

Figure 3.6 Quadtree representation with B-nodes: (a) Region. (b) Quadtree.



(a) (b) (c) (d)

Figure 3.7 Boundary patterns for B-nodes: (a) One boundary edge.
(b) Two intersecting boundary edges.
(c) Two non-intersecting boundary edges.
(d) Three boundary edges.

0 but also leaf quadrants of degree 1, 2 or 3. In the latter case, the leaf quadrant is said to be *irregular.*

An irregular quadrant can be separated into two regions: the region with subquadrants and the one without. The region without subquadrants behaves much like a regular leaf quadrant and we call such a region the leaf region of an irregular quadrant. Examples of irregular quadrants are shown in Figure 3.8, where the leaf regions are shaded.

The concepts regarding to nodes, ancestors, descendants and paths discussed earlier can be extended to irregular quadrant nodes. From here on, unless stated otherwise, a generalized quadtree is assumed, i.e. a quadtree can contain both the B-nodes and the irregular nodes.

3.2.4 One Level Difference Rule and Balanced Quadtree

Mesh distribution patterns have a major impact on the quality of a mesh. It has been known that a drastic change of mesh density can cause ill-shaped triangles to be generated [BERN91]. For a quadtree based mesh generator, this observation implies that the tree should be kept well balanced between neighbor quadrants. In the example of Figure 3.9, the quadtree is not balanced because the node */TR/BL/BL* is of level 3 and */TL* is of level 1 and the two nodes are next to each other.

For an unbalanced quadtree, ill-shaped triangles such as (p_4, p_{13}, p_9) may be generated. To improve the mesh quality based on the given mesh density requirement, a further subdivision for both */TL* and */BR* is required as shown in Figure 3.10. To ensure well shaped triangles for a quadtree based mesh, a well known “one level difference” rule needs to be observed during the spatial decomposition process [BERN91].

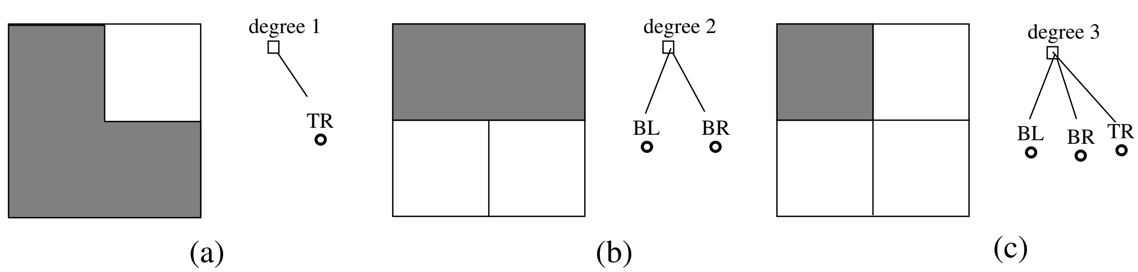


Figure 3.8 Examples of irregular quadrants: (a) The quadrant of degree 1. (b) The quadrant of degree 2. (c) The quadrant of degree 3.

Definition 3.2 A quadtree is said to satisfy *one level difference* rule if the level difference between two neighboring leaf nodes (either in the x direction or the y direction) does not exceed 1. Such a quadtree is also called a *balanced* tree.

To generate a balanced quadtree, the following brute force technique may be used [MULL92]. Each time a quadrant is subdivided, a validation of the rule will be performed. If violations occur, the neighbor quadrants will be divided. This process continues until all violations are eliminated. This approach is rather inefficient because of the following two reasons. First each quadrant may be visited many times in the decomposition process. Second each subdivision of a quadrant can cause a chain reaction in that the effect may be propagated to other quadrants. In the following section, a different technique is proposed to overcome these inefficiencies.

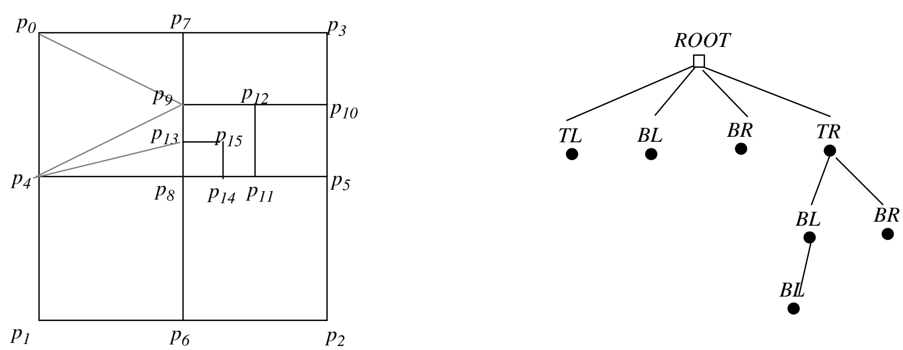


Figure 3.9 Non-balanced quadtree and ill-shaped triangles.

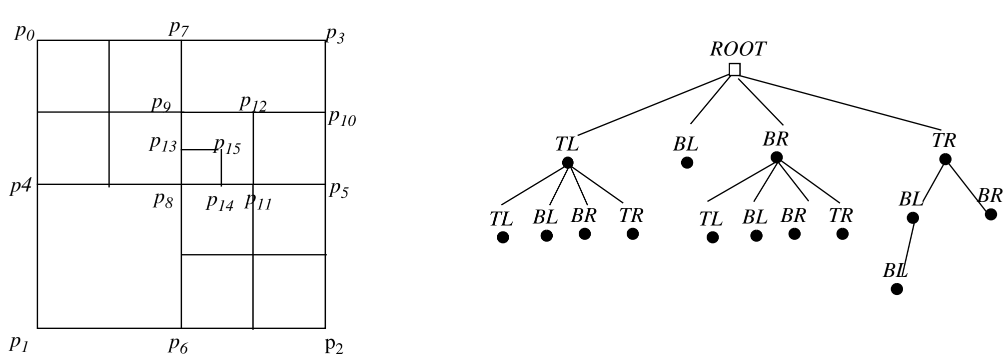


Figure 3.10 Quadtree after balancing.

3.3 Level Control Function

As noted in the previous discussion, to ensure a quality mesh, the quadtree needs to abide by the one level difference rule. This section introduces a new concept called level control function (LCF), which is used not only for controlling the mesh density distribution but also providing an efficient way to generate an optimal balanced quadtree.

This section is organized as follows. In 3.3.1, the level control function concept and related properties are discussed. In 3.3.2, we introduce a measure for quantifying the optimality of a balanced quadtree with respect to a level control function. A proof of the existence of optimal balanced quadtrees is also presented. In 3.3.3, we describe a computational algorithm for generating an optimal balanced quadtree and in 3.3.4, we verify the correctness of the algorithm by proving that the generated balanced quadtree is optimal.

Before we proceed the discussion of the level control function, we review two concepts which are used extensively in this section: (i) an *irregular* quadrant is the quadrant with 1, 2 or 3 subquadrants; and (ii) for a given quadtree *QT*, the level of a vertex p, $Level_{QT}(p)$, is defined to be the level of the deepest quadrant containing p. These two concepts are illustrated by the example shown in Figure 3.11. In the example, there are eight leaf quadrants. Among them, there is one irregular leaf quadrant, i.e. */TR*. The levels for vertices p_1, p_2, p_3, p_5 and p_6 are 1 and the levels for other vertices are 2.

