SIMPLICIAL MESH GENERATION WITH APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Douglas William Moore

August 1992

SIMPLICIAL MESH GENERATION WITH APPLICATIONS

Douglas William Moore, Ph.D.

Cornell University 1992

Many problems in computer graphics and in engineering analysis require for their solution the construction of a mesh of simple polytopal elements that approximately fill the interior of an object or cover its boundary. The simplest polytopes are the simplexes, and simplicial meshes have particular advantages when solving interpolation problems, graphically rendering objects approximated by boundary meshes, and in many other applications. Boxes are often used where simplexes should be considered because the properties of boxes are more familiar.

This dissertation develops several techniques that aid in the construction of simplicial meshes in any dimension. It presents the essential combinatorial and geometric properties of simplexes, and presents simple techniques for decomposing simplexes into smaller simple objects. It describes the simplicial quadtree, a useful representation in the construction of simplicial meshes, and presents a technique for converting an arbitrary simplicial quadtree into a balanced quadtree, then into a triangulation. It also describes mesh displacement, a technique for improving the quality of a boundary triangulation while reducing its size. Several degenerate

behaviors can arise from mesh displacement, and the dissertation discusses methods for detecting and compensating for these degeneracies.

Simplicial mesh generation techniques can be applied to many kinds of problems. To illustrate this, the final chapters describe the implementation of a polygonalizer for algebraic sets and a system that applies the operations of constructive solid geometry to sets defined algebraically.

Biographical Sketch

Doug Moore was born 29 November, 1960, in Arlington, Texas, to Charles and Lavella Moore. With his parents and his sister, Lisa, he lived in Arlington for most of eighteen years. He attended Arlington public schools and was the valedictorian of his 1979 Arlington High School class.

From 1979 to 1983, he studied at Rice University in Houston, Texas, majoring in computer science and mathematical sciences. He graduated *magna cum laude* with a Bachelor of Arts degree in 1983.

From 1983 to 1989, he lived in Ithaca, New York, while a graduate student at Cornell Univerity. He received a Master of Science degree in computer science from Cornell in 1987. In his early graduate work, he studied parallel algorithms for solving sparse linear systems.

He took a position at lecturer and research associate in computer science at Rice University in 1989, where he taught intermediate programming courses and studied combinatorial and computational aspects of geometry. He received his Ph. D. in computer science from Cornell University in 1992 and vowed thereafter to lead a richer and more well-rounded life.

To Mrs. B., a great teacher who taught more than she knew.

Acknowledgements

This dissertation could not have been completed without many people helping me, loving me, or nagging me at the right times. I am lucky to have such good friends.

Thanks go to my advisors Stephen Vavasis, John Gilbert, Tom Coleman and Robert Bland for their forbearance. Michael J. Todd was not on my committee, but in our few conversations I got some very helpful perspectives. Many other people made my stay in Ithaca pleasant, including Mark, Jim, Mark, Steve and Mark. The heart of that group, though, was Chris Buckley and Ellen Voorhees and the Friday picnics and games and pizzas. I thank them for the warm, friendly atmosphere they created for a Texas boy in the frozen northland.

I have plenty of people at Rice to thank, too. Ken Kennedy taught me well the first time I was here and made possible the miracle that let me return to Rice to teach and learn. I thank Corky Cartwright for that miracle as well. Michael Pearlman was a good teacher, but only on my return have I learned what an excellent and loyal friend he is too. His gentle nagging has helped me finish this dissertation. I am grateful to have friends like Keri, Mark (again), Bob and John as well.

Back home, my family still loves me. That is an anchor I can hold onto when times are bad, and, of course, I owe them all that I am. My sister is a wonderful, honest person who can do anything she puts her mind to. My father has taught me much about how to treat people. My mother is kindest, most patient person I will ever know. I love them all, and hope they'll forgive me for not telling them more often.

The biggest debt of gratitude I owe, however, to Joe Warren. He has been, at various times over the last decade, my roommate, classmate, colleague, teacher and advisor. He has always been my friend. Our work together over the last three years has inspired this dissertation. From him I have learned much about mathematics and geometry, about academia, and also much about the value of self-confidence. I intend to repay my debt to him with a lifetime of brotherly love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Computing exact solutions to problems involving physical objects is difficult. One reason is that most physical objects do not have simple descriptions. Another is that the physical laws that govern the behavior of objects rarely lead to simple systems of equations that can be solved exactly. Practical algorithms for solving such problems generally manipulate approximations to the physical objects and simplified equations that model the actual problem well enough, but not precisely. Mesh generation underlies many such algorithms.

Informally, a mesh is a collection of simple elements that fit together to form an approximation to a possibly complex solid. Depending on the application, the elements may be two or three dimensional. Depending on programming decisions and special problem requirements, the elements may be polyhedral, bounded entirely by flat surfaces, or curved, bounded by surfaces of low algebraic degree. They may be uniformly of one shape, or of several different shapes. They may be arbitrarily polygonal, or of only a few sides. They may be boxlike or triangular.

Applications of mesh generation are many. In engineering, the finite-element method is a basic tool for analyzing the mechanical or thermal properties of an object. It treats a complex object as a collection of simple objects with easily calculable properties in isolation, and simple, usually linear, relationships among adjacent elements. In computer graphics, meshes often represent smooth curved surfaces, and techniques like Gouraud shading and Phong shading help obscure the edges in the mesh, to make the rendered surface appear more realistic. In computer-aided geometric design, an automobile body panel, a ship hull, or an airplane wing are likely to be represented by a mesh of surface patches. In solid modeling or robot path planning, complex intersection problems can often be expressed as a collection of simpler element intersection problems, if a mesh decomposition of the primary objects is available.

The properties that classify a mesh good or bad are mostly independent of application. The next section defines some of these properties and illustrates why their presence is desirable in applications like those above. The following section describes and evaluates several general algorithmic approaches to mesh generation. The final section states the thesis of this dissertation; that the hierarchical decomposition of simplices forms the basis of a practical approach to mesh generation.

1.2 Desirable Mesh Properties

Several properties characterize high quality meshes. These properties are not, in general, mutually compatible. A mesh ideal in one property is likely to be hopelessly deficient in another, so that, in practice, someone must decide appropriate tradeoffs between them.

One measure of mesh quality is element shape. Rounder, more nearly regular elements are preferred to elements that are much smaller in one dimension than in another. In finite element methods, for example, a mesh of distorted elements produces an ill-conditioned system of equations that relate adjacent elements, and leads to inaccurate numerical solutions. In computer graphics, distorted mesh elements produce abrupt changes in brightness when Gouraud shading is used, degrading the quality of the image. Regular elements are ideal by this criterion.

For representing geometry of mixed complexity, adaptivity is an essential mesh property. Physical objects are usually simple and smooth in some parts, complex and contorted in others. For accuracy in engineering calculations, for realistic graphical rendering, or for detail in geometric design, elements in a region of complex geometry must be sufficiently small. However, limited computational resources may make it impossible to manipulate a mesh made uniformly of elements small enough to model accurately the most complex region. Thus, small elements and large elements must be able to coexist in a mesh. Collections of well-shaped elements of various sizes are easy to build, so that the first two properties are not mutually incompatible.

