Chapter 4

3D Mesh Generation

4.1 Introduction

One advantage for the tree based meshing algorithms is that their 2D implementations can be extended to 3D. In this chapter, we present the 3D extension of the algorithms discussed in Chapter 3. We first review briefly the mesh process for generating 2D mesh. As summarized in Figure 4.1, there are four steps for generating a mesh; (1) generating a balanced quadtree, (2) creating a δ-zone, (3) triangulating the interior quadrants and (4) triangulating the δ-zone. The 3D extension of the first 2 steps is much simpler than the last two steps.

It has been well known that in computational geometry many 2D computational geometry properties do not hold for 3D problem. For example, any 2D polygonal geometry region can be triangulated, i.e. partitioned into non-overlapping triangles [PREP85], but this property cannot be ensured for 3D polyhedrons. Another example is that the Delaunay mesh in 2D satisfies the min-max angle property, i.e. the maximum angle for all elements is minimal among all possible meshes for a given set of vertices. However, a 3D Delaunay tetrahedron mesh does not necessarily have this property.

The major part of this chapter is devoted to the discussion of Steps (3) and (4), i.e. the tetrahedralization of the interior octants and the δ-zone. The chapter is organized as follows: In 4.2, we first present a brief review of the definitions and terminologies discussed in Chapter 3. We then describe a 3D extension of the algorithm discussed in Chapter 3 for generating a balanced octree. In 4.3, a detailed discussion on how to tetrahedralize interior octants is provided. We also present a proof that the tetrahedron mesh generated by the proposed algorithms satisfies the Delaunay criterion. In 4.4, we discuss the algorithm for tetrahedralizing the δ-zone. In the last section, a brief summary is presented.

4.2 3D Spatial Decomposition and Octree

4.2.1 Review of Definitions and Terminologies[1]

An *octree* is a data structure to represent a spacial decomposition of a 3D region. The top octant, also called the *root*, encloses the region. Each octant is called the *node* of the

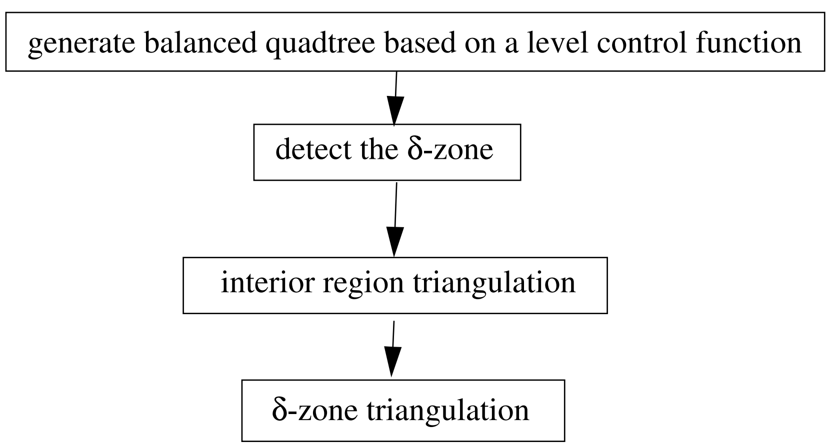


Figure 4.1 Review of mesh process for generating 2D mesh.

1. Most definitions and terminologies have been defined in Chapter 3. This section concerns with the terminologies that are specific to the octree representation.

octree and can be divided into eight *sub-octants*, which are labelled as the *Left-Front-Bottom(LFB), Right-Front-Bottom(RFB), Left-Back-Bottom(LBB), Right-Back-Bottom(RBB), Left-Front-Top(LFT), Right-Front-Top(RFT), Left-Back-Top(LBT), Right-Back-TOP(RBT)* octants respectively, as shown in Figure 4.2. With the hierarchical tree structure, any octant can be addressed by using a relative path mechanism. For the example shown in Figure 4.3, the full path name for the octant $\{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ is */RFT/LFT*.

An octant node is a *leaf* if it does not have any sub-octants; otherwise, it is an *intermediate* node. An octant is an *I-node*, an *O-node* or a *B-node* if it is inside, outside or on the boundary of the region. An octant node is *regular* if it has 0 or 8 sub-octants; otherwise, it is *irregular*. Examples in Figure 4.4 illustrate irregular octants with different number of sub-octants. The *level* of an octant is defined to be the length of the path between the octant and the root of the octree. In the example shown in Figure 4.3, the

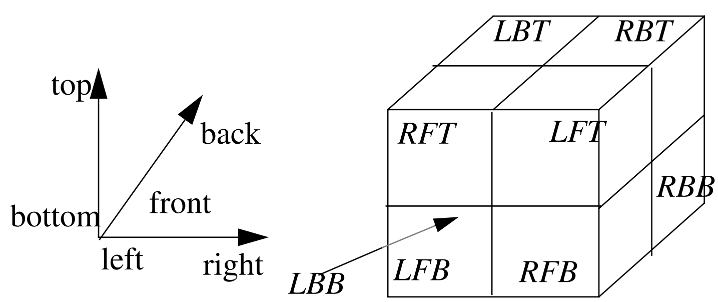


Figure 4.2 Local addresses for sub-octants.

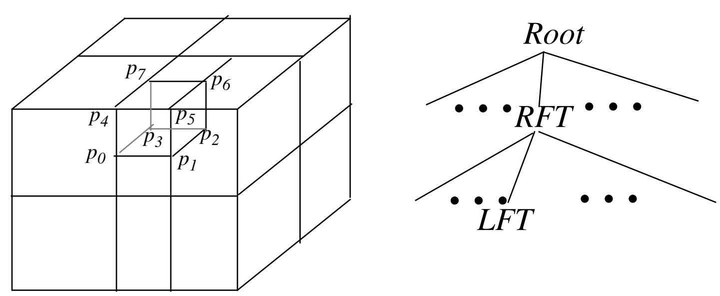


Figure 4.3 An illustration of the relative path name of an octant.

octant */RFT/LFT* is of level 2. An octree is *balanced* if it satisfies the one-level difference rule.

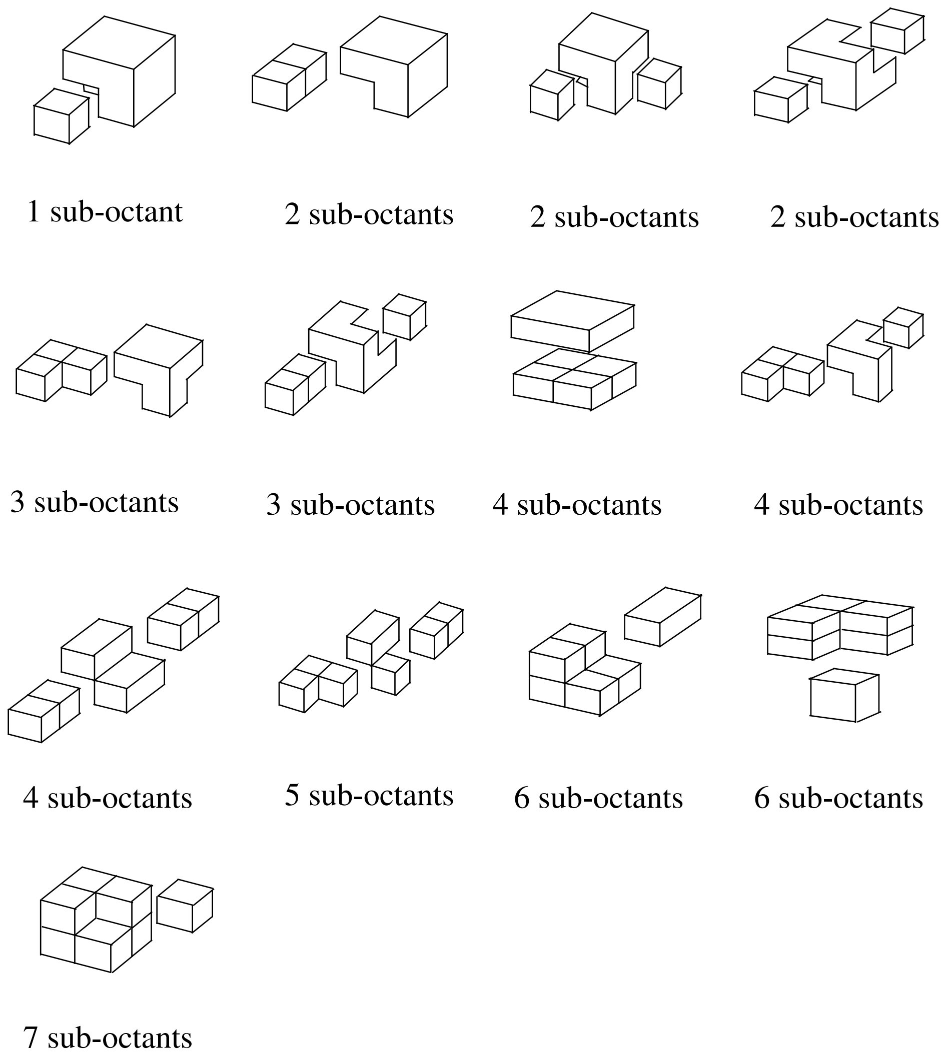


Figure 4.4 Illustration of irregular octants with different number of sub-octants.