The following conventions are adopted for notations used in this section. *QT, RT* are used for representing quadtrees; *Q, R* for quadrants in a quadtree; *p, q* for vertices of a quadrant; *m* for mid-edge vertex of a quadrant and *c* for the center vertex of a quadrant. $L(p)$ is the value of the level control function L at vertex p. $Level_{QT}(p)$ is the level of the

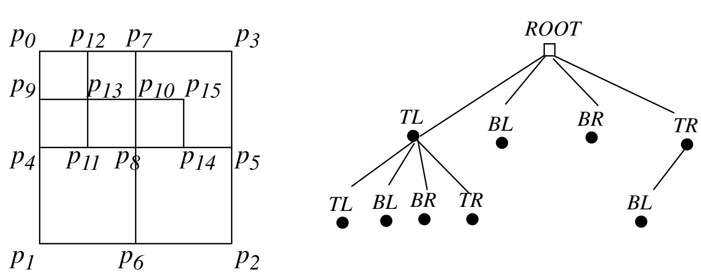


Figure 3.11 An example of irregular quadrants and vertex levels.

vertex p with respect to the quadtree QT. $Level_{QT}(Q)$ is the level of the quadrant Q with respect to QT. In some cases, subscript QT may be omitted if the quadtree QT is clear in the context.

3.3.1 Level Control Function

Definition 3.3 A *level control function* L is a non-negative and bounded integer function defined on a geometry region. Given a level control function L, the *distance* between a quadtree QT and L is defined to be

$$dist(L, QT) = \sum_{p} \left| L(p) - Level_{QT}(p) \right| \tag{3.1}$$

where the summation is for all vertices in the quadtree. A quadtree QT is *compatible* with L if the quadtree level equal to the L values on all vertices; in other words, the distance *dist(L, QT)* is 0.

In the example of Figure 3.12, the values of L are shown in parenthesis for each vertex. The quadtree is compatible with L for case (a) but not for case (b) because in the latter case, $Level(p) = 2$ but $L(p) = 4$.

A quadrant Q is said to be *abnormal* with respect to a level control function L if one of its mid-vertices (mid-edge vertices or center vertex) has the L value less than or equal to the quadrant's level. For the example in Figure 3.13, the quadrant */BL* is abnormal with

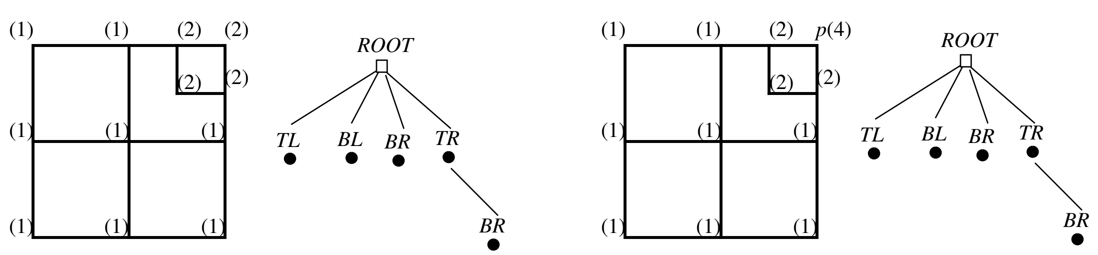


Figure 3.12 Level control function: (a) A compatible quadtree.
(b) A non-compatible quadtree.

respect to L because the level of the quadrant */BL* is 1 and the center vertex c of the quadrant */BL* has the L value equal to 1.

There is no compatible quadtree if a level control function has an abnormal quadrant. However, for a given level control function, we can always eliminate abnormal quadrants by modifying the values of the level control function without changing the mesh density distribution. In the following discussion, we assume that a level control function does not have any abnormal quadrant.

A level control function is directly related to the mesh density distribution. If we know the size of the root quadrant *RootSize*, the following formula gives the precise relation between the size of a quadrant Q and the level of the quadrant.

$$Size(Q) = \frac{RootSize}{2^{Level(Q)}}$$

There are many different ways to define a level control function. For semiconductor device simulation, a good candidate is the normalized gradient of the doping profile for initial mesh and error estimation function for mesh refinement.

Since the level control function is a good indicator for mesh density distribution, it would be ideal if the quadtree generation process based on a level control function guarantees a balanced and compatible quadtree. Unfortunately, this situation is not always

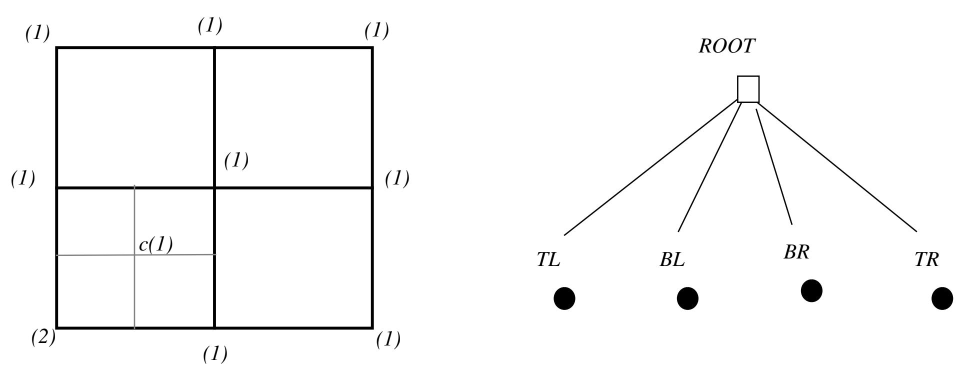


Figure 3.13 An abnormal quadrant.

true. In the example of Figure 3.14, for the given level control function (values shown in parenthesis), a compatible balanced quadtree does not exist.

Because a compatible balanced quadtree is not guaranteed, the challenge is to find a most compatible (optimal) balanced quadtree. To do so, we need to address three questions.

1. How do we quantify "optimality"?
2. Does an optimal balanced quadtree exist?
3. How do we computationally find the optimal balanced quadtree?

These questions are addressed in the following subsections.

3.3.2 Existence of Optimal Balanced Quadtree

Before we prove the existence of optimal balanced quadtrees, we first need to have a measure for quantifying "optimality". In set theory, a measure for optimality can be defined in terms of an ordering relationship.

Definition 3.4 A quadtree *QT* is said to be a *subtree* of another quadtree *RT* if each quadrant of *QT* is a quadrant of *RT*, denoted by $QT \subseteq RT$.

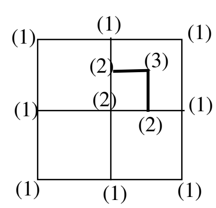


Figure 3.14 LCF with no compatible quadtree

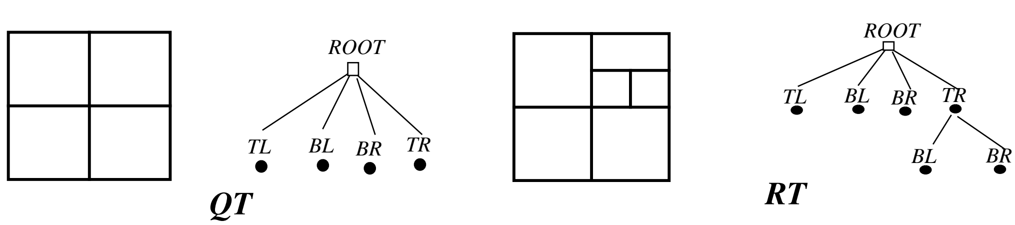


Figure 3.15 Order relationship for quadtree $QT \subseteq RT$.

Definition 3.5 A balanced quadtree *QT* is a *support* for a given level control function *L* if for each quadtree vertex *p*, the quadtree level at *p* is always less than the level control function's value at *p*, i.e.$L(p) \geq Level_{QT}(p)$. Likewise, *QT* is called a *cover* of *L* if the quadtree level at *p* is always larger than the level control function's value at *p*, i.e. $L(p) \leq Level_{QT}(p)$.