Another important property, often called conformality, describes how elements fit together. A collection of elements is conformal if any two elements intersect in a common face; a mesh denotes a conformal collection of elements. In graphics, the absence of conformality can combine with small numerical inaccuracies to produce visible cracks in rendered surfaces. In finite element analysis, it reduces the sparseness of large linear systems and can introduce linear dependencies. In geometric design, it complicates continuity conditions in the design of smooth piecewise polynomial surfaces. However, conformality cannot coexist with shape-regularity and adaptivity. Shape-regularity requires that all edges of an element have about the same length. Conformality requires that edges of neighboring elements have about the same length. Adaptivity requires short edges and long edges.

The algorithms that generate a mesh should be efficient. The efficiency of a mesh generator should be measured not only by its running time as a function of input size, but also by its running time as a function of output size. The input to the mesh generator may be small, as the equation $x^2 + y^2 + z^2 = 1$ is a short description of a sphere. A graphics program to display the image of a sphere may produce hundreds or thousands of elements from that equation. Likewise, adaptive finite element meshes must satisfy error criteria that have short descriptions but can affect the size of the mesh dramatically. A mesh generator should be considered efficient if its running time is proportional to the size of the mesh it produces.

The data structures that represent a mesh should be simple. This is a practical consideration, not a theoretical one. Mesh generators are often written, used, and customized by people with little knowledge of advanced algorithmic techniques. Until software can be packaged in a way that provides extensibility without granting access to source code, complex mesh generation algorithms will be slow to gain acceptance.

1.3 Common Mesh Generation Techniques

Shephard [She88] identifies four basic approaches to automatic mesh generation:

1. removal of individual subdomains,
2. recursive subdivision of the domain, and
3. point placement followed by triangulation,
4. spatial decomposition followed by subdomain meshing.

The first two approaches work with boundary representations of solids. The first decomposes a solid into simple components by identifying a topological feature, like a vertex of low degree or an edge of a certain type, and breaking the neighborhood of the feature away from the rest of the object, then decomposing the rest of the object similarly. The second works by triangulating the boundary, then tracing a cycle of edges on the boundary to divide the solid into parts of roughly equal complexity. After triangulating the splitting face common to both parts, the process is repeated on each part. Ultimately, the solid is represented as a hierarchical collection of simple parts.

Point placement techniques have been more extensively studied than the first two. The two basic steps of these techniques are, first, the distribution of points that become vertices of the final mesh throughout the domain and, second, the construction of a mesh by grouping sets of the vertices into edges, faces and cells. The distribution of points provides adaptivity, as various techniques are used to put more points in regions that require a fine mesh and fewer where a coarse mesh suffices. The second part of the process relates to classical problems in computational geometry, like the computation of the Delaunay triangulation and various other minimum weight and angle constrained triangulations. Delaunay triangulations are often unsuitable in three dimensions, so Field [Fie86] has developed a technique for distributing points in way that makes Delaunay triangulations acceptable. Chew [Che89] and Bern, Epstein and Gilbert [BEG90] have developed efficient algorithms to produce meshes with guaranteed shape quality, given a planar polygonal region and an arbitrary point set, respectively.

Spatial decomposition techniques usually begin with a single box enclosing the domain, and then adaptively refine boxes to produce a quadtree [Sam90]. The method of Yerry and Shephard [YS84] takes a quadtree satisfying certain properties and divides its squares or cubes in several different ways to produce a mesh of triangles or tetrahedra for finite element analysis. Von Herzen and Barr have used a similar approach to build partitions of parameter space for use in parametric surface rendering [VB87].

1.4 Box versus Simplex

For ease of programming, uniform spatial decomposition based on squares or cubes is hard to beat. Data structures like the arrays present in most languages are isomorphic to uniform square and cubic meshes, so programmers naturally tend to favor them in the absence of other considerations. The iterative control structures of traditional languages make examining each element of a mesh simple. The ubiquitous square grid is even influencing the design of multiprocessors; because so many numerical programs are formulated on square grids, Intel has replaced the hypercube interconnection network by the square grid in its newest multiprocessors.

Meshes of squares, however, do not permit both adaptivity and conformality. In some applications, a square mesh is called adaptive if it is possible to refine it by bisecting all the squares in a column with a single line, and possibly repeating the process to produce rectangular elements. Refining a single square several times in both dimensions can produce sufficiently small elements, but the effect is not just local. Elements distant from the area of refinement are also split into narrow rectangles. Conversely, quadtree based methods like those of Yerry and Shephard initially sacrifice conformality, by permitting adjacent elements of different sizes, and eventually restore it when boxes are broken into simplices.

Meshes of triangles are sometimes used in two-dimensional applications. For example, Bank, Sherman and Weiser [BSW83] consider both square and triangular meshes. However, general simplicial meshes are not well understood. Senechal [Sen81] reports that for centuries, mathematicians were divided on the question of whether a mesh of regular tetrahedra could fill three-dimensional space (they cannot). A book by Coxeter [Cox73] contains the answers to many questions about meshes in higher-dimensional spaces, and classifies the simplices that can fill space. None of these authors present general algorithms for simplex manipulation and simplicial mesh generation that could make simplices as popular as boxes.

The goal of this dissertation is to show that simplices are not much harder to understand than boxes, even in high dimensions, and that programs to manipulate simplices need not be inefficient or complex. To that end, the following chapters illustrate simple, efficient algorithms for manipulating simplices and simplicial quadtrees, and discusses several techniques for producing adaptive meshes with well-shaped elements from them.

Chapter 2

Simplices

Among the possible shapes for elements of a mesh, simplices are among the easiest to represent. A simple list of its vertices serves to describe a simplex, because any subset of its vertices form a face. Other polytopal elements require topological, as well as geometrical, information. This chapter describes useful techniques for manipulating simplices and defines the sets of simplices closed under these manipulations.

2.1 Definitions

This section defines several terms from linear algebra and affine geometry that are essential to a formal understanding of simplices, the primary objects of this thesis. A basic reference for this material is the text by Fleming [Fle77]. Those already familiar with simplices, or those content with an informal understanding, may choose to skip this section.

2.1.1 Linear Algebra and Vectors

Definition A triple $(S, +, F)$, where $(S, +)$ defines a group and F is a field, defines a *vector space* when $\alpha \in F$ and $\boldsymbol{v} \in S$ implies that there is an element $\alpha\boldsymbol{v} \in S$ and

- $\alpha, \beta \in F, \boldsymbol{v} \in S \Rightarrow (\alpha + \beta)\boldsymbol{v} = (\alpha \boldsymbol{v}) + (\beta \boldsymbol{v})$,
- $\alpha \in F, \boldsymbol{v}, \boldsymbol{w} \in S \Rightarrow \alpha(\boldsymbol{v} + \boldsymbol{w}) = (\alpha \boldsymbol{v}) + (\alpha \boldsymbol{w})$,
- $\alpha, \beta \in F, \boldsymbol{v} \in S \Rightarrow \alpha(\beta \boldsymbol{v}) = (\alpha\beta)\boldsymbol{v}$, and
- $1\boldsymbol{v} = \boldsymbol{v}$.

An element of S is called a *vector*.

Definition A *linear combination* of the vectors $\{\boldsymbol{v}_j\}$, $0 \leq j < k$, is vector of the form

$$\boldsymbol{v} = \sum_{0 \leq j < k} \alpha_j \boldsymbol{v}_j \text{ with } \forall_{0 \leq j < k}(\alpha_j \in F)$$

Definition A set of vectors $\{\boldsymbol{v}_j\}$, $0 \leq j < k$ is called *linearly independent* when

$$\sum_{0 \leq j < k} \alpha_j \boldsymbol{v}_j = 0 \Longleftrightarrow \forall_{0 \leq j < k}(\alpha_j = 0)$$

Definition A maximal linearly independent set of vectors forms a *basis* for a vector space. The size of a basis for a vector space is the *dimension* of the vector space.