Definition 4.1 (*one level difference rule*) An octree satisfies the one-level difference rule if for each leaf octant, the difference of the octant's level and the level of any neighboring leaf octant (left, right, front, back, bottom, top) is at most 1.

A non-balanced octree may generate a mesh with ill-shaped elements, as discussed in Chapter 3 for the quadtree mesh. To ensure a balanced octree, the level control function concept is extended to 3D. A *level control function* is a non-negative integer function which is defined on a meshing region. An octree is said to be *compatible* with a level control function L if each leaf octant has the level equal to the level control function's value on each of the octant's vertices. In the next subsection, a general algorithm to generate a balanced octree is presented which is the extension of the algorithm discussed in Section 3.3.5. The algorithm guarantees that the octree is balanced and compatible with the level control function. Furthermore, it can be shown that the resulting octree is optimal with respect to the level control function.

4.2.2 Balanced Octree Generation

In this section, we discuss an octree generation algorithm for a given level control function, which is similar to the one discussed in 3.3.5 for 2D. We start the octree construction with the creation of a cube which bounds the entire meshing region. Let us assume that the level control function L is defined on the cube. The cube is the root node for the octree and is used as the base octant for the recursive refinement process. For each vertex of the octant, if the level of the octant is less than the L value on the vertex, a sub-octant is created around the vertex, as illustrated in Figure 4.5 where the L values are in parenthesis. The refinement is recursively applied to all sub-octants until the octree generated is compatible with the level control function L.

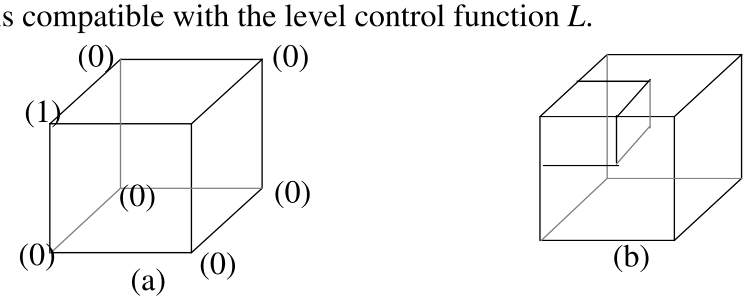


Figure 4.5 Illustration of the refinement process: (a) Before refinement. (b) After refinement.

To guarantee the one-level difference, some modifications of the level control function L have to be made during the refinement process. The algorithm to modify the level control function for 3D problems is more complex than the one discussed in 3.3.5. To facilitate the comprehension of the modified algorithm, we introduce the following definition.

Definition 4.2 Suppose p is a vertex of an octant O and p_1, p_2, p_3 are three neighboring vertices of p. The vertex p is said to be "*refinement equivalent*" with respect to the octant O if $L(p) = Level(O)$ and $L(p_1) = L(p_2) = L(p_3) = L(p)+1$. An octant O is said to be *fully refinable* if one of the following two conditions is satisfied:

1. One of the octant's vertices is refinement equivalent;
2. The octant has eight sub-octants.

The definition is illustrated in Figure 4.6, where the L values are shown in parenthesis. In the example, the vertex p has the L value of 0 while all three neighboring vertices p_1, p_2 and p_3 have the L values of 1. Since the level for the octant O is 0, the vertex p is refinement equivalent. Hence, the octant O is fully refinable.

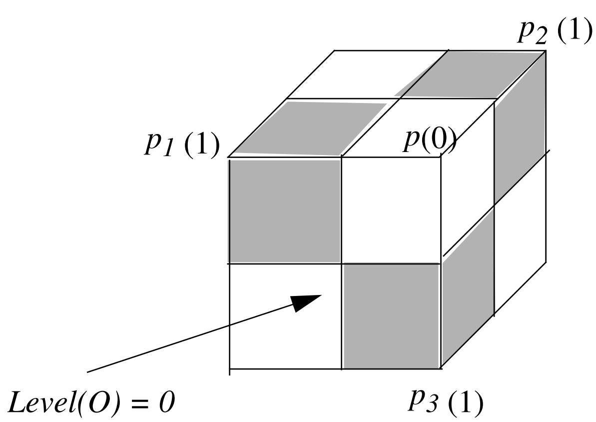


Figure 4.6 Illustration of a *"refinement equivalent"* vertex and a *"fully refinable"* octant.

A fully refinable octant is always divided into eight sub-regions. For the example shown in Figure 4.7, there are two fully refinable octants: O_1 has only four sub-octants and O_2 has eight sub-octants.

Based on the previous definition, we can now describe the algorithm. In the process of the octree generation, there are three cases in which the level control function values need to be modified, namely, modification of a mid-octant vertex, modification of a mid-face vertex and modification of a mid-edge vertex.

1. (*modification of a mid-octant vertex*): Let m_o be the mid-octant vertex of the octant O, as shown in Figure 4.8(a). If the octant O is not fully refinable, then the L value at the mid-octant vertex m_0 is reduced to the level of the octant plus 1, i.e. $L(m_o) = Level(O)+1$; otherwise, the L value at m_o remains the same.

2. (*modification of a mid-face vertex*): Let m_f be the mid-face vertex for both octants O_1 and O_2, as shown in Figure 4.8(b). If both of the two octants are not fully-refinable, then the L value at the vertex m_f is reduced to the level of the octant plus 1, i.e. $L(m_f) = Level(O_1)+1$; otherwise the L value at m_f remains the

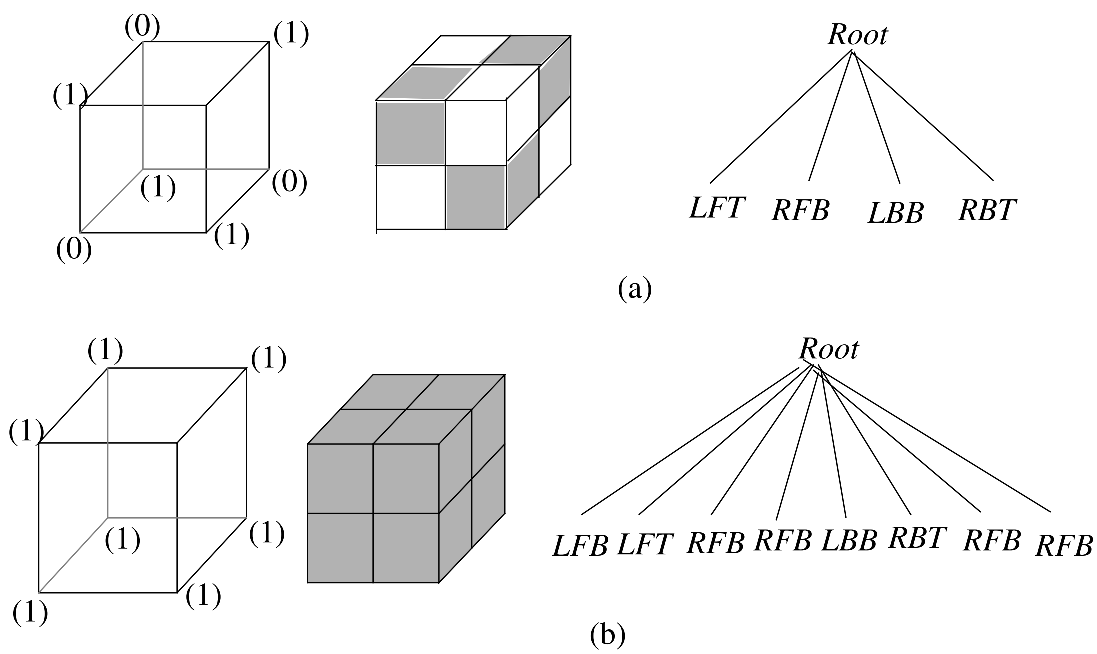


Figure 4.7 Fully refinable octants: (a) A fully refinable octant with 4 sub-octants. (b) A fully refinable octant with 8 sub-octants.

same.

3. (*modification of a mid-edge vertex*): Let p_0 and p_1 be two neighboring vertices of four neighboring octants, O_1, O_2, O_3 and O_4. Let m_e be the mid-edge vertex of (p_0, p_1), as shown in Figure 4.8(c). If all four octants are not refinable, then the L value at the vertex m_e is reduced to the level of the octant plus 1, i.e. $L(m_e) = Level(O)+1$; otherwise the L value at m_f remains the same.