Before we prove the existence of an optimal balanced quadtree, we need to introduce an operation called *balance removal*, which makes the tree more balanced by eliminating a "bad" node. For a non-balanced quadtree, there is a pair of leaf quadrants which are neighbors to each other and the level difference between them is greater than 1. If we remove the quadrant of the higher level, the quadtree becomes more balanced. This removal operation is called the *balance removal* and is illustrated in Figure 3.16.

For any non-balanced quadtree *QT*, the balance removal operations can be applied repeatedly until the quadtree is balanced. We denote the resulting balanced quadtree as *balance(QT)*.

Lemma 3.1 Let RT' be a balanced quadtree of *RT* after applying the balance-removal operation. If *QT* is a balanced sub-quadtree of *RT*, i.e. $QT \subseteq RT$, then *QT* is also a sub-quadtree of the new balanced quadtree RT', i.e. $QT \subseteq RT'$.

Proof. Suppose R_1 and R_2 are a pair of leaf quadrants in *RT* which violates the one level difference rule and $Level(R_1) < Level(R_2)$. Let RT' be the result of removing R_2 from *RT*. Since *QT* is balanced, R_2 cannot be a quadrant in *QT*, which implies that *QT* is a sub-quadtree of RT'. ❑

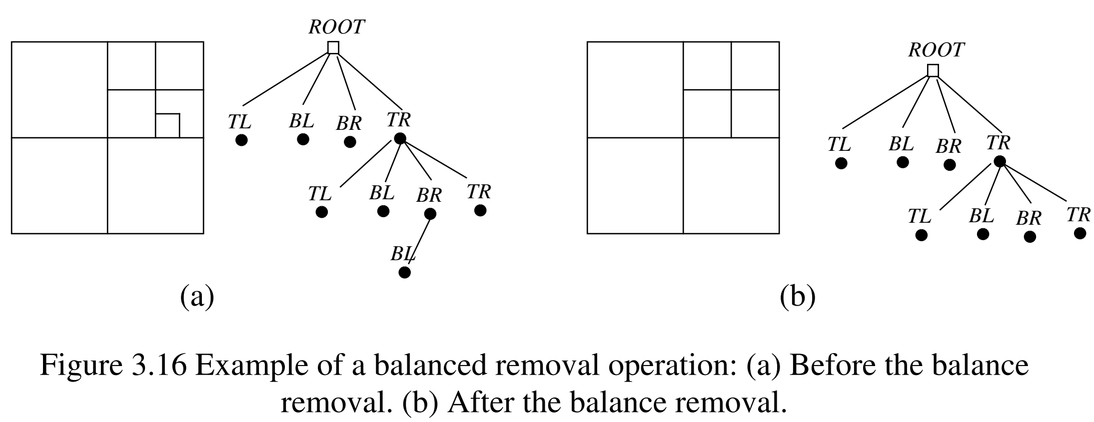


Figure 3.16 Example of a balanced removal operation: (a) Before the balance removal. (b) After the balance removal.

Lemma 3.2 Let *QT* and *RT* be two quadtrees where *QT* is a balanced quadtree and $QT \subseteq RT$. Let *R* be a leaf quadrant in the quadtree *RT* but not in *QT*. If *QT* is the resulting quadtree by removing *R* from *RT*, then $QT \subseteq balance\,(RT') \subset RT$.

Proof. By the assumptions of the lemma, we have (i) the only difference between *RT* and RT' is the quadrant *R*, (ii) *R* is not in *QT* and (iii) *QT* is a subtree of *RT*. These three assumptions imply that we have $QT \subseteq RT'$. Let

$$balance\,(RT') = RT_k \subseteq RT_{k-1} \subseteq \ldots \subseteq RT_2 \subseteq RT_1 \subseteq RT'$$

where RT_i is the result of applying the balance removal operation on RT_{i+1}. From *Lemma 3.1*, we know that $QT \subseteq RT_i$ for all i. Thus $QT \subseteq RT \subseteq balance\,(RT')$. Since $RT' \subset RT$, $balance\,(RT') \subset RT$. This completes the proof. ❑

Theorem 3.1 (***Existence of optimal balanced quadtrees***) According to the partial ordering relation defined in *Definition 3.4*, there is one maximal support and one minimal cover for any level control function *L*.

Proof. We first show that there is one maximal support for the level control function *L*.

Since *L* is a bounded function, there is an integer *maxL* which is an upper bound for *L*. Let *MaxQT* be the full quadtree of level *maxL*. *MaxQT* is balanced and any support quadtree of *L* is a subtree of *MaxQT*.

Let Σ be the set of all support quadtrees, i.e. $\Sigma = \{ST \mid ST \text{ is a support for } L\}$. Σ is not empty because it always contains the quadtree with only one quadrant, i.e. the root quadrant. Let Ω be the set of all balanced quadtrees. Then $\Sigma \subseteq \Omega$ and $MaxQT \in \Omega$. Since *MaxQT* is an upper bound of Σ, by the partial ordering theorem in the set theory [KNUT70], Σ has a minimal upper bound *UpperT* in Ω. We show that *UpperT* is also a support, i.e. $UpperT \in \Sigma$, by proving

1. *UpperT* is balanced;
2. $Level_{UpperT}(p) \le L(p)$ is true for all vertices in *UpperT*.

UpperT is balanced because it is in Ω. We show, by contradiction, that for any vertex *p* in the quadtree *UpperT*, $Level_{UpperT}(p) \le L(p)$.

Suppose there is a vertex p in $UpperT$ such that $L(p) < Level_{UpperT}(p)$. Then there is a leaf quadrant Q containing p and $Level(Q) = Level_{UpperT}(p)$. We remove Q from $UpperT$ and obtain a new quadtree $UpperT'$. For any support ST in Σ, ST does not contain Q. So ST is a subtree of $UpperT'$. By *Lemma 3.2*, $ST \subseteq balance(UpperT') \subset UpperT$. Since ST is any support, $balance(UpperT')$ must also be an upper bound for Σ in Ω, which contradicts that $UpperT$ is the minimal upper bound for Σ in Ω. Therefore, for any vertex p in the quadtree $UpperT$, $Level_{UpperT}(p) \leq L(p)$. Since $UpperT$ is the minimal upper bound for Σ and it is also in Σ, $UpperT$ is the maximal support.

Using the similar argument, we can show that there is one minimal cover for the function L. ❑

Corollary 3.1 For a given level control function L, if the maximal support is equal to the minimal cover, the balanced quadtree is compatible with L.

From Theorem 3.1, we conclude that there are two balanced quadtrees which are optimal with respect to a given level control function. One is the maximal support and the other the minimal cover. While some meshing schemes (such as MESHBUILD [MULL92]) try to find the minimal cover, our proposed scheme is to find the maximal support instead. The advantages of the proposed scheme include:

1. The scheme produces an optimal balanced quadtree.
2. The scheme is computationally efficient by visiting each quadrant only once during the entire quadtree generation process.

Section 3.3.3 provides a description of the algorithm for generating an optimal balanced quadtree and Section 3.3.4 gives the proof of the algorithm's correctness.

3.3.3 An Algorithm for Generating an Optimal Balanced Quadtree

A quadtree is generated by a recursive refinement process, i.e. quadrants being divided into sub-quadrants. For a level control function L, a quadrant Q is refined according to the L values at the Q's four corner vertices. For each vertex p of the quadrant Q, if the L value at the vertex p exceeds the level of the quadrant Q, we create a subquadrant around the vertex p. The refinement step is localized and each quadrant is visited only once in the generation of the quadtree. The refinement operation is summarized by the following procedure.

```
PROCEDURE   SimpleRefine(Q: Quadrant)
    Let p_i, i=0, 1, 2, 3, be the 4 vertices of Q;
    FOR i:=0 TO 3 DO
        IF Level(p_i) > Level(Q) THEN
            (* create a sub-quadrant around the vertex p_i *)
            Q_i = SubQuadrant(Q, p_i);
            (* recursive refinement *)
            SimpleRefine(Q_i);
        END;
    END;
END         SimpleRefine
```

One problem in the simple refinement procedure is that it may produce an unbalanced tree, as illustrated by the example shown in Figure 3.17, where the level control function values are shown in parenthesis.