Definition A *linear subspace* is a set of vectors that can be expressed as a linear combination of vectors from a linearly independent set. The linearly independent set is a basis for the subspace.

2.1.2 Affine Geometry

There is a natural correspondence between vector spaces over the reals and real affine geometry. In this correspondence, a vector represents the motion of a point a certain distance in a certain direction, vector addition represents the composition of two motions, and scalar multiplication represents a scaling in the magnitude of a motion. The vector that represents the motion necessary to move a point $\boldsymbol{p}$ to a point $\boldsymbol{q}$ we denote $(\boldsymbol{q}-\boldsymbol{p})$, and the addition of a vector to a point we define so that $\boldsymbol{p}+(\boldsymbol{q}-\boldsymbol{p})=\boldsymbol{q}$.

Definition An *affine combination* of the points $\{\boldsymbol{p}_j\}$, $0 \leq j \leq k$, is a point of the form

$$\boldsymbol{p}=\sum_{0\leq j\leq k}\alpha_j\boldsymbol{p}_j \text{ with } \forall_{0\leq j\leq k}(\alpha_j\in F) \text{ and } \sum_{0\leq j\leq k}\alpha_j=1.$$

This definition is mathematically legitimate because it can be recast as the sum of a point and a linear combination of vectors.

Definition A set of points $\{p_j\}$, $0\leq j\leq k$ is called *affinely independent* when the vectors $\{p_j-p_k\}$ are linearly independent for $0\leq j<k$.

Definition An *affine subspace* or *affine hull* is the set of points that can be expressed as an affine combination of points from an affinely independent set.

2.1.3 Convexity

Definition A *convex combination* of points $\{p_j\}$, $0\leq j\leq k$, is a point of the form

$$p=\sum_{0\leq j\leq k}\alpha_j p_j \text{ with } \forall_{0\leq j\leq k}(\alpha_j\in F),$$

where $\sum_{0\leq j\leq k}\alpha_j=1$ and $\forall_{0\leq j\leq k}\alpha_j\geq 0$.

Definition A *convex* set is a set closed under the convex combination of two of its points. That is, the line segment connecting two points of the set lies entirely within the set.

Definition The *convex hull* of a set is the closure of the set under convex combination.

The intuition behind planar convex hulls is simple; the convex hull is just that set of points enclosed by a rubber band that snaps tightly to the points at the extremes of the set. In three dimensions, a similar analogy involving a rubber sheet applies.

Theorem 1 *If $\boldsymbol{q}$ lies in the convex hull of the point set $\{\boldsymbol{p}_j\}$, $0 \leq j \leq m$, then $\boldsymbol{q}$ can be expressed as a convex combination of the points in the set.*

Definition The convex hull of points from a finite set V' is a *convex polytope*. The *vertices* of the polytope are the elements of the smallest set $V \in V'$ for which $hull(V) = hull(V')$. The convex hull of $F \subset V$ is a *face* of the polytope if none of its points can be expressed as convex combinations involving points outside F.

Definition The convex hull of points from an affinely independent set V is a *simplex*. The points of V are the vertices of the simplex. The convex hull of any subset of V is a face of the simplex. Vertices of a simplex are 0-dimensional faces, *edges* are 1-dimensional faces, and *facets* are faces of the simplex with dimension one less than the dimension of the simplex.

In the affine space of daily experience, a simplex is a point, a line segment, a triangle, or a tetrahedron. In general, the dimension of a simplex is one less that the number of its vertices.

2.2 Recursive simplicial decompositions

A principal component of most of the algorithms and ideas that follow is *recursive simplicial decomposition*, the division of a simplex into similar, but smaller simplices. A particularly important type of decomposition is a triangulation.

Definition A *conformal partition* of a set P is a set of elements $\mathcal{E}$ for which $\bigcup_{E \in \mathcal{E}} E = P$ and, for any elements E and $E' \in \mathcal{E}$, $E \cap E'$ is a face of E. A conformal partition is also called a *mesh*. A *triangulation* is a mesh of simplicial elements.

Before considering the triangulation of the simplex, however, it is instructive to consider the triangulation of a more familiar figure, the box.

2.2.1 Kuhn triangulations

A *box* is the general term that encompasses the point, the line segment, the square, the cube, and the tesseract. Boxes are also called hypercubes or d-cubes. The *unit box* of a Cartesian coordinate system is the set of points satisfying $0 \leq x_i \leq 1$ for all coordinate axes x_i.

Triangulations of boxes are much studied in the literature on homotopy methods [Lee81,Sal82]. A particularly important triangulation of the box is called the *Coxeter-Freudenthal-Kuhn triangulation* [Cox34,Fre42,Kuh60], or simply the Kuhn triangulation. To build a Kuhn triangulation of the unit d-box, select a pair of opposite vertices, such as $(0,\ldots,0)$ and $(1,\ldots,1)$, that define a principal diagonal for the box, and consider the set of shortest paths from vertex to the other. These paths are of length d. Each edge of the path crosses the box in a different dimension, and each of the $d!$ permutations of the d dimensions corresponds to a path on the boundary of the box. Each path contains $d+1$ vertices, and the convex hull of the path is simplex, of a type that Coxeter calls an *R-simplex* [Cox34], that is exactly the region with $0 \leq x_{p(1)} \leq \ldots \leq x_{p(d)} \leq 1$ for a permutation p. The set of these $d!$ R-simplices is the Kuhn triangulation of the box. Figure 2.1 illustrates the Kuhn triangulation of a cube.

It is well known that a d-dimensional space can be filled with an infinite grid of d-boxes [Cox73]. This grid and the Kuhn triangulation of the box lead to two different methods for triangulating a d-dimensional space. Using Todd's terminology [Tod76], the triangulation that fills space with the R-simplices by translating the unit box to each position in the infinite grid is *Freudenthal's triangulation*, denoted K_1. *Tucker's triangulation* J_1 fills space with the R-simplices that arise by reflecting the unit box about each of its facets to fill neighboring boxes, and then repeating this reflection process to fill space. Figure 2.2 illustrates the K_1 and J_1 triangulations of the plane.