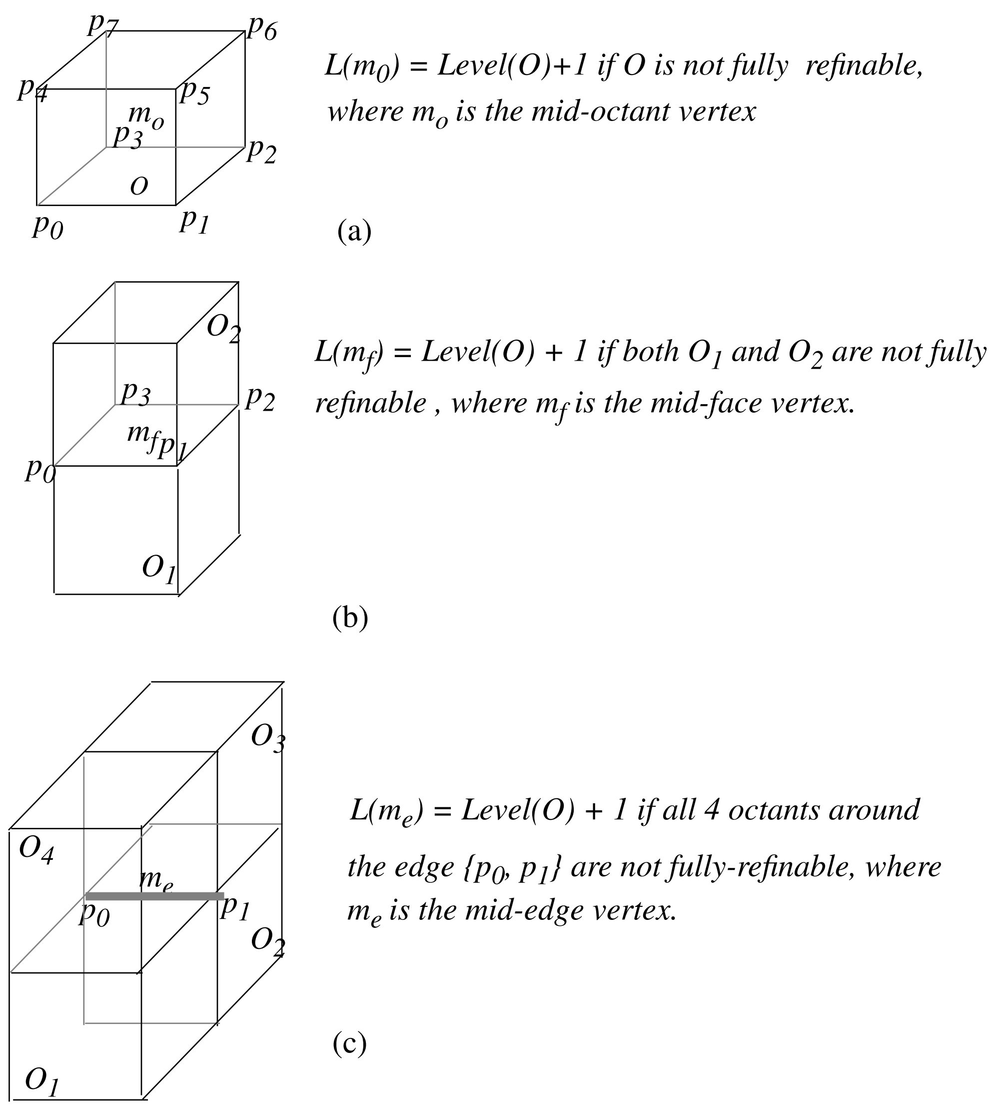


Figure 4.8 Modification of the level control function L: (a) Modification of the mid-octant vertex. (b) Modification of the mid-face vertex. (c) Modification of the mid-edge vertex.

The following three procedures outline the algorithm for generating a balanced octree based on the level control function. It can be shown that the balanced octree generated is optimal with respect to the level control function. The proof is similar to the one for the 2D quadtree generation algorithm presented in Section 3.3.5.

```
PROCEDURE   RefineOctant(O: Octant)

    (* modify the level control function values if necessary *)
    ModifyLCF(O);
    (* refine the sub-octants of O *)
    FOR each vertex p of the octant O DO
        Let subO be the sub-octant of Q containing p;
        RefineOctant(subO);
    END;

END         RefineOctant;

PROCEDURE   OctantIsRefinable(O: Octant: O): BOOLEAN

    FOR each vertex p of the octant O DO
        Let p_i, p_j, p_k be three neighboring vertices of p;
        IF L(p) = Level(O) THEN
            IF (  L(p_i) = Level(O)  OR
                  L(p_j) = Level(O)  OR
                  L(p_k) = Level(O)    ) THEN
                  RETURN TRUE;
    END;
    RETURN FALSE;

END         OctantIsRefinable
```

```
PROCEDURE   ModifyLCF(O: Octant)

    (* modify the level control function value for the mid-octant vertex *)
    IF (OctantIsRefinable(O) is not TRUE) THEN
        m = MidOctantVertex(O);
        L(m) = Level(O)+1;
    END;
    (* modify the level control function values for the mid-face vertices *)
    FOR each face F of the octant O DO
        Let O1 and O2 be two octants sharing F;
        IF OctantIsRefinable(O1) AND OctantIsRefinable(O2)  THEN
            m = MidFaceVertex(F);
            L(m) = Level(O) + 1;
        END;
    END;
    (* modify the level control function values for the mid-edge vertices *)
    FOR each edge E of the octant O DO
        Let Oi (i=1, 2, 3, 4)  be 4 octants sharing E;
        IF ( OctantIsRefinable(O1) OR
             OctantIsRefinable(O2) OR
             OctantIsRefinable(O3) OR
             OctantIsRefinable(O4)    ) THEN
            m = MidEdgeVertex (E);
            L(m) = Level(O) + 1;
        END;
    END;

END             ModifyLCF
```

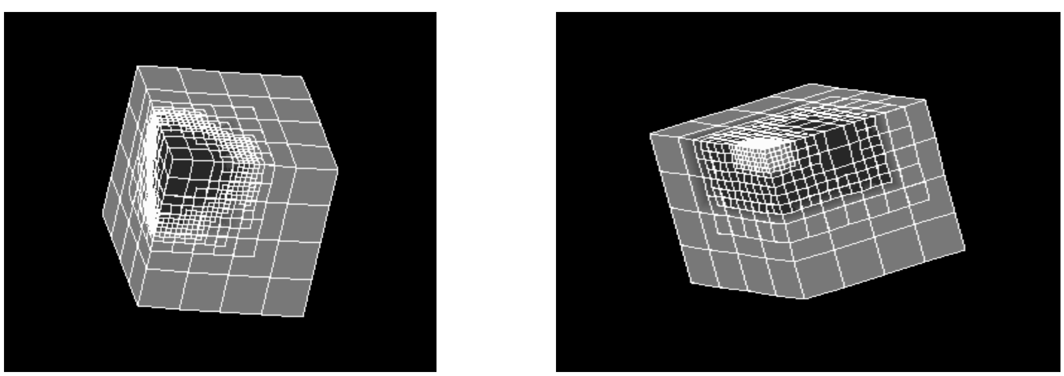
Two examples shown in Figure 4.9 illustrate the applications of the algorithm to generate octrees for semiconductor devices, where (a) is a p-n diode and (b) is an n-p-n$^+$

bipolar transistor. For both examples, the variation of the doping profile is used as the level control function to control the mesh density.

4.3 Tetrahedralization of Interior Octants

In the previous section, we discussed the algorithm to generate an optimal balanced octree for a given level control function. To generate a mesh, we need to tetrahedralize all leaf octants, i.e. decompose leaf octants into tetrahedrons. There are two types of leaf octants in an octree, δ-zone octants and interior octants. In this section, an algorithm for tetrahedralizing interior octants is presented. The basic idea is first to decompose the octant into a set of sub-regions, each of which belongs to one of five canonical types. Each sub-region is then decomposed into tetrahedrons.

The section is organized as follows: 4.3.1 provides an overview of the tetrahedralization procedure for interior octants. 4.3.2 describes the tetrahedralization procedure for a sub-region with no child octants. 4.3.3 describes the tetrahedralization procedure for a sub-region with child octants. In 4.3.4, we prove that the resulting mesh generated by the proposed algorithms satisfies the Delaunay criterion.



(a) (b)

Figure 4.9 Illustration of balanced octrees: (a) The octree for a p-n diode. (b) The octree for a n-p-n^+ bipolar transistor.

4.3.1 Overview

After an octree is being created, a tetrahedralization step is required to generate the final mesh. One constraint for tetrahedralization is the mesh conformity. In other words, a 3D mesh needs to satisfy that

(i) a mesh vertex is not inside any mesh edge or any mesh face or any mesh element;

(ii) any part of a mesh edge is not inside any mesh face or any mesh element;

(iii) any part of a mesh face is not inside any mesh element.

For a 2D quadtree based mesh, it is relatively easy to enforce the conformity due to the following property:

> *A quadtree based mesh is conformal as long as the triangulation is conformal within each quadrant of the quadtree.*

However, in 3D, the local conformity does not guarantee the global conformity, as illustrated in Figure 4.10. In the example, the geometry region consists of two pyramids and each of the two pyramids is decomposed (tetrahedralized) into 2 tetrahedrons. For each pyramid, the tetrahedralization is conformal. But the tetrahedralization is not conformal for the whole region because part of the edge $\{p_0, p_2\}$ is inside the face $\{p_0, p_1, p_3\}$, which is the face of another tetrahedron $\{p_0, p_1, p_3, p_5\}$.