To ensure the one level difference rule, we modify the refinement algorithm with a new technique called *rapid descending*, which is explained as follows. When the quadrant Q is refined, the value of the level control function L is modified if either of the following two cases occurs:

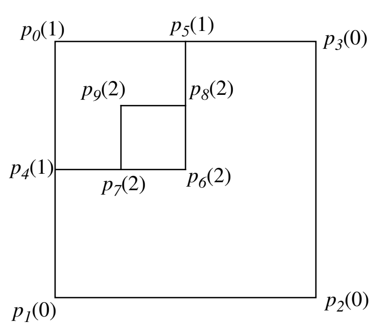
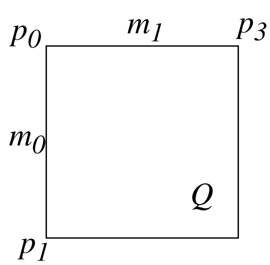


Figure 3.17 An unbalanced quadtree generated by the SimpleRefine.

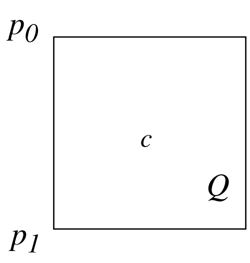
1. The level of the quadrant Q is equal to the L value on any one of the 4 vertices. Suppose $Level(Q) = L(p_0)$, as shown in Figure 3.18(a). If $L(Q)$ is also equal to one of $L(p_1)$ and $L(p_3)$, i.e. $Level(Q) = L(p_1)$ and/or $Level(Q) = L(p_3)$, where p_1 and p_3 are two neighbor vertices of p_0, then the L values on two mid-edge vertices m_0 and m_1 are reduced to $Level(Q) + 1$.

2. The level of the quadrant Q is equal to the L values on the two vertices which are on the same edge of Q. Suppose $Level(Q) = L(p_0) = L(p_1)$, as shown in Figure 3.18(b). In this case, the L value on the center vertex c is reduced to $Level(Q)+1$.



IF $Level(Q) = L(p_0)$ AND
$(Level(Q) = L(p_1)$ OR $Level(Q) = L(p_3))$ THEN
$L(m_0) = L(m_1) = Level(Q)+1$

(a)



IF $Level(Q) = L(p_0)$ AND $Level(Q) = L(p_1)$ THEN
$L(c) = Level(Q)+1$

(b)

Figure 3.18 Modification of the level control function.

The modified refinement algorithm is summarized by the following procedures.

```
PROCEDURE   ModifiedRefine(Q: Quadrant)

    (* modify the level control function values if necessary *)
    ModifyLCF(Q);
    (* refine the quadrant *)
    SubQuadrantGeneration(Q);

END         ModifiedRefine

PROCEDURE   ModifyLCF(Q: Quadrant)

    (* modify the level control function for mid-edge vertices if necessary *)
    FOR each vertex p of Q DO
        Let p_i and p_j be two neighbor vertices of p with respect to Q;
        m_i = MidVertex(edge(p, p_i));
        m_j = MidVertex(edge(p, p_j));
        c = CenterVertex(Q);
        IF L(p) = Level(Q) THEN
            IF L(p_i) = Level(Q) OR L(p_j) = Level(Q) THEN
                (* modified the L values for the mid-edge vertices *)
                L(m_i) = Level(Q)+1;
                L(m_j) = Level(Q)+1;
                (* modified the L value for the center vertex *)
                L(c) = Level(Q)+1;
            END;
        END;
    END;

END         ModifyLCF
```

With the modified refinement procedure, we can show, in the next subsection, that the quadtree generated by the modified refinement algorithm is indeed balanced.

```
PROCEDURE   SubQuadrantGeneration(Q: Quadrant)

    Let {p_i: i=0, 1, 2, 3} be corner vertices of Q;
    Let c be the center vertex of Q;
    FOR i:=0 TO 3 DO
        IF L(p_i) > Level(Q) THEN
            (* create a sub-quadrant around vertex p_i *)
            Q_i = SubQuadrant(p_i, Q);
            ModifiedRefine(Q_i);
        ELSE IF Irregular(Q) AND L(c) > Level(Q) + 1 THEN
            (* create a sub-quadrant around vertex p_i *)
            Q_i = SubQuadrant(p_i, Q);
            ModifiedRefine(Q_i);
            Delete Q_i but not its subquadrants;
        ENDIF;
    END;

END            SubQuadrantGeneration
```

3.3.4 Correctness of the Modified Refinement Algorithm

To verify the correctness of the modified refinement algorithm discussed in the previous subsection, we need to prove that the quadtree generated by the modified refinement algorithm is the maximal support for a given level control function. We shall prove this assertion in three parts. First, we show that the generated quadtree is balanced (*Lemma 3.3*). We then show that the generated quadtree is a support of the level control function (*Lemma 3.4*). Finally we show that the generated quadtree is an upper bound for all supports of the level control function (*Lemma 3.5*).

Lemma 3.3 For a given level control function L, the quadtree QT generated by the modified refinement algorithm is balanced.

Proof. There are three types of leaf quadrants in the quadtree QT. They are the non-refined, corner-refined and side-refined quadrants as shown in Figure 3.19. The shaded area corresponds to the leaf region of the quadrants. In Figure 3.19(a), the leaf region of the non-refined quadrant is bounded by the edges (p_0, p_1), (p_1, p_2), (p_2, p_3) and (p_3, p_0). In Figure 3.19(b), the leaf region of the corner-refined quadrant is bounded by the edges (m_0, p_1),

(p_1, p_2), (p_2, p_3), (p_3, m_1), (m_1, c) and (c, m_0). In Figure 3.19(c), the leaf region of the side-refined quadrant is bounded by the edges (p_0, m_0), (m_0, c), (c, m_1), (m_1, p_3) and (p_3, p_0). To prove the quadtree *QT* is balanced, it is sufficient to show that for each of three leaf quadrants, there is no mid-edge vertex on the boundary of the leaf region.

Case 1. *Q* is a non-refined leaf quadrant, as shown in Figure 3.19(a). Because $Level(Q) = Level(p_0) = Level(p_1)$, there is no further subdivision necessary around both vertices p_0 and p_1. Thus there is no mid-edge vertex on the edge (p_0, p_1). Similarly, we can show that there are no mid-edge vertices on the edges (p_1, p_2), (p_2, p_3) and (p_0, p_3) because $Level(Q) = Level(p_0) = Level(p_1) = Level(p_2) = Level(p_3)$.

Case 2. *Q* is a corner-refined leaf quadrant, as shown in Figure 3.19(b). We have $Level(Q) = Level(p_1) = Level(p_2) = Level(p_3)$ and $Level(m_0) = Level(m_1) = Level(c) = Level(Q)+1$. Using the similar argument as discussed in case 1, we can show that there are no mid-edge vertices on the edges (m_0, p_1), (p_1, p_2), (p_2, p_3), (p_3, m_1), (m_1, c) and (c, m_0).

Case 3. *Q* is a side-refined leaf quadrant, as shown in Figure 3.19(c). We have $Level(Q) = Level(p_0) = Level(p_1)$ and $Level(m_0) = Level(m_1) = Level(c) = Level(Q)+1$. Similar arguments as in cases 1 and 2 hold and there are no mid-edge vertices.