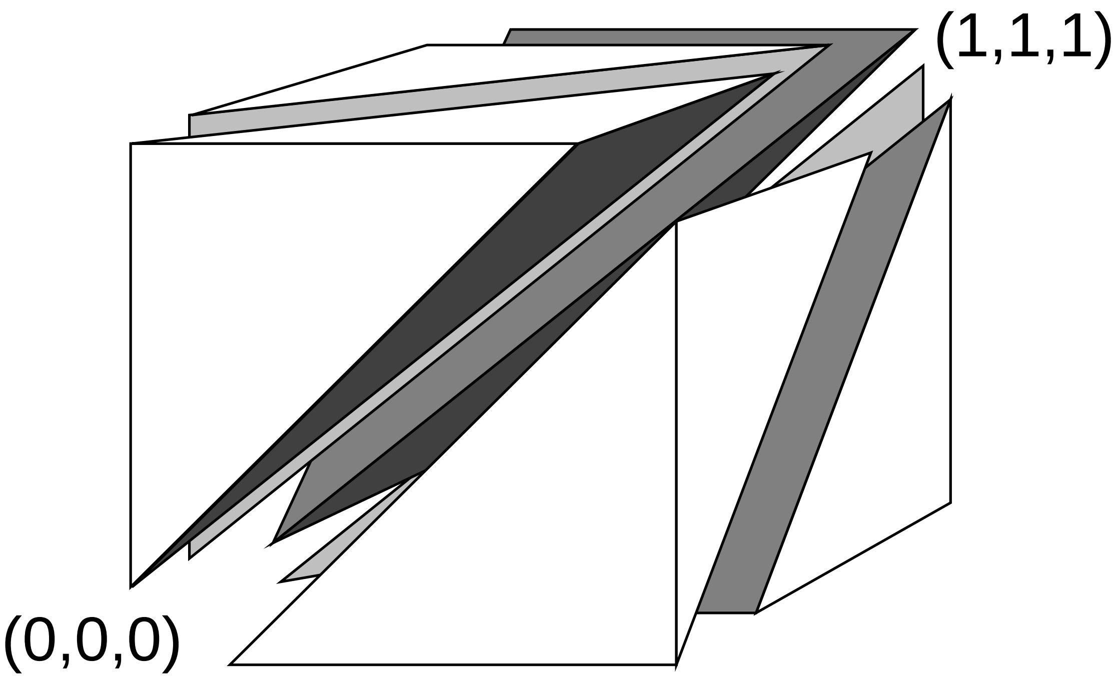


Figure 2.1: Kuhn triangulation of a cube

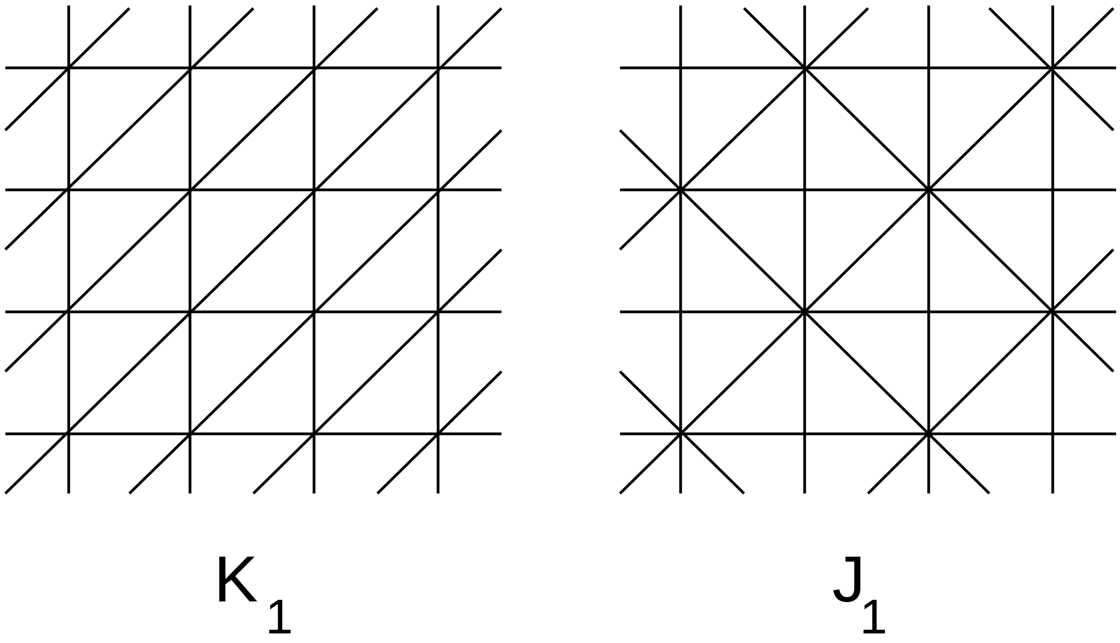


Figure 2.2: Two Kuhn triangulations of the plane

2.2.2 Recursive decomposition of boxes and simplices

The unit d-box is divided into 2^d identical boxes by the planes $x_i = 1/2$ for $1 \leq i \leq d$. This recursive decomposition of the unit box is the basis for the spatial data structure called the *region quadtree* [Sam90]. There are two ways to build similar decompositions for simplices, by considering the interplay between the triangulation of a mesh of boxes and the decomposition of a box into subboxes.

Let $\boldsymbol{v}_j$, $0 \leq j \leq d$, denote the point whose first j coordinates are 1 and whose remaining $d-j$ coordinates are 0. Define the convex hull of $\{\boldsymbol{v}_0, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_d\}$ as the R-simplex in standard position. To divide that parent simplex into 2^d smaller R-simplices, one can apply a standard recursive procedure called *Whitney's policy* [Whi57] in the literature of fixed-point and homotopy methods. In the first step, a hyperplane cuts through the midpoint of edge $\overline{\boldsymbol{v}_0\boldsymbol{v}_d}$ and through all the other vertices to divide the parent simplex into two simplices. In the second step, a hyperplane cuts one simplex through the midpoint of $\overline{\boldsymbol{v}_0\boldsymbol{v}_{d-1}}$ and all its other vertices, while another cuts the other simplex through the midpoint of $\overline{\boldsymbol{v}_1\boldsymbol{v}_d}$ and all its other vertices. Proceed at step k by cutting edge $\overline{\boldsymbol{v}_j\boldsymbol{v}_{j+(d-k)+1}}$, $0 \leq j < k$, and each step cuts one edge per simplex and doubles the number of simplices. The result of the dth step is the *recursive* decomposition of the simplex. Figure 2.3 illustrates this process for a tetrahedron.

The diagram of figure 2.4 provides a nonstandard, but concise, way of viewing the recursive decomposition of a tetrahedron. To build the diagram that corresponds to a d-simplex, list $d+1$ pairs (i,i), $0 \leq i \leq d$, and above any two adjacent pairs (j_1, k_1) and (j_2, k_2) list the pair (j_1, k_2). The process ends with a single pair $(0,d)$ at the top of a triangular arrangement of pairs. In the diagram, the pair (j_0, j_1) represents the midpoint of edge $\overline{\boldsymbol{v}_{j_0}\boldsymbol{v}_{j_1}}$ of the parent tetrahedron, and (j,j) represents vertex $\boldsymbol{v}_j$. Each path from the bottom of the diagram to the top represents a set of vertices that define a simplex of the recursive subdivision. To generate just the kth simplex of the decomposition ($0 \leq k < 2^d$), start at the