Different conformal tetrahedralization schemes have been implemented [YERR84, SAXE90, CONT90]. In this section, a new tetrahedralization algorithm is proposed for

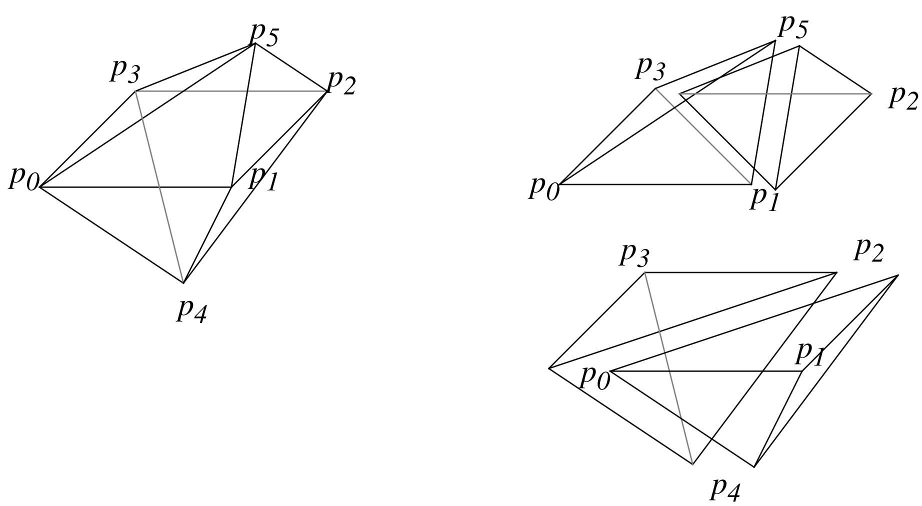


Figure 4.10 Non-conformal mesh in 3D.

interior octants of a balanced octree. The algorithm is computationally efficient and it guarantees the tetrahedron mesh to be conformal. Furthermore, it can be shown that the resulting mesh satisfies the Delaunay criterion.

The tetrahedralization is performed for each interior leaf octant. There are two types of leaf octants, namely regular and irregular octants. A regular octant does not have any sub-octants and an irregular octant contains one to seven sub-octants. For a regular octant, it is decomposed into six tetrahedrons. One of such decomposition is shown in Figure 4.11.

For an irregular octant, the tetrahedralization process is more complex. The following discussion provides an outline of the process. First we decompose the irregular octant into six (or less) sub-regions based on the octant's six faces, as illustrated in Figure 4.12. Then each sub-region is decomposed into a set of tetrahedrons. The example in Figure 4.13 depicts the entire tetrahedralization process for an irregular octant.

(a) (b)

Figure 4.11 Tetrahedralization of a regular octant: (a) The octant. (b) Decomposition to 6 tetrahedrons.

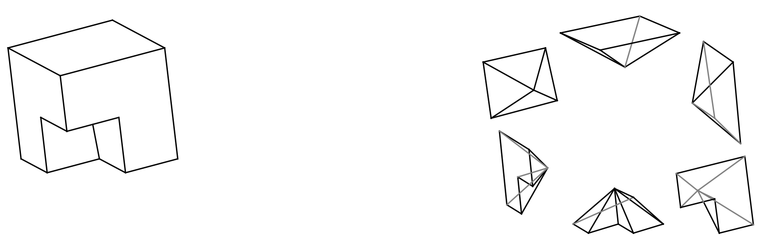


Figure 4.12 The sub-region decomposition of an irregular octant with one child octant.

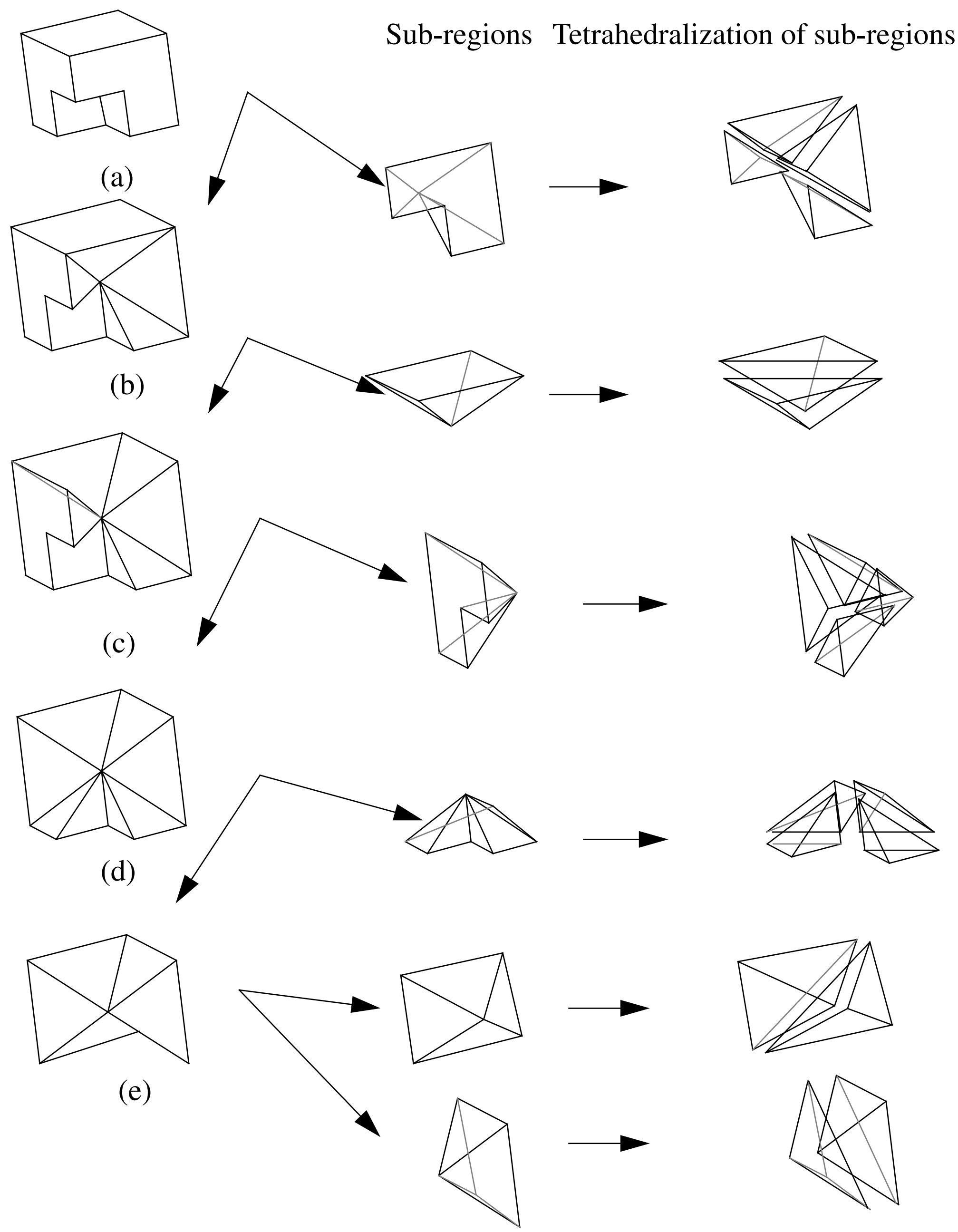


Figure 4.13 Tetrahedralization of an irregular octant: (a) Octant. (b) After the front face sub-region removed. (c) After the top face sub-region removed. (d) After the left face sub-region removed. (e) After the bottom face sub-region removal.

The above discussion reveals that the tetrahedralization of an irregular octant involves two steps: (1) decomposition of the octant into sub-regions and (2) decomposition of sub-regions into tetrahedrons. In the remaining part of this subsection, we concentrate on the first step and the next two subsections are devoted to the second step.

Each irregular octant can be decomposed into sub-regions by connecting the mid-octant vertex with each of the six faces. Sometimes, the face does not exist due to the existence of sub-octants. For the example shown in Figure 4.14, the bottom face does not exist for the octant. In such case, we have less than six sub-regions.

It can be shown that there are five canonical types of sub-regions, as illustrated in Figure 4.15. All other sub-regions can be transformed into one of these five types through translations and/or rotations. In the following discussion, we always assume that each of the canonical types contains the bottom face of the octant. The algorithm to decompose these five sub-regions into tetrahedrons is presented in the next two subsections.

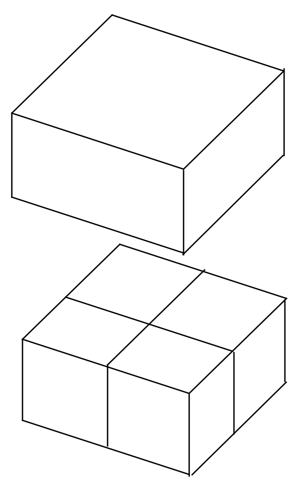


Figure 4.14 An example of an irregular octant without the bottom face.