Since there is no mid-edge vertice on the boundary of any of the three leaf regions, the quadtree *QT* is balanced. ❑

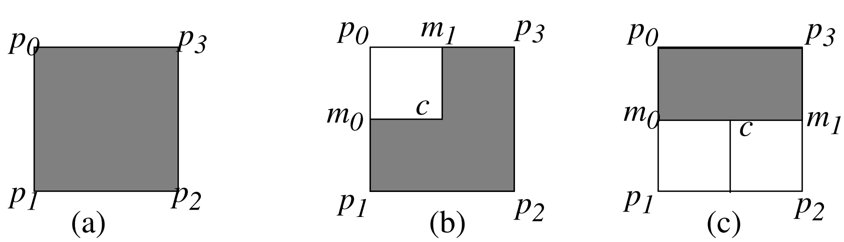


Figure 3.19 Three types of leaf quadrants: (a) Non-refined. (b) Corner-refined. (c) Side-refined.

Lemma 3.4 The quadtree *QT* generated by the modified refinement algorithm is a support for the level control function *L*.

Proof. For a quadtree *QT* to be a support, two conditions must be satisfied.

1. *QT* is balanced.
2. For any vertex *p* in the quadtree *QT*, $Level_{QT}(p) \le L(p)$.

The first condition is the result of *Lemma 3.3*. The proof of the second condition is given as follows.

If the level control function *L* is not modified in the process of the refinement, then the proof can be derived from the fact that for any quadrant *Q* in the quadtree *QT*,

$$Level(Q) \le min\{L(p_0), L(p_1), L(p_2), L(p_3)\} \tag{3.2}$$

where $\{p_i\}$ are the four corner vertices of *Q*. With the proposed modified algorithm, however, the level control function may be modified during the process of the refinement. The modification of the level control function only reduces the function's values. Thus, the inequality (3.2) still holds.

Since *QT* satisfies both conditions, *QT* is a support. ❑

Lemma 3.5 Let *QT* be the quadtree generated by the modified refinement algorithm. Then any support *ST* of the level control function *L* is a sub-quadtree of *QT*, i.e. $ST \subseteq QT$. In other words, *QT* is an upper bound for all supports.

Proof. The proof consists of 2 steps:

1. We shall first show that if *ST* is not a subtree of *QT*, then there is a non-empty vertex set Π such that any vertex *p* in Π satisfies $Level_{ST}(p) > Level_{QT}(p)$.
2. We shall then show that the existance of the non-empty set Π contradicts the fact that *ST* is a balanced quadtree.

Suppose *ST* is not a subtree of *QT*. Then there is a leaf quadrant *Q* in *ST* but not in *QT*. Tracing back the parent link of *Q*, we can always find a quadrant Q' such that the quadrant Q' is in *ST* but not in *QT* and the quadrant $parent(Q')$ is in both *ST* and *QT*. For the convenience of the discussion, we assume *Q* is Q', as shown in Figure 3.20.

Let p be the vertex common to both Q and $parent(Q)$. Then p is a vertex in both ST and QT. Since $parent(Q)$ is a leaf quadrant of QT and QT is constructed by the modified refinement algorithm,we have $Level_{QT}(p) = Level(parent(Q)) < Level(Q) = Level_{ST}(p)$. This shows that Π is not empty.

Let us prove that the non-empty vertex set Π induces a contradiction. Let p be a vertex in Π such that the level of p is the least among all other vertices in Π, i.e. for any $p' \in \Pi$, $Level_{QT}(p) \le Level_{QT}(p')$. Since $p \in \Pi$,

$$Level_{QT}(p) < Level_{ST}(p) \tag{3.3}$$

Since ST is a support quadtree of L,

$$Level_{ST}(p) \le L(p) \tag{3.4}$$

Combining (3.3) and (3.4), we get $Level_{QT}(p) < L(p)$. This implies that the level control function is modified at the vertex p during the generation of QT. There are two cases in which values of the level control function are modified, which are shown in Figure 3.21. We claim that either case induces a contradiction.

Case 1. p is the mid-edge vertex of the quadrant Q in the quadtree QT, as shown in Figure 3.21(a). Since the level control function at p is modified, we have $Level_{QT}(p_0) = Level(Q)$.

Since p_0 is not in Π, $Level_{ST}(p_0) \le Level_{QT}(p_0) = Level(Q)$. This implies that

$$Level_{ST}(p_0) = L(Q) \tag{3.5}$$

Based on the modified refinement algorithm, we have

$$Level_{QT}(p) = Level(Q) + 1 \tag{3.6}$$

Since $p \in \Pi$, we have

$$Level_{ST}(p) > Level_{QT}(p) \tag{3.7}$$

Combining (3.5), (3.6) and (3.7), we have $Level_{QT}(p_0) - 1 < Level_{ST}(p)$, which contradicts that ST is a balanced quadtree.

Case 2. p is the center vertex of a quadrant Q, as shown in Figure 3.21(b). In this case, we still have $Level_{QT}(p_0) = Level(Q)$. This implies (3.5) is still true. Because both (3.7) and (3.8) still hold. We get the same contradiction as in case 1.

This completes the proof of the second condition. Thus the quadtree *QT* is an upper bound for any support of the level control function. ❑

Theorem 3.2 (***Correctness of the algorithm***): The quadtree *Q* generated by the modified refinement algorithm is the maximal support for the level control function.

Proof. The proof follows directly from Lemma 3.3, Lemma 3.4 and Lemma 3.5.❑

(a) quadtree *ST* (b) quadtree *QT*

Figure 3.20 Quadrant *Q* is in *ST* but not in *QT* and *parent(Q)* is in both quadtrees: (a) Quadtree *ST*. (b) Quadtree *QT*.

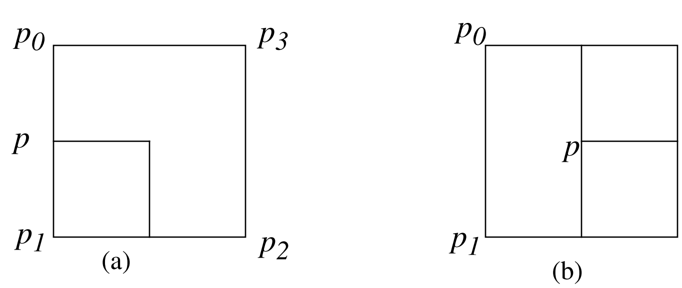


Figure 3.21 Two cases where the level control function values at p are modified: (a) $Level(p_0) = Level(p_3) = Level(Q)$. (b) $Level(p_0) = Level(p_1) = Level(Q)$

From the above theorem, we conclude that the quadtree generated by the modified refinement algorithm, as discussed in 3.3.3, is an optimal balanced quadtree with respect to a given level control function.

3.4 Interior Quadrant Triangulation

In this section, we discuss the triangulation of a region represented by a balanced quadtree. There are two types of leaf quadrants, regular and irregular. A regular leaf quadrant has no sub-quadrants, i.e. of degree 0. An irregular leaf quadrant has 1, 2 or 3 sub-quadrants. For a regular leaf quadrant, since the quadtree is balanced, at most there is one mid-edge vertex for each side. There are five patterns that need to be considered; (a) no mid-edge vertex, (b) one mid-edge vertex, (c) 2 mid-edge vertices (two opposite edges), (d) 2 mid-edge vertices (two neighboring edges) and (e) 3 mid-edge vertices. As shown in Figure 3.22, for all five patterns, the triangles generated do not have obtuse angles.