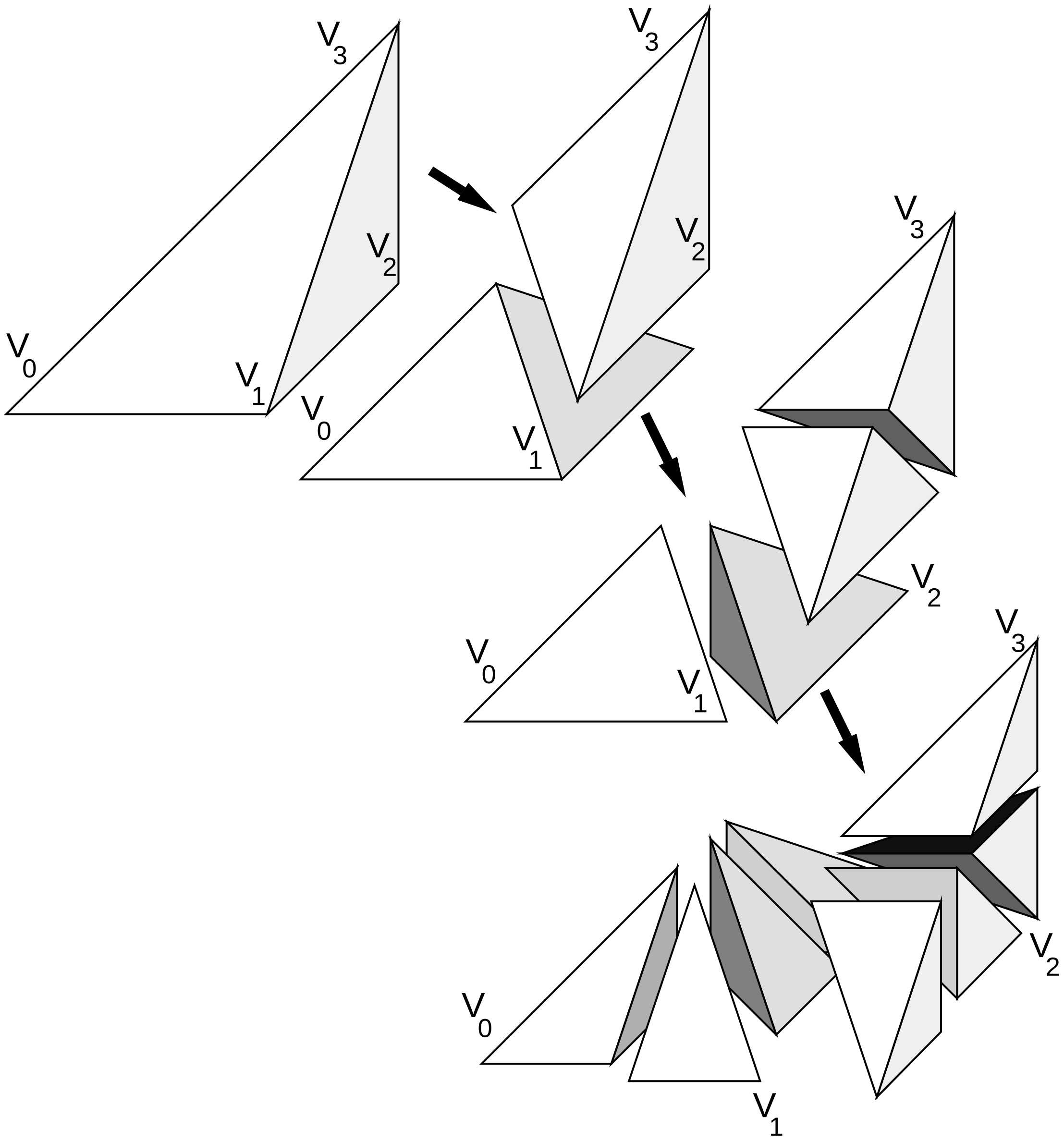


Figure 2.3: Phases of recursive subdivision

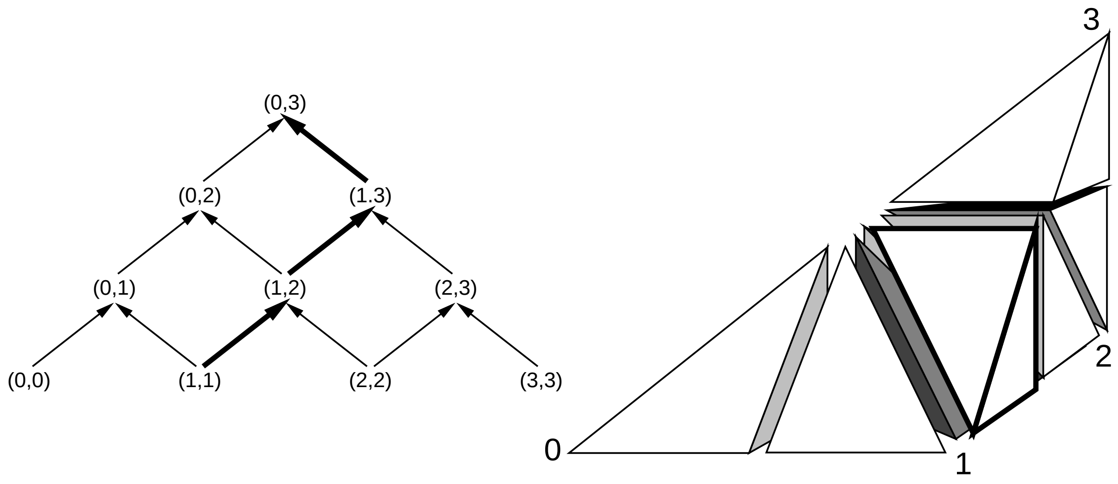


Figure 2.4: Recursive subdivision

node labeled $(b(k), b(k))$, where $b(k)$ denotes the number of 1 bits in the binary representation of k. Starting with the least significant bit, traverse a path from the bottom of the diagram to the top, turning right with every 0 bit and left with every 1 bit. The vertices encountered in that traversal are the vertices of the kth subsimplex. The diagram highlights the traversal for the 4th subsimplex of the recursive subdivision, and that subsimplex has a bold outline in the accompanying illustration. The algorithm appears in figure 2.5.

Theorem 2 *The simplices that the diagram of figure 2.4 represents form a triangulation of the larger parent simplex.*

Proof For concreteness, consider the simplex with vertices $2\boldsymbol{v}_0, \ldots, 2\boldsymbol{v}_d$. A path in the diagram from $(b(k), b(k))$ to $(0, d)$ represents a path in d-space from $2\boldsymbol{v}_{b(k)}$ to $\boldsymbol{v}_d$ that lies on the boundary of the box with lexicographically least vertex $\boldsymbol{v}_{b(k)}$. The set of such paths defines a subset of a Kuhn triangulation of the box about the diagonal from $2\boldsymbol{v}_{b(k)}$ to $\boldsymbol{v}_d$. Thus, within each unit-sized box, simplices form a triangulation.

Since the interiors of the boxes do not overlap, it follows that two simplices in different unit-sized boxes generated by the diagram have disjoint interiors. As

```
Simplex RecSimplex::subElement(int k) const
// kth recursive subsimplex of this simplex
{
   RecSimplex child(nVtxs());
   int id[2] = {-bitCount(k), bitCount(k)};
   for (int j = 0; j ≤ nVtxs(); ++j)
      child.vtx(j) = midPoint(vtx(-id[0]), vtx(id[1])),
      id[odd(k)? 1: 0]++,
      k /= 2;
   return child;
}
```

Figure 2.5: Algorithm for recursive subdivision

shown by Dahmen and Micchelli [DM82], the Kuhn triangulation of any box induces Kuhn triangulations of its faces. Thus, where two different boxes share a common face, Kuhn triangulations of each box subdivide that face. By the construction, the first and last vertices of the triangulation of the face of a unit-sized box are the vertices corresponding to the pairs (i, j) with least and greatest value of $j - i$. Thus, the principal diagonals of the Kuhn triangulations of the face are the same for each box and the triangulations coincide on the face common to the two boxes. □

The recursive decomposition of a simplex is related to the J_1 triangulation of space. Consider the 2^d boxes containing the point $\boldsymbol{v}_d$ in a box filling of space, and triangulate each according to the J_1 triangulation. Consider at the same time the large box made up of those 2^d smaller boxes, and consider an element of the Kuhn triangulation of the larger box. Whichever Kuhn simplex is selected, its volume coincides with 2^d smaller simplices from the J_1 triangulation, and the smaller simplices are arranged in the larger one as they would be in the recursive decomposition of the larger simplex.