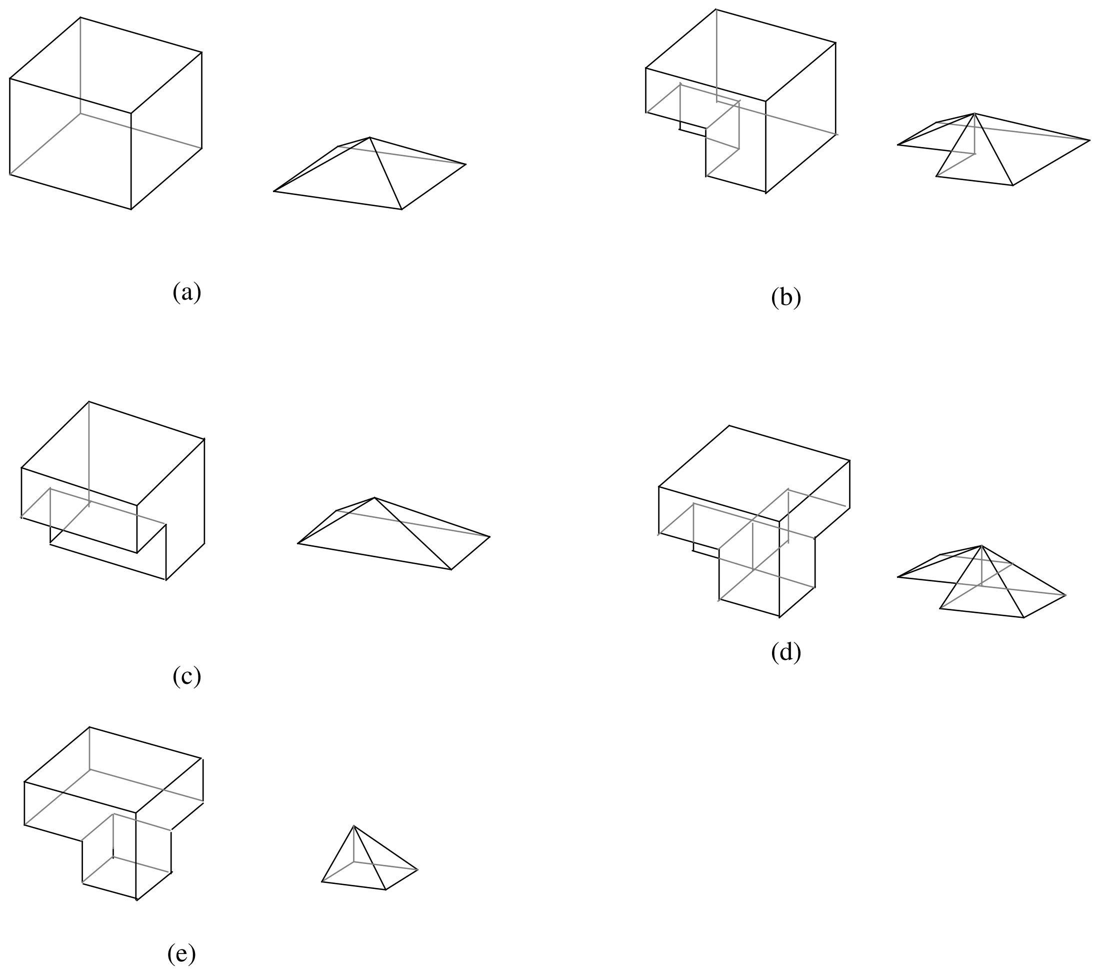


Figure 4.15 Five types of sub-regions: (a) With no child octants. (b) With one child octant. (c) With two neighboring child octants (d) With two diagonal opposite child octants(e) With three child octants

We conclude this subsection with a procedure which summarizes the tetrahedralization algorithm for an interior octant, where *DecomposeSubregionWithoutChildOct* and *DecomposeSubregionWithChildOct* are described in Section 4.3.2 and Section 4.3.3, respectively.

```
PROCEDURE   TetrahedralizeInteriorOctant(O: Octant)

    IF (IsRegular(O)) THEN
        (* decompose the regular octant into 6 tetrahedrons. *)
        DecomposeRegularOct(O);
    ELSE
        FOR each sub-region R of the octant O DO
            IF ChildOctNumForSubregion(R) = 0 THEN
                TetrahedralizeSubRegionWithoutChildOct(F, O);
            ELSE
                TetrahedralizeSeubRegionWithChildOct(F, O);
            ENDIF;
        END;
        (* recursively tetrahedralize all sub-octants *)
        FOR each sub-octant S of the octant O DO
            TetrahedralizeInteriorOctant(S);
        END;
    ENDIF;

END         TetrahedralizeInteriorOctant
```

4.3.2 Sub-Region Tetrahedralization 1

In this subsection, we discuss how to create tetrahedrons for the type I sub-region, i.e. the sub-region without child octants. Since the sub-region is a pyramid, it can be split into two tetrahedrons as shown in Figure 4.16. However it is not difficult to see that the mesh generated by the simple algorithm does not ensure the Delaunay criterion for the case as shown in Figure 4.17, where the two octants O_1 and O_2 share the same face F and the two sub-regions containing the face F are both of type I. To ensure the Delaunay criterion, we use the decomposition as shown in Figure 4.18.

The following procedure describes the complete algorithm for decomposing a type I sub-region into tetrahedrons.

Figure 4.16 The type I sub-region: (a) The canonical sub-region. (b) Splitting the pyramid into two tetrahedrons.

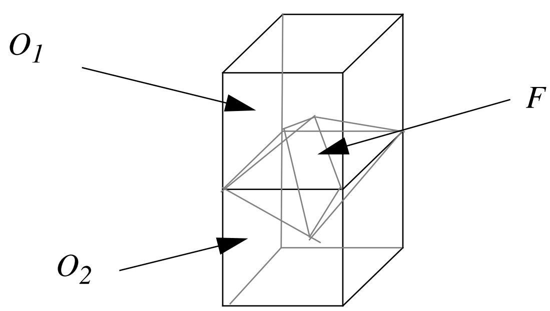


Figure 4.17 The decomposition which violates the Delaunay criterion.

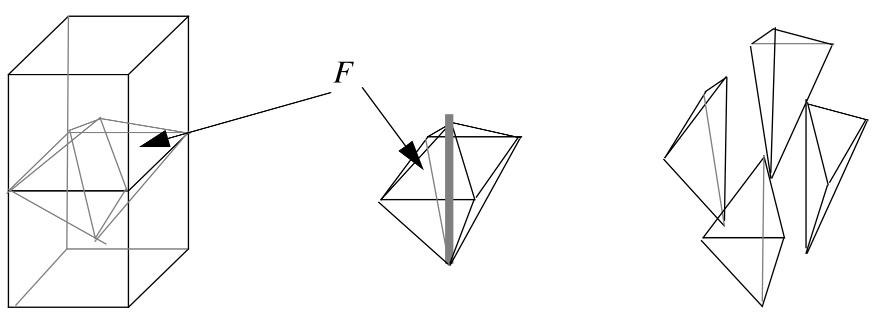


Figure 4.18 Four tetrahedrons shared by two octants.

PROCEDURE TetrahedralizeSubregionWithoutChildOct(*F*: OctantFace, *O*: Octant)

```
Let p_i (i=0, 1, 2, 3) be 4 vertices of F.
Let O' be the other octant sharing the face F with O;
IF the sub-regions for F inside both O and O' are of type I THEN
    Let m_o and m_o' be mid-octant vertices for O and O';
    FOR i := 0 to 4  DO
        j = (i+1) module 4;
        CreateNewTetra(m_0, m_o', p_i, p_j);
    END'
ELSE
    CreateNewTetra(m_0, p_0, p_1, p_2);
    CreateNewTetra(m_o, p_0, p_3, p_2);
ENDIF;
```

END TetrahedralizeSubregionWithoutChildOct

4.3.3 Sub-Region Tetrahedralization 2

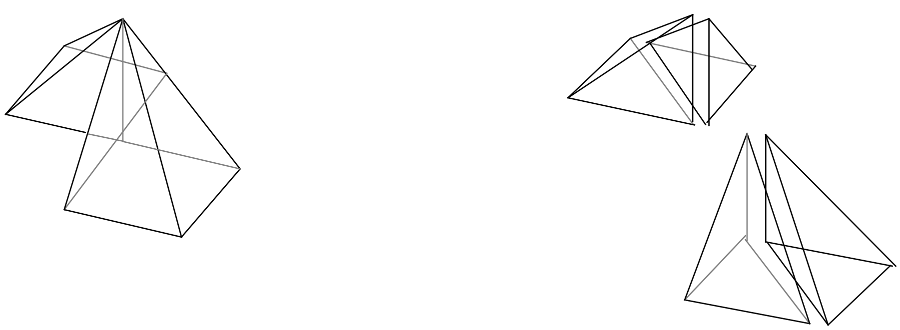
This subsection describes the tetrahedralization for sub-regions of types II, III, IV and V. For type II, four tetrahedrons are created, as shown in Figure 4.19. For type III, there are three tetrahedrons, as shown in Figure 4.20. For type IV, there are four tetrahedrons, as shown in Figure 4.21. For type V, there are two tetrahedrons, as shown in Figure 4.22.

(a) (b)

Figure 4.19 Tetrahedralization of the type II sub-region: (a) Type II sub-region. (b) Decomposition to four tetrahedrons.

(a) (b)

Figure 4.20 Tetrahedralization of the type III sub-region: (a) Type III sub-region. (b) Decomposition to three tetrahedrons.



(a) (b)

Figure 4.21 Tetrahedralization of the type IV sub-region: (a) Type IV sub-region. (b) Decomposition of four tetrahedrons.

The following procedure describes the process that tetrahedralizes type II, III, IV or V sub-regions. In the description of the procedure, the vertices p_i *(i=0, 1, 2, 3)*, m_i *(i=0, 1, 2, 3)* and m_o are depicted in Figure 4.23.