For irregular leaf quadrants, we only need to triangulate the leaf region. Two types of irregular quadrants require different triangulation schemes other than the one for the regular quadrants: a corner-refined quadrant (see Figure 3.23), and a side refined-quadrant (see Figure 3.24). Since the quadtree is balanced, there are three triangulation patterns for a corner-refined irregular quadrant as shown in Figure 3.23. For a side-refined irregular

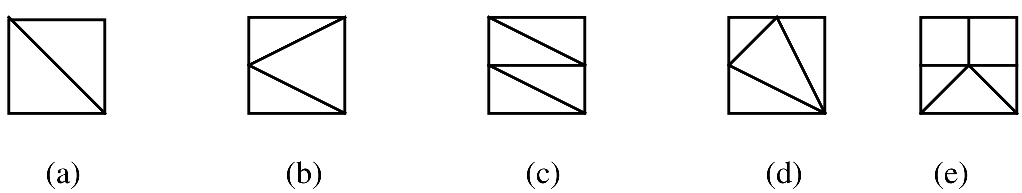
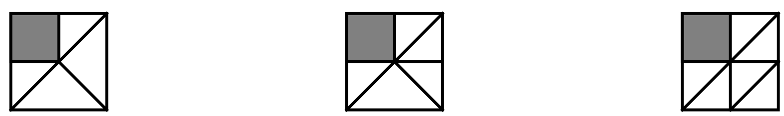


Figure 3.22 Triangulation of leaf triangles.

quadrants, there are four different triangulation patterns, as shown in Figure 3.24. In either case, the triangulation does not lead to triangles with obtuse angles.

There are two other types of irregular leaf quadrants, one with two diagonal subquadrants and the other with three subquadrants, as shown in Figure 3.25. Both can be triangulated as a regular quadrant discussed earlier.

Figure 3.26 shows a mesh example of an n^+-p-n bipolar transistor. The level control function is based on the gradient of the doping profile. The mesh generated reflects the change of the doping profile. The top right corner is heavily doped by electrons; therefore, the region has a much denser mesh ring around it. The example also shows that the mesh element sizes can vary significantly in a short range without introducing any obtuse



(a) (b) (c)

Figure 3.23 Triangulation for a corner-refined quadrant: (a) No mid-edge vertex. (b) One mid-edge vertex. (c) Two mid-edge vertices.

(a) (b) (c) (d)

Figure 3.24Triangulation for a side-refined quadrant: (a) No mid-edge vertex. (b) One mid-edge vertex on the outside. (c) One mid-edge vertex on the inside. (d) Two mid-edge vertices.

(a) (b)

Figure 3.25 Irregular quadrants triangulated as leaf quadrants.

triangles. This feature is often desired in the semiconductor device simulation where large size triangles should be used in the device substrate where not much activities occur.

3.5 δ-Zone Triangulation

3.5.1 Small Feature Elements

One problem with quadtree based mesh generation schemes is that many small elements may be generated along the boundary of the region. This is because the boundary information is not taken into consideration when a quadrant is subdivided. If the boundary objects do not conform well with the quadrants, small elements are created, as illustrated by the example in Figure 3.27(a). Many of these small elements may have poor aspect ratio. The small elements are introduced not because of the mesh density requirement or the need of the boundary approximation. They are the result of artificial cut lines of the quadtree. Such small feature elements not only consume the unnecessary computing resource for the simulator, but also have a negative effect on the simulation results. With

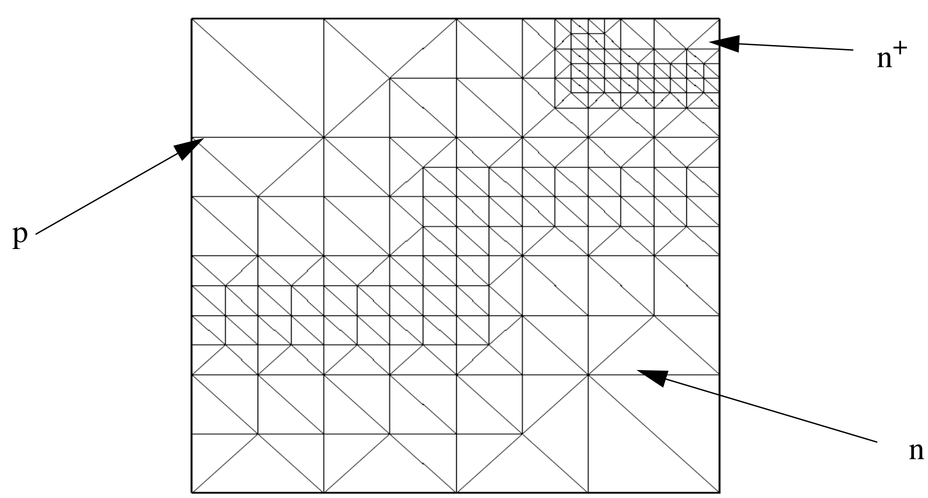
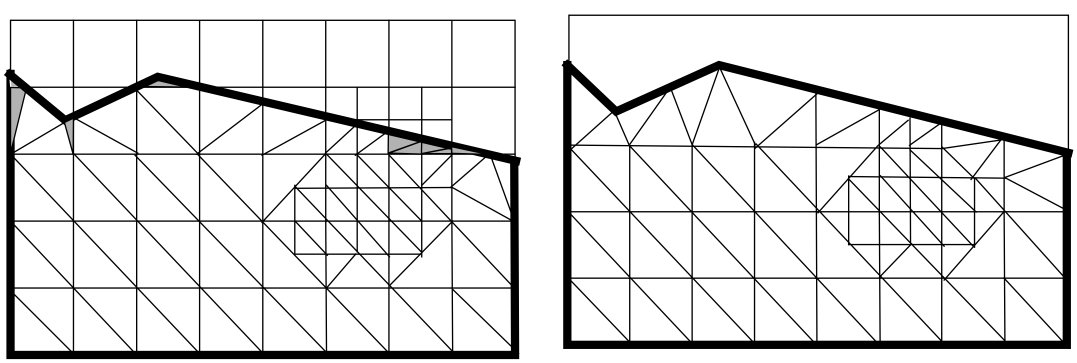


Figure 3.26 A quadtree mesh base on the modified level control function for a bipolar device.

some simple modifications at the boundary quadrants, we can greatly improve the mesh quality along the boundary (see Figure 3.27 (b)).

Different approaches have been proposed to resolve the generation of small feature elements. One approach is to add a post-processing step after the decomposition tree is created. The idea is that, after the creation of the decomposition tree, we modify the boundary quadrants (without moving boundary objects, i.e. boundary edges, vertices, etc.) to make them conformal to the boundary of the domain. A common technique taken is to collapse the quadrant objects (vertices, edges). This is illustrated by the example shown in Figure 3.28. In this example, vertices A and A' are very close to each other. So we collapse A' with A. Similarly, we collapse B' with B, C' with C, D' with D and E' with E. After collapsing the vertices around the boundary region, all small feature elements are eliminated, as shown in Figure 3.28 (b). This method has been explored in [ALRU90]. The



(a) (b)

Figure 3.27 Small feature elements for a quadtree based mesh.

potential problem of this approach is that a more elaborate data structure is required and the triangulation algorithm becomes rather complex.

Another approach is to create initial rectangles such that they capture the edge vertices [MULL92]. Rectangles are further refined into sub-rectangles. Such a scheme works well as long as the boundary is fairly smooth. But for a rough boundary with many zigzag lines, the approach could result in many unnecessary small mesh elements along the boundary, as illustrated by the example shown in Figure 3.29.

In the next section, we introduce a new technique called the δ-zone triangulation, which consists of two steps: (1) δ-zone detection and (2) δ-zone triangulation.