Recursive subdivision is not a new technique. A common heuristic for simplex refinement is bisection of the longest edge of the simplex [PPW87]; as described above, recursive subdivision in two and three dimensions is just the application of the heuristic several times to an R-simplex. A binary tree describes a hierarchy of simplices built by recursive subdivision, so many familiar algorithmic techniques can be applied to the product of recursive subdivision.

2.2.3 Symmetric subdivision of simplices

Any triangle can be divided into four similar triangles, in which the subtriangle at each corner has the same shape and orientation as the parent, and the remaining triangle has the same shape but a different orientation. That division of the triangle is not obtained by recursive subdivision. Indeed, recursive subdivision of an arbitrary triangle with vertices arbitrarily labeled $\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2$ generally yields dissimilar subtriangles. In this section, we characterize a subdivision property that we seek to preserve and produce a subdivision method that preserves it.

Definition The subdivision of a d-dimensional element into 2^d smaller elements has the *corner-matching* property if each vertex of the parent element is incident to exactly one subelement, and the subelements are congruent and similar to the parent. The congruence of the definition is an *unoriented* congruence, in which two elements are considered congruent if rigid-body transformations and reflections can bring their faces into one-to-one correspondence. A stricter *oriented* congruence that forbade reflections would not consider all the R-simplices in the Kuhn triangulation of a box congruent. With this definition, recursive simplicial subdivision does not have the corner-matching property, but box subdivision does.

A subdivision technique that has the corner-matching property for general simplices is the *symmetric* subdivision method. The symmetric subdivision of an R-simplex can best be understood by considering the K_1 triangulation of space. Consider again the 2^d boxes containing the point $\boldsymbol{v}_d$ in a box filling of space, and triangulate the set of boxes according to the K_1 triangulation. Consider also an element of the Kuhn triangulation of the large box made up of all 2^d of the small boxes. This element overlaps exactly 2^d smaller R-simplices, and that packing of the large simplex defines the symmetric subdivision of the simplex. Figure 2.6 illustrates this construction.

Symmetric subdivision can also be understood by the diagram in the left half of figure 2.7. In general, the diagram for symmetric subdivision is obtained by reorienting the directions of the edges of the recursive subdivision diagram so that they are directed from left to right rather than from bottom to top. To generate the kth subsimplex of the symmetric subdivision ($0 \leq k < 2^d$), start at the node labeled $(0, b(k))$, and, starting with the least significant bit, traverse a path from the left side of the diagram to the right, turning up with every 0 bit and down with every 1 bit. The vertices encountered in that traversal are the vertices of the kth subsimplex. The figure highlights the traversal for the 4th subsimplex of the symmetric subdivision. The right half of the figure illustrates the symmetric subdivision of a tetrahedron, with the 4th subsimplex outlined in bold. Figure 2.8 presents a program to that generate the kth subsimplex of a simplex.

Theorem 3 *The simplices that the diagram of figure 2.7 represents form a triangulation of the larger parent simplex, and the triangulation has the corner-matching property.*

Proof Consider again the simplex with vertices $2\boldsymbol{v}_0, \ldots, 2\boldsymbol{v}_d$. A path in the diagram from $(0, b(k))$ to $(b(k), d)$ represents a path in d-space from $\boldsymbol{v}_{b(k)}$ to $\boldsymbol{v}_{b(k)} + \boldsymbol{v}_d$ that lies on the boundary of the box with lexicographically least vertex $\boldsymbol{v}_{b(k)}$.