Figure 4.22 Tetrahedralization of the type V sub-region: (a) Type V sub-region. (b) Decomposition into two tetrahedrons.

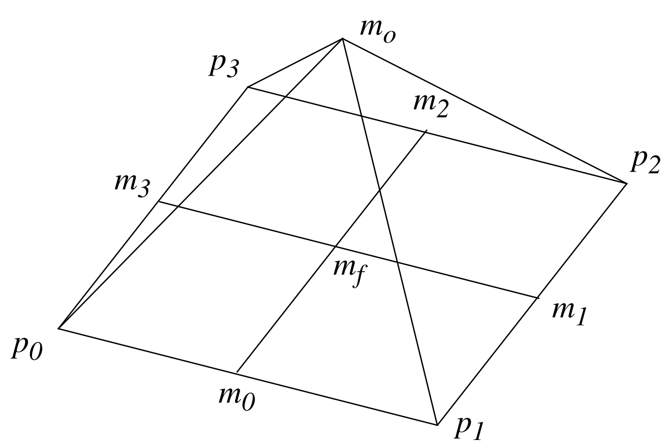


Figure 4.23 Vertex labels for a sub-region.

```
PROCEDURE   TetrahedrazeSubregionWithChildOct(F: FACE, O: Octant)

    type = GetSubregionType(F, O);
    (* convert sub-region into one of five canonical type *)
    F = ConvertToCanonicalForm(F, O);
    Let p_i(i=0, 1, 2, 3) be the 4 vertices and m_i (i=0, 1, 2, 3) the 4 mid-edge vertices;
    Let m_f be the mid-face vertex and m_o the mid-octant vertex;
    SWITCH the type of Subregion
    CASE II
        BEGIN
            createTetra(m_0, p_1, m_f, m_o); createTetra(p_1, p_2, m_f, m_o);
            createTetra(p_2, p_3, m_f, m_o); createTetra(p_3, m_3, m_f, m_o);
        END
    CASE III
        BEGIN
            createTetra(m_1, p_2, m_f, m_o);
            createTetra(p_2, p_3, m_f, m_o);
            createTetra(p_3,m_3, m_f, m_o);
        END
    CASE IV
        BEGIN
            createTetra(m_0, p_1, m_f, m_o); createTetra(p_1, m_1, m_f, m_o);
            createTetra(m_2,p_3, m_f, m_o); createTetra(p_3, m_3, m_f, m_o);
        END
    CASE V
        BEGIN
            createTetra(m_0, p_1, m_f, m_o); createTetra(p_1, m_1, m_f, m_o);
        END
    END OF SWITCH;


END             TetrahedralizeSubregionWithChildOct
```

4.3.4 Verification of the Delaunay Criterion

In this subsection, we show that a mesh generated by the algorithms described in the last two subsections satisfies the Delaunay criterion of the *in-circle* condition, stated as follows:

A tetrahedron mesh satisfies the in-circle criterion if and only if no vertex of any tetrahedrons in the mesh is interior to the circumsphere of any other tetrahedrons.

The proof consists of three parts. First we prove that the criterion is true for the decomposition of a regular octant, i.e. the octant without any child octants. Then we prove that it is true for the decomposition of the sub-region of type I. Finally we prove that it is true for the decomposition of the sub-regions of types II, III, IV and V.

Lemma 4.1 Suppose O is a leaf regular octant, as shown in Figure 4.24 and the octant is decomposed into six tetrahedrons. Interior to the circumsphere of any of the six tetrahedrons, there are no other mesh vertices.

Proof. The circumspheres for all 6 tetrahedrons are the same. Each circumsphere passes through all 8 vertices of the octant. The center for the sphere is the center of the octant. If the length of the octant is $2a$, then the radius of the sphere is $a\sqrt{3}$. Since the octant does not have any sub-octants, there are no mid-octant, mid-face and mid-edge vertices in the octant. Thus the closest vertex to the sphere other than the 8 vertices of the octant are at least $2a$ away from the center of the sphere. This implies that all mesh vertices other than the ones for the octant are outside the sphere. ❑

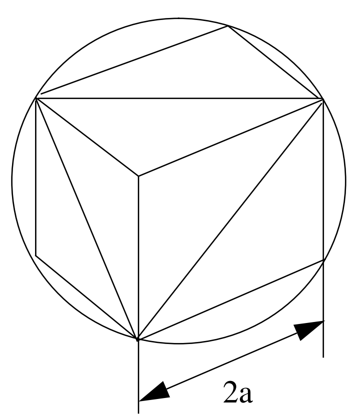


Figure 4.24 Proof of *Lemma 4.1*

Lemma 4.2   Suppose O is an irregular octant with a bottom face F and the sub-region containing the face F is of type I, as shown in Figure 4.25. Then for any tetrahedron contained in the sub-region, the interior of its circumsphere does not contain any other mesh vertices.

Proof.   There are two different decompositions for the type I sub-region, as shown in Figure 4.25(a) and Figure 4.25(b). In each case, we can calculate the exact location of the center and the radius of the tetrahedron's circumsphere. We can then calculate the distance from the mesh vertices to the center of the circumsphere. By comparing the distances between the closest vertex to the tetrahedron and to the center of the circumsphere, we can verify that there is not any mesh vertex located inside the circumsphere. This completes the proof. ❑

Lemma 4.3   For the decomposition of the sub-regions of types II, III, IV and V, interior to the circumsphere of any tetrahedrons, there are no other mesh vertices.

Proof.   The proof is very similar to the proof for *Lemma 4.2* by calculating the center and the radius of the tetrahedron's circumsphere. We can then show that there is no mesh vertices located inside any of the tetrahedron's circumspheres. ❑

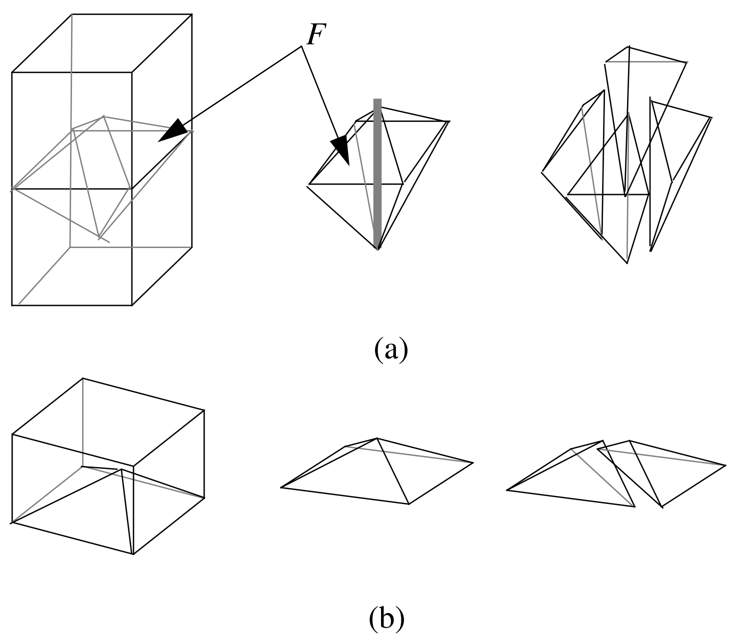


Figure 4.25 Proof of *Lemma 4.2*

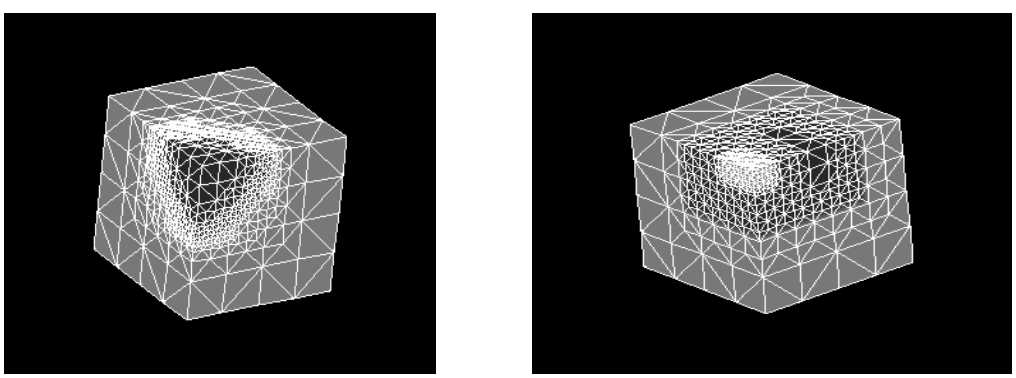
Theorem 4.1 Suppose *OT* is a balanced octree generated by the algorithm described in 4.2.2 and the corresponding mesh is generated by the algorithms described in 4.3.1, 4.3.2 and 4.3.3. Then the mesh satisfies the Delaunay criterion.

Proof. The proof is the combination of *Lemma 4.1*, *Lemma 4.2* and *Lemma 4.3*. ❑

We conclude this section with two examples which illustrate the tetrahedralization for interior octants. Two examples are a p-n diode and a n-p-n$^+$ bipolar transistor, as shown in Figure 4.26.