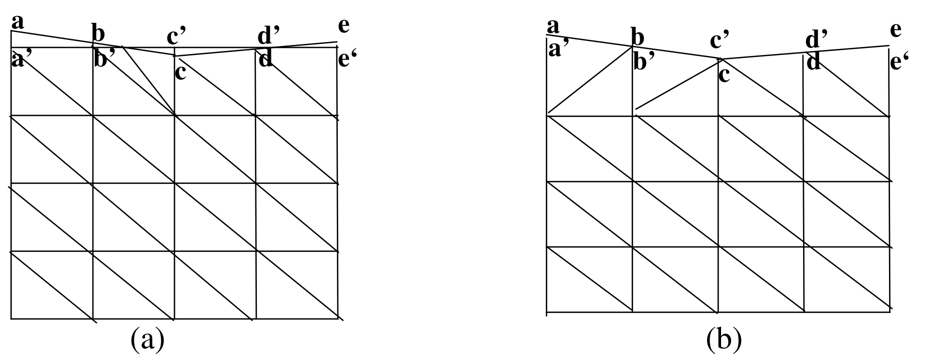


Figure 3.28 Collapsing of edges and vertices near the boundary.

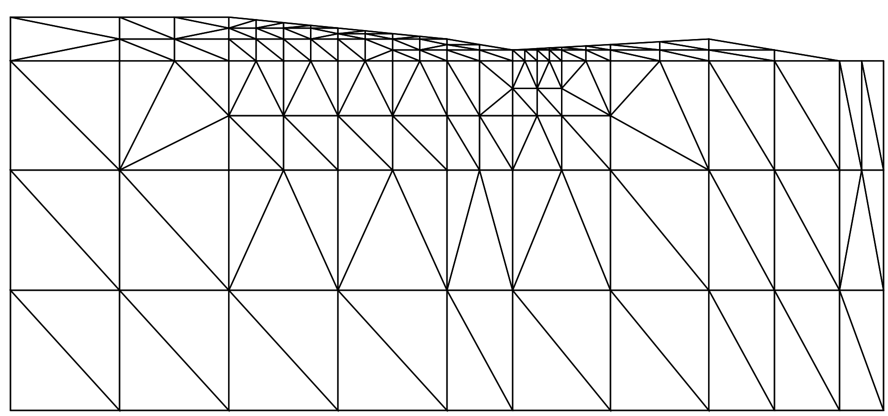


Figure 3.29 A different approach for handling boundary.

3.5.2 δ-Zone and δ-Zone Detection

In this subsection, a new scheme is introduced to handle small feature triangles near the boundary. The basic idea is to separate the boundary of the region from the interior by a narrow band, called the δ-zone. Inside the δ-zone, vertices and edges of the quadrants are eliminated and a triangulation scheme which is more suitable for general polygon triangulation is applied.

Definition 3.6 Given a quadrant R, the δ-*cover* of R, $\delta(R)$, is a square which is created by extending R on each side with a distance δ. For a given point p in the geometry region, a quadrant Q is called a δ-*cover* of p, if p is contained in $\delta(Q)$ but not any descendent quadrant of Q. For a given geometry region D and a quadtree QT, the δ-*zone* of the geometry region with respect to the quadtree QT, denoted as $\delta(D, QT)$, is defined to be the collection of δ-covers of all boundary points, i.e., $\delta(D, Q\mathrm{T}) = \{\delta(p) \mid p \in \Gamma(D), Q \in QT$, where $\Gamma(D)$ is the boundary of $D\}$. We call δ the width of the δ-zone boundary.

These concepts are illustrated by Figure 3.30, Figure 3.31 and Figure 3.32. In Figure 3.31, the shaded area corresponding to the δ-zone of the region *abcdef*, where δ is chosen to be 1/4-th length of the smallest quadrant's length.

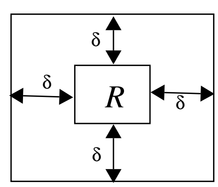


Figure 3.30 The δ-cover for a quadrant *R*.

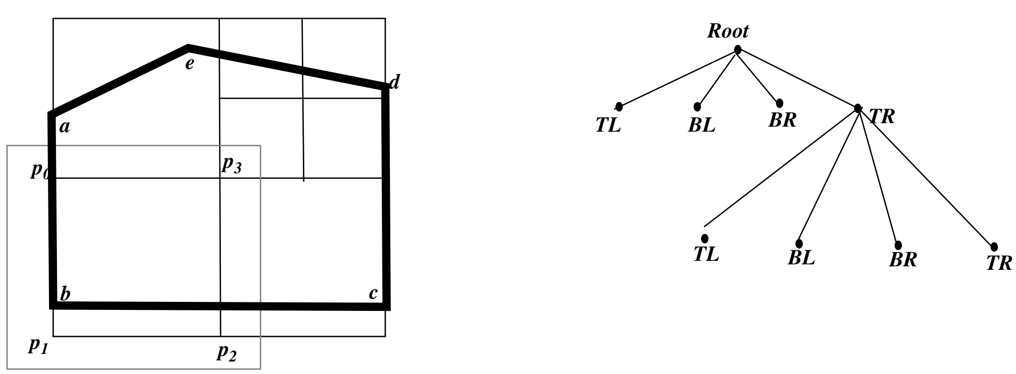


Figure 3.31 An example of the δ-cover for a point: δ(b) = */BL*.

For a given geometry region *D* and a quadtree *QT*, the corresponding δ-zone varies according to δ. There are two extreme cases. One is when δ is too large such that the δ-zone boundary covers the whole region *D*. The opposite extreme is when δ is very small and the δ-zone is degenerated to the set of all B-node quadrants[1].

Generally speaking, as δ becomes larger, the quadtree structure becomes less effective. In other words, the quadtree becomes more degenerated. To achieve a quality mesh, a proper δ should be selected. Clearly, the best choice of δ is problem dependent. Generally speaking, a smoother boundary requires a smaller δ, which implies the quadtree based mesh scheme is more effective. In our implementation of the mesh generator, the default value δ is taken between 1/8-th and 1/4-th of the length of the smallest quadrant's length.

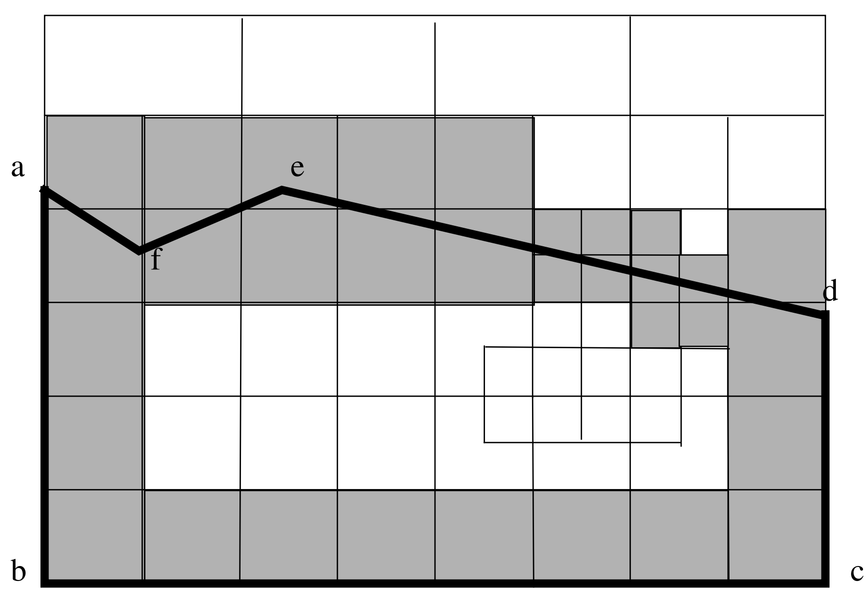


Figure 3.32 An example of the δ-zone.

1. The concept of the B-node is defined in Section 3.2.3.

To conclude this subsection, we describe a procedure for finding the δ-zone.