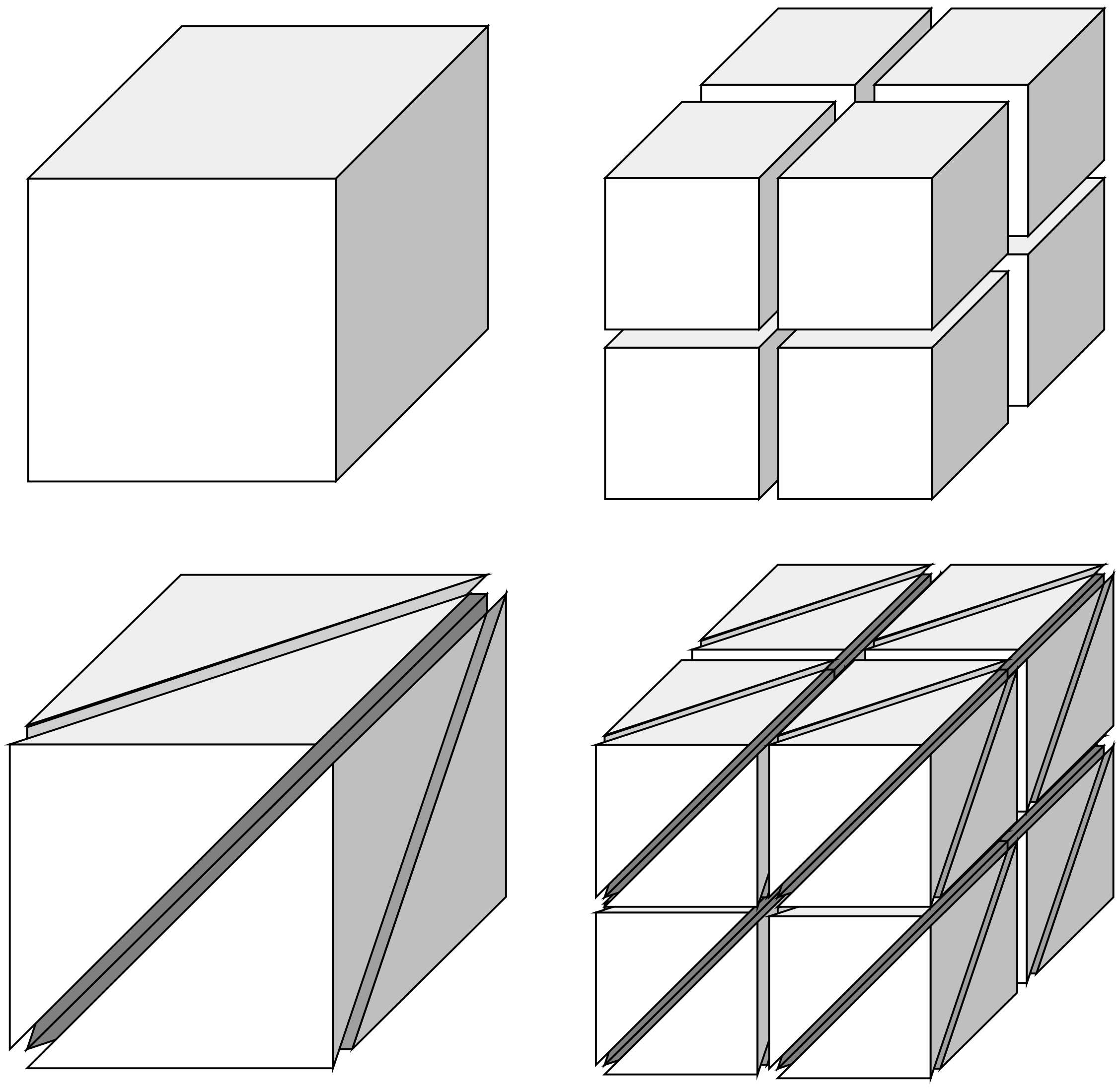


Figure 2.6: Symmetric subdivision from box subdivision

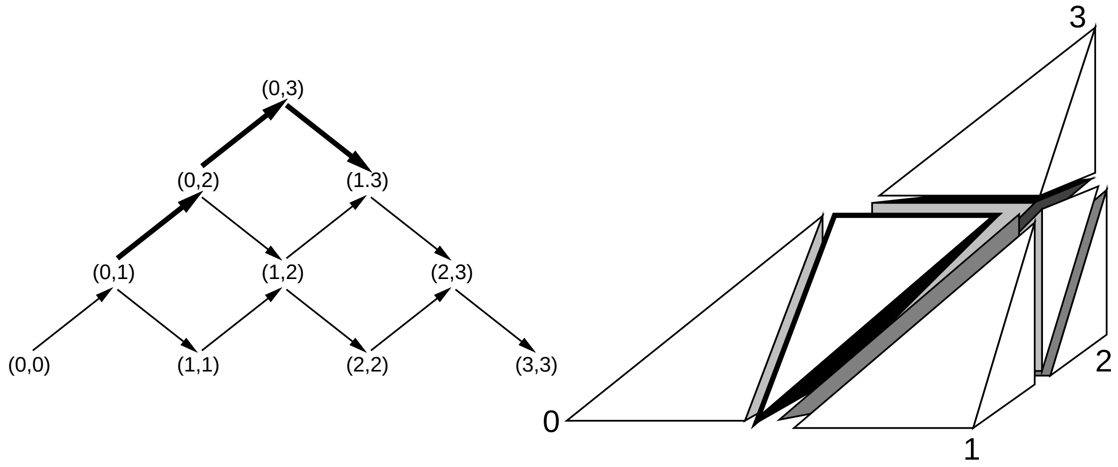


Figure 2.7: Symmetric subdivision

```
Simplex SymSimplex::subElement(int k) const
// kth symmetric subsimplex of this simplex
{
   SymSimplex child(nVtxs());
   int id[2] = {0, bitCount(k)};
   for (int j = 0; j ≤ nVtxs(); ++j)
      child.vtx(j) = midPoint(vtx(id[0]), vtx(id[1])),
      id[odd(k)? 0: 1]++,
      k /= 2;
   return child;
}
```

Figure 2.8: Algorithm for symmetric subdivision

The proof that the set of simplices so generated forms a triangulation is much like the proof of theorem 2.

The jth vertex of the parent simplex is a vertex of only the (2^j-1)th subsimplex of the symmetric subdivision. For any index k with more than j one bits in its binary representation, the kth subsimplex does not contain $\boldsymbol{v}_j$ because there is no path from $(0, b(k))$ to (j, j) in the directed diagram. For an index k with fewer than j one bits, there is no path from (j, j) to $(b(k), d)$ in the directed diagram. The only path from $(j, 0)$ to (j, j) corresponds to an index in which the first j bits of the binary representation are zero, so only the $(2^j - 1)$th path passes through (j, j). Thus, the symmetric decomposition satisfies the corner-matching property. $\square$

2.2.4 Ordered simplices and decomposition

Each of these techniques for subdividing R-simplices really subdivides *ordered* R-simplices. The numbering of the vertices affects the way in which the simplex is divided. For R-simplices, the most natural order is the one in which each pair of consecutive vertices lie on an edge of a box.

The ordering of the vertices of a simplex naturally induces an ordering of the vertices of any of its faces, in which $\boldsymbol{v}_{j_0} < \boldsymbol{v}_{j_1}$ on the face if and only if $\boldsymbol{v}_{j_0} < \boldsymbol{v}_{j_1}$ in the ordering of the vertices of the simplex. A pair of simplices sharing a common face are *consistently ordered* if and only if the orders of the vertices on that common face induced by each of the simplices are the same. If there is a partial order on the set of all vertices that induces a total order on vertices of each element of a set of simplices, then all pairs of simplices in that set are consistently ordered. The recursive and symmetric subdivision algorithms of the previous section generate partial orders of subsimplex vertices, so the recursive and symmetric subdivision schemes produce consistently ordered refined triangulations from consistently ordered initial triangulations.

Another useful order-related property of each subdivision scheme concerns the relations of the subsimplices to certain hyperplanes. For any $j \in \{1, \ldots, d\}$, there is a hyperplane that separates the first j vertices of the simplex from the last $d+1-j$, and contains the midpoint of each edge $\overline{\boldsymbol{v}_{j_0}\boldsymbol{v}_{j_1}}$, where $0 \leq j_0 < j \leq j_1 \leq d$. For the R-simplex in standard position, this is just the hyperplane $x_j = 1/2$. In either subdivision scheme, no subsimplex has vertices on opposite sides of one of these hyperplanes. Thus, to subdivide a simplex so that a particular hyperplane is a separating hyperplane, an ordering can be assigned to its vertices so that the first several and last several vertices lie on opposite sides of the plane.

Symmetric subdivision also induces a total order on vertices that appear along any line that contains an edge after several iterations of subdivision. For example, for every ordered edge $\overline{\boldsymbol{v}\boldsymbol{v}''}$ in an R-simplex in standard position, $\boldsymbol{e}^T\boldsymbol{v} < \boldsymbol{e}^T\boldsymbol{v}''$, where $\boldsymbol{e}$ is the vector of ones. Symmetric subdivision produces the subsimplex containing $\boldsymbol{v}$ before the subsimplex containing $\boldsymbol{v}''$, so in visiting each of the subsimplices in order, one always visits the edge $\overline{\boldsymbol{v}\boldsymbol{v}'}$ before the edge $\overline{\boldsymbol{v}'\boldsymbol{v}''}$, where $\boldsymbol{v}'$ denotes the midpoint of $\overline{\boldsymbol{v}\boldsymbol{v}''}$. Every edge of every subsimplex has the same orientation as parallel edges in the parent simplex.

2.3 A special class of simplices

2.3.1 Congruence of subsimplices

The two subdivision schemes for R-simplices produce smaller R-simplices. The same techniques can be applied to an arbitrary ordered simplex to produce a consistent triangulation of the simplex. Most of the subsimplices are not, in general, similar to the parent simplex. Recursive subdivision of a d-simplex produces 2 congruent subsimplices similar to their parent, and, by the corner-matching property, symmetric subdivision produces $d+1$ congruent subsimplices similar to their parent. This section considers the properties that a simplex must have so that all its recursive or symmetric subsimplices are congruent.

Theorem 4 *For $d > 2$, if any two subsimplices produced in the recursive subdivision of a d-simplex are congruent, then the parent simplex is an R-simplex.*

Proof Consider an ordered d-simplex S that can be recursively subdivided into congruent simplices. A linear transformation T takes S to the R-simplex in standard position. Subdivide the R-simplex several times, enough so that the descendants of the simplex pack the volume around some point in the interior of the transformed S, forming a d-box. Invert the transformation. The box transforms to an arbitrary d-parallelepiped, made up of 2^d smaller parallelepipeds, and each of those is Kuhn triangulated with descendants of S. For the descendants to be congruent, the distance between every pair of opposite vertices of the parallelepiped must be same. Thus, the parallelepiped must be rectangular.

The Kuhn triangulation of a rectangular parallelepiped does not necessarily produce congruent simplices. In three dimensions, for example, the Kuhn triangulation of a $1 \times 2 \times 3$ parallelepiped produces three different shapes of simplices; in two, the only path connecting all the vertices and with all edges perpendicular has a length 1 edge as its middle edge. In two more, the length of the middle edge is 2 and in two others it is 3. In general, for a rectangular d-parallelepiped, there are $d!$ different shapes among the simplices of a Kuhn triangulation, or $d!/2$ if reflections are permitted, so that an ordering and its reverse can be identified.