4.4 δ-Zone Tetrahedralization in 3D

The δ-zone definition in 3D is very similar to the definition described in Section 3.5.2 except that we replace a quadrant by an octant. The emphasis for this section is on the tetrahedralization for a δ-zone. As discussed in Section 3.5.3, a δ-zone on a 3D region can be decomposed into a set of boundary rectangles, each of which consists of a set of δ-zone quadrants. In 3D, we decompose a δ-zone into a set of δ-*zone bricks*, also called bricks. Each of the δ-zone bricks consists of a set of δ-zone octants. By decomposing each brick into tetrahedrons, a δ-zone in 3D is tetrahedralized.



(a) (b)

Figure 4.26 Illustration of interior meshes: (a) Mesh for a p-n diode. (b) Mesh for a n-p-n$^+$ bipolar transistor.

This section is organized as follows. In 4.4.1, an overview for tetrahedralization is presented. In 4.4.2, a tetrahedralization algorithm for simple δ-zone bricks is described and in 4.4.3, a general tetrahedralization algorithm for polyhedrons is presented.

4.4.1 Overview of Tetrahedralization

The process of tetrahedralization of a δ-zone is divided into two steps. First we decompose the zone into a set of bricks. Then each brick is decomposed into tetrahedrons. Each brick of a δ-zone consists of a number of δ-zone octants. Several brick examples are shown in Figure 4.27. To perform a tetrahedralization for a brick is much more complex than to tetrahedralize interior octants because the geometry of a brick is arbitrary. In the following two subsections, we discuss two different strategies for tetrahedralizing δ-zone bricks. The first one is computationally efficient but only applicable to certain geometry shapes. The second one is based on the algorithm by Woo which decomposes a general polyhedron into tetrahedrons [WOO84]. Our implementation combines both techniques to ensure that a general δ-zone is tetrahedralized in an efficient manner. The example shown

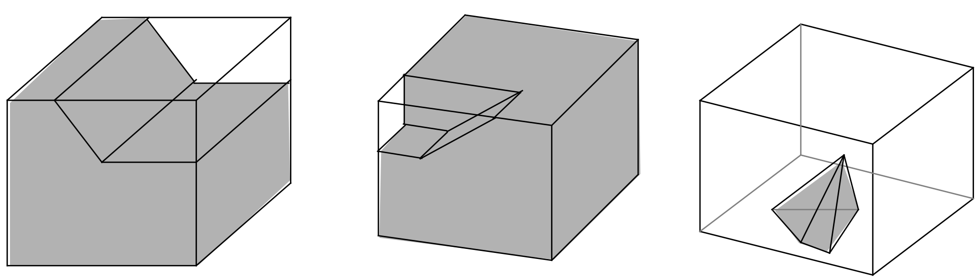


Figure 4.27 Examples of boundary bricks.

in Figure 4.28 illustrates the result of the implementation, where the mesh is generated for a MOSFET transistor with a trench.

The following procedure outlines the computational process for tetrahedralizing a δ-zone brick, where *SimpleDecomposition* is discussed in Section 4.4.2 and *ComplexDecomposition* in Section 4.4.3.

PROCEDURE TetrahedralizeBoundaryBrick(*B*: Brick)

IF *One boundary face crosses B* THEN
SimpleDecomposiion(B);
ELSE
ComplexDecomposition(B);
ENDIF;

END TetrahedralizeBoundaryBrick

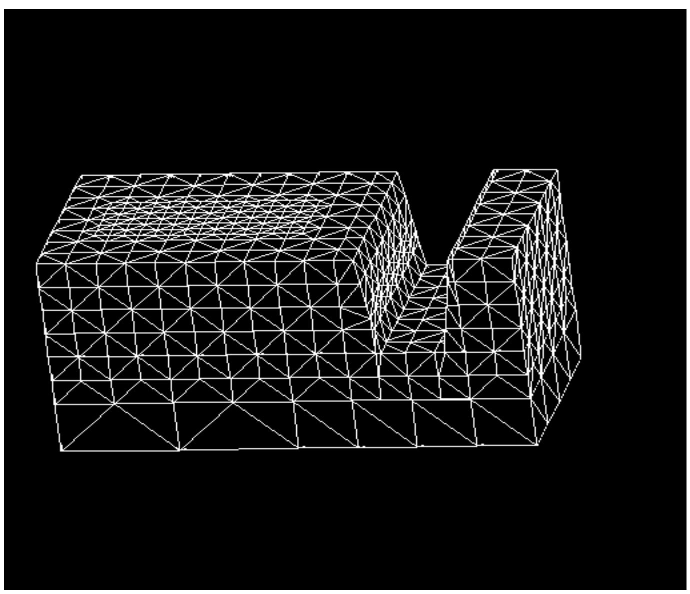
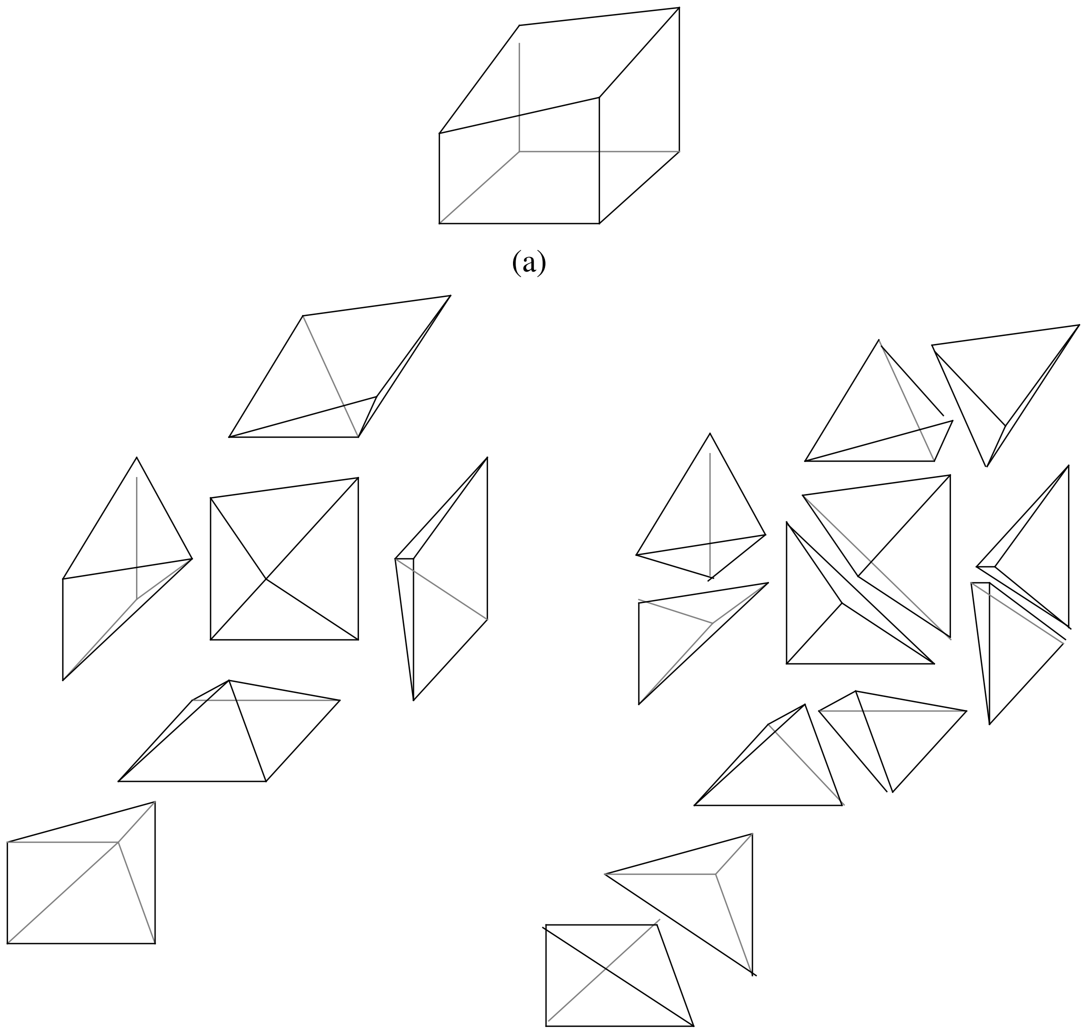


Figure 4.28 A mesh for a MOSFET transistor with a trench.

4.4.2 Tetrahedralization of Simple Boundary Bricks

There are three types of simple δ-zone bricks which can be easily decomposed into a set of tetrahedrons. The three types of bricks (Type 1, 2, 3) are characterized by the number of boundary faces intersecting with a brick. In the rest of the section, we discuss the tetrahedralization process for each type. If a δ-zone brick does not belong to any of these three types, a general tetrahedralization technique is applied which is discussed in the next subsection.

The first type (Type 1) is a brick which intersects with only one boundary face of the mesh region as shown in Figure 4.29(a). For this type of bricks, we first calculate the center of the region within the brick and decompose the region into a set of pyramids by connecting the center with each boundary face of the region, as shown in Figure 4.29(b). Then we triangulate each pyramid's base using the Delaunay triangulation algorithm discussed in Section 3.5.3 and decompose each pyramid into a set of tetrahedrons, as shown in Figure 4.29(c).