PROCEDURE δZoneDetection (*D*: GeometryRegion, *QT*: Quadree, *Q*: Quadrant, δ: width)

IF δ(*Q*) is not a δ-cover for any boundary point THEN
 RETURN;
IF *Q* is a leaf node and its leaf region contains some boundary point of *D* THEN
 Q is a δ-zone quadrant;
FOR each subquadrant *SubQ* of *Q* DO
 δ*ZoneDetection* (*D*, *QT*, *SubQ*, δ);
END;

END δZoneDetection

3.5.3 δ-Zone Triangulation

Once the δ-zone is created, many triangulation schemes can be applied. In this section, a constrained Delaunay triangulation (CDT) scheme is proposed, which is the modification of the algorithm by Lee and Lin [LEE86].

To apply a CDT, we first need to collect all vertices and boundary edges. The whole δ-zone is sandwiched by two non-intersecting boundaries, external and internal. For the example shown in Figure 3.33, the external boundary is *abcdef* and the internal one is $p_0p_1p_2p_3p_4$. To ensure the conformity of the final mesh, all quadtree vertices on the internal boundary should be included in the vertex set for CDT. For the external boundary, all edge vertices for the original boundary should be included in the vertex set. In the example shown, vertices *a*, *b*, *c*, *d, e* and *f* should be included in the vertex set. In addition, we add points which are intersections of quadrant edges and the external boundary. This is illustrated by the example shown in Figure 3.33(b), where p_1 and p_2 are two intersection points of the external boundary edge (*e, f)* with quadrants. However, we must exclude those intersection points which are too close to each other; otherwise, small feature elements could be produced. In the example, only one of q_1 and q_2 should be included in the vertex set. The following procedure describes the process of creating the vertex set *V* for CDT.

PROCEDURE CreateVertexSet : VertexSet

Initialize the vertex set V to be empty;
FOR each quadtree vertex p on the internal boundary of δ-zone DO
add vertex p to V;
END;
FOR each vertex p on the external boundary DO
add vertex p to V;
END;
FOR each intersection point p of quadrant edges and the external boundary DO
IF in the δ neighborhood of p there is no other vertex in V THEN
add vertex p to V;
END;
RETURN V;

END CreateVertexSet

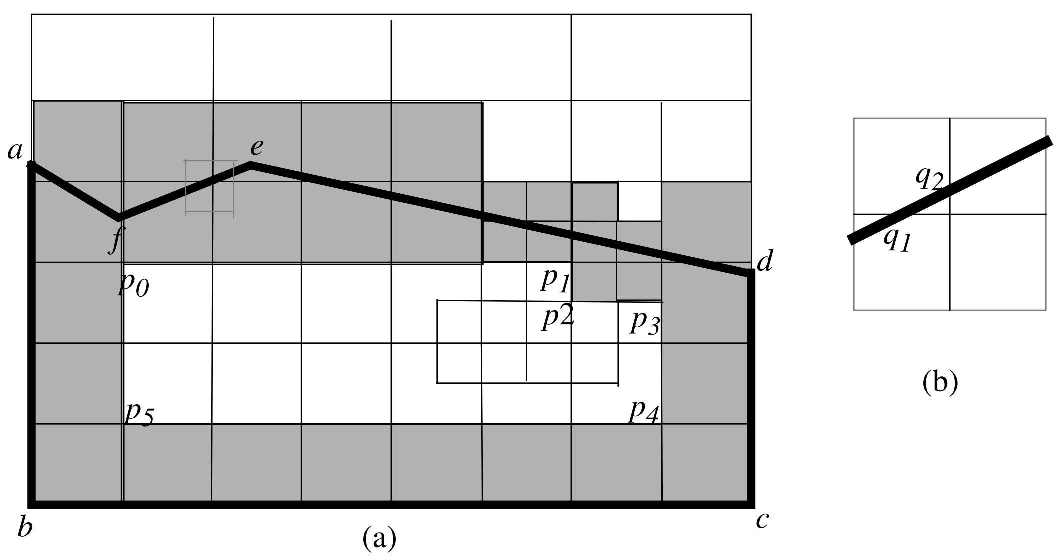


Figure 3.33 Internal and external boundaries for a δ-zone: (a) δ-zone.
(b) Dotted area with enlargement

With the vertex set *V*, we can create the edge set *E* for CDT. For each pair of vertices in *V*, the edge connecting these two vertices is included in the edge set *E* if the edge is a part of the external or internal boundary. The edge should not be included in the edge set *E* if it is covered by another edge in *E*. The process is described by the following procedure.

PROCEDURE CreateEdgeSet: EdgeSet

Initialize the edge set *E* to be empty;
FOR each pair of vertices p_1, p_2 in *V* DO
IF $edge(p_1, p_2)$ is a part of the external or internal boundary THEN
IF no other vertices in *V* is on $edge(p_1, p_2)$ THEN
Add $edge(p_1, p_2)$ to *E*;
END;
RETURN *E*;

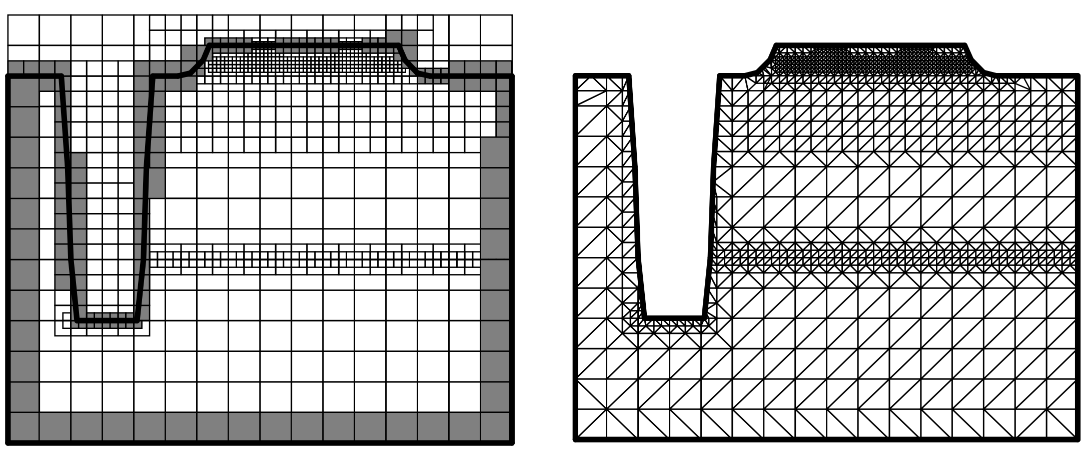
END CreateEdgeSet

Given a vertex set *V* and an edge set *E*, the boundary region can be triangulated with any constrained Delaunay triangulation scheme. Two schemes are mentioned here. One is proposed by Lee and Lin [LEE86], which first computes the visibility set [ASON86] and then finds all Delaunay incident edges. The scheme is simple to implement and the complexity is $O(n^2)$. The other scheme is based on the divide-and-conquer idea [SHEW89], which is of complexity O(n log n). We follow the scheme by Lee and Lin because of its simplicity and because the boundary region for a polygon is relatively small.

An example of a bipolar device with a deep trench is shown in Figure 3.34. Figure 3.34(a) is the quadtree decomposition with δ-zone (shaded area) and Figure 3.34(b) is the final mesh after triangulation.

3.6 Summary

In this chapter, the concept of the quadtree data structure and some related properties have been discussed. Two new techniques are introduced for quadtree based mesh scheme: the level control function and the δ-zone triangulation. We have shown that the balanced quadtree generation method based on the modified level control function is one of the optimal approximations which satisfy both the mesh density requirement and the one level difference rule. The second part of the chapter discussed the δ-zone concept and its usefulness for eliminating small feature elements. A constrained Delaunay triangulation scheme is also presented for triangulating the δ-zone.



(a) (b)

Figure 3.34 A bipolar device with a deep trench: (a) Quadtree decomposition with the δ-zone. (b) Final quadtree mesh.