(a)

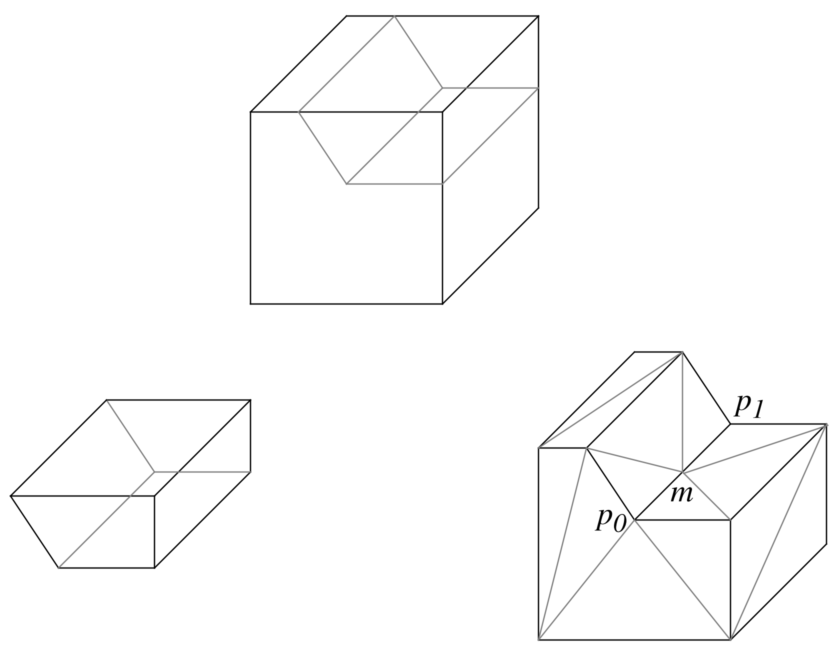
(b) (c)

Figure 4.29 Tetrahedralization of a boundary brick which intersects with one boundary face: (a) A brick of Type 1. (b) Decomposition into a set of pyramids. (c) Decomposition into tetrahedrons.

The second type (Type 2) of brick intersects two boundary faces and the intersection edge of these two boundary faces goes through the brick as shown in Figure 4.30. A brick of type 2 falls into two sub-groups. For bricks in the first sub-group, the mesh region contained in the brick is convex as shown in Figure 4.30(a). For bricks in the second sub-group, the mesh region contained in the brick is not convex as shown in Figure 4.30(b). If the region is convex, we use the same algorithm to perform tetrahedralization as discussed before for the type 1 bricks.

For a non-convex region as shown in Figure 4.30(b), we calculate the center m of the edge p_0p_1 where the edge p_0p_1 is the intersection edge for two boundary faces. Then by connecting the vertex m with each of the brick's faces except for the two faces which contain m, we can decompose the mesh region into a set of pyramids. Each pyramid can be decomposed into a set of tetrahedrons by triangulating the base of each pyramid.

The third type (Type 3) of bricks intersects three boundary faces and the intersection point of the three boundary faces is inside the brick as shown in Figure 4.31. Bricks of

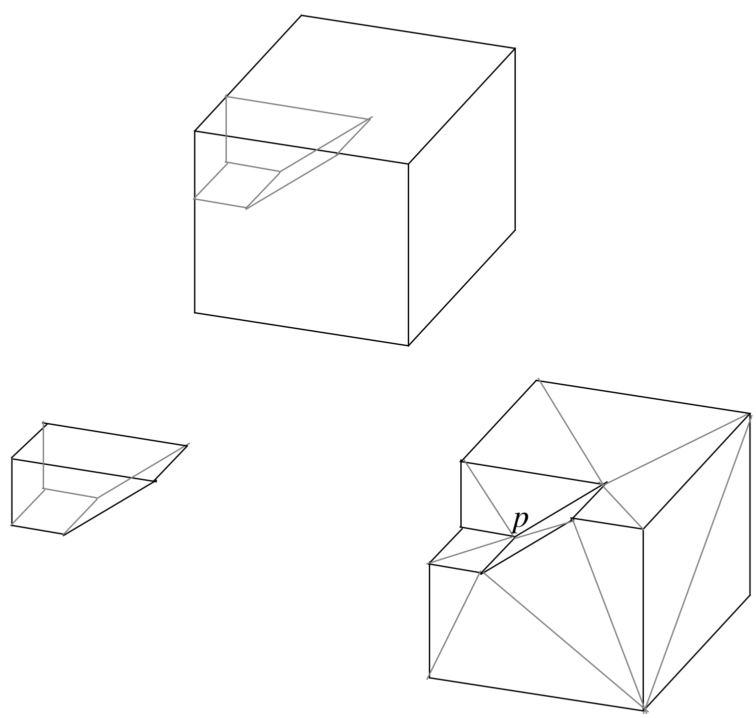


(a) *(b)*

Figure 4.30 A brick intersects two boundary faces: (a) The region is convex.
(b) The region is not convex.

type 3 falls into 2 sub-groups. For a brick in the first sub-group, the mesh region contained in the brick is convex as shown in Figure 4.31(a). For a brick in the second sub-group, the region is not convex as shown in Figure 4.31(b). If the mesh region is convex, the same algorithm for type 1 bricks as discussed previously can be applied.

If the mesh region is not convex, we select the intersection vertex p of three boundary faces, as shown in Figure 4.31(b). By connecting p with each face of the region except for the three faces which touch the vertex p, we can decompose the region into a set of pyramids. We then decompose each pyramid into a set of tetrahedrons.



(a) (b)

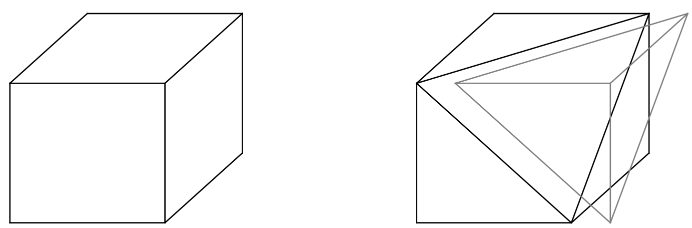
Figure 4.31 A simple brick intersecting three boundary faces: (a) The region is convex. (b) The region is not convex.

4.4.3 A General Tetrahedralization Algorithm for Polyhedrons

In the previous subsection, we present a tetrahedralization algorithm for simple δ-zone bricks. These algorithms are simple to implement and computationally efficient. But there are cases where these algorithms are not applicable. In this subsection, we present an algorithm based on the algorithm by Woo for tetrahedralization of a general polyhedron [WOO84].

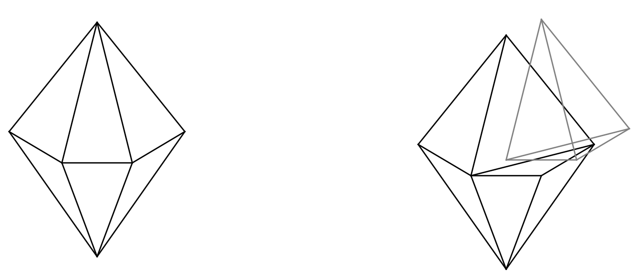
First we need to introduce two operators, the vertex removal operator and the edge removal operator, which are used to remove a tetrahedron one at a time from the polyhedron, as illustrated in Figure 4.32 and Figure 4.33.

A vertex removal operator is used if the polyhedron has a vertex with three incident edges (trivalent). We can form a tetrahedron from the vertex and the three edges. If the resulting tetrahedron is contained in the polyhedron, then we reduce the polyhedron by removing the tetrahedron.



(a) (b)

Figure 4.32 The vertex removal operator: (a) Before applying the vertex removal operator. (b) After applying the vertex removal operator.



(a) (b)

Figure 4.33 The edge removal operator: (a) Before applying the edge removal operator. (b) After applying the edge removal operator.

An edge removal operator is used if there is a convex edge in the polyhedron, as shown in Figure 4.33. Using the convex edge, we can form a tetrahedron which is inside the polyhedron. By removing the tetrahedron, we can reduce the polyhedron.

We can repeatedly apply the vertex removal and the edge removal operators until the whole polyhedron is decomposed into a set of tetrahedrons. The decomposition process is summarized by the following procedure.

```
PROCEDURE   TetrahedralizePolyhedron(P: Polyhedron)

    IF P is not a tetrahedron THEN
        WHILE P is not a tetrahedron DO
            IF VertexForRemovalExists(P) THEN
                Tetra = VertexRemovalOperator(P);
                P = Subtract(P, Tetra);
            ELSE
                Tetra = EdgeRemovalOperator(P);
                P = Subtract(P, Tetra);
            ENDIF
        END;


END         TetrahedralizePolyhedron
```

4.5 Summary

In this chapter, we extended the concepts and algorithms discussed in Chapter 3 to 3D mesh generation. A major challenge for the extension is the tetrahedralization of octants after a balanced octree is generated. The tetrahedralization algorithm proposed in this chapter for interior octants guarantees that the resulting mesh for the interior region satisfies the Delaunay criterion. The combination of a simple but efficient tetrahedralization algorithm with a general polyhedron decomposition algorithm ensures that the δ-zone can be tetrahedralized in an efficient manner.