Thus, only rectangles among all the rectangular parallelepipeds can divide into congruent simplices. In dimensions higher than two, the only way for all the simplices to be congruent is for all the edge lengths of the rectangular parallelepiped to be the same. A rectangular parallelepiped satisfying that requirement is a box, and the simplices from which it is made must be R-simplices. □

Symmetric subdivision provides a bit more flexibility than recursive subdivision, as the following theorem shows.

Theorem 5 *For $d > 2$, a one degree of freedom family contains all d-simplices of fixed volume for which symmetric subdivision produces congruent subsimplices.*

Proof Consider an ordered d-simplex S for which symmetric subdivision produces congruent subsimplices. As in the proof of theorem 4, a transformation takes S to an R-simplex in standard position. Subdivide symmetrically until descendants of the transformed S completely surround a point of its interior and map that d-box back to the original domain.

A Kuhn triangulation divides each of the parallelepipeds surrounding the point into d-simplices similar to S, but all the main diagonals of the boxes are parallel, not adjacent as in the recursive case. For the d-simplices of the Kuhn triangulation of a parallelepiped to be congruent, it is necessary for all edges in any path around the outside of the parallelepiped from one end of the main diagonal to the other to have the same length. Moreover, to make the angles of the turns in such a path all the same, it is necessary for all pairs of edges radiating from the initial vertex of the Kuhn triangulation to have the same angle θ between them, so that the angle at every turn is $(\pi - \theta)$. Thus, the set of vertices of the box adjacent to the initial vertex must form a regular $(d-1)$-simplex. The only remaining degree of freedom is in the choice of the angle θ. $\square$

These constraints describe the set of *rhombic* parallelepipeds. Rotations that hold the ends of the main diagonal fixed and interchange other vertices, and reflections in the direction of the main diagonal, can bring any simplex in the Kuhn triangulation of a rhombic parallelepiped into line with any other. In each dimension, one choice of the angle θ produces symmetrically decomposable simplices with a particularly useful shape.

2.3.2 Aspect ratio analysis of R-simplices

For aesthetic reasons in computer graphics, and for reasons of numerical accuracy in engineering analysis problems, some simplex shapes are undesirable. Several measures of the shape of a simplex have appeared. Parthasarathy has surveyed several of the most popular measures for tetrahedra, judging them on their sensitivity to different types of degeneracies and the computational efficiency of calculating them [Par91]. Dobkin et al. define two interesting probabilistic measures of simplex shape based on the expected behavior of randomly distributed points and lines [DLTW90], and refers to Todd as a source of still more useful shape metrics [Tod76]. Several conflicting definitions of "aspect ratio" have appeared, each corresponding to one of these metrics. Most of these definitions have taken a regular simplex to have minimum aspect ratio and a degenerate simplex to have infinite aspect ratio.

The *aspect ratio* of a simplex as defined by Cavendish et al. [CFF85] is the ratio of the radius of a circumscribing ball to the radius of an inscribed ball, divided by the dimension. The aspect ratio of a regular simplex is one in all dimensions, and grows larger as the simplex becomes less regular. Parthasarathy considers this definition of aspect ratio acceptable, but computationally inefficient, and Dobkin et al. report that it appears to correlate with their other metrics.

Theorem 6 *The aspect ratio of an R-simplex in d dimensions is given by*

$$\frac{d-1+\sqrt{2}}{\sqrt{2d}}.$$

In particular the R-tetrahedron has aspect ratio $\sqrt{1/3}+\sqrt{2/3}\approx 1.394$.

In the following, we assume that the R-simplex S_d is the convex hull of the point sequence $\{\boldsymbol{v}_j\}(0 \leq j \leq d)$, where $\boldsymbol{v}_j$ has -1 as its first j coordinates and 1 as the rest.

Proof The circumcenter of S_d is the origin and its circumradius is $r_{circum}=\sqrt{d}$. The incenter $(c_1,\ldots,c_d)$ lies equidistant from each of the hyperplanes that bound the R-simplex, and that distance is the inradius r_{in} of the R-simplex. Formally, the constraints are

$$\begin{aligned} r_{in} &= 1-c_1 \text{ (the hyperplane } x_1=1), \\ &= (c_i-c_{i+1})/\sqrt{2}, (1\leq i<d) \text{ (the hyperplane } x_i=x_{i+1}), \text{ and} \\ &= c_d+1 \text{ (the hyperplane } x_d=-1) \end{aligned}$$

which suggests that the c_i form an arithmetic sequence symmetric about zero. The substitution $c_i = D((d+1)/2 - i)$ reduces the constraints to two,

$$\begin{aligned} r_{in} &= 1 - D(d-1)/2 \\ r_{in} &= D/\sqrt{2} \end{aligned}$$

and leads to the conclusion that $r_{in} = \frac{\sqrt{2}}{d-1+\sqrt{2}}$. Dividing r_{circum} by dr_{in} yields $\frac{d-1+\sqrt{2}}{\sqrt{2d}}$, as stated. $\square$

2.3.3 P-simplices

The degree of freedom present in the family of symmetrically subdividable simplices permits the choice of simplices with better aspect ratio than R-simplices. To express this degree of freedom, define

$$M_d(\alpha) = I_d + (1/\alpha - 1)(\boldsymbol{e}_d \boldsymbol{e}_d^T)/d,$$

where I_d denotes the $d \times d$ identity matrix and $\boldsymbol{e}_d$ is the d-vector of ones. For any d-vector $\boldsymbol{v}$, $M_d(\alpha)v$ is a translation of v parallel to $\boldsymbol{e}_d$, and $M_d(\alpha)\boldsymbol{e}_d = \boldsymbol{e}_d/\alpha$. Thus, $M_d(\alpha)$ is a transformation that compresses by a factor α in the direction $\boldsymbol{e}_d$. Compressing the box with principal diagonal from $\boldsymbol{e}_d$ to $-\boldsymbol{e}_d$ produces a rhombic parallelepiped with principal diagonal from $\boldsymbol{e}_d/\alpha$ to $-\boldsymbol{e}_d/\alpha$, and the elements of the Kuhn triangulation of the parallelepiped along its principal diagonal are symmetrically subdividable. This section reveals the choice of α that produces the best aspect ratio among all symmetrically subdividable simplices.

As in the consideration of the aspect ratios of R-simplices, begin with the convex hull of the point sequence $\{\boldsymbol{v}_j\}(0 \le j \le d)$, where $\boldsymbol{v}_j$ has -1 as its first j coordinates and 1 as the rest. Under the transformation $M_d(\alpha)$, the constraints that determine the incenter change only slightly. All the hyperplanes that bound the simplex remain unchanged, except that $1 - x_1 = 0$ becomes

$$1 - x_1 + \frac{1-\alpha}{d}\boldsymbol{e}_d^T \boldsymbol{x} = 0$$

and $x_d + 1 = 0$ becomes

$$x_d + 1 - \frac{1-\alpha}{d}\boldsymbol{e}_d^T\boldsymbol{x} = 0.$$

As a result, the constraints that determine the incenter and inradius are

$$\begin{aligned} r_{in} &= (1 - c_1 + \frac{1-\alpha}{d}\boldsymbol{e}_d^T\boldsymbol{c})\frac{d}{\sqrt{d^2-d+\alpha^2 d}}, \\ &= (c_i - c_{i+1})/\sqrt{2}, (1 \le i < d), \text{ and} \\ &= (c_d + 1 - \frac{1-\alpha}{d}\boldsymbol{e}_d^T\boldsymbol{c})\frac{d}{\sqrt{d^2-d+\alpha^2 d}}, \end{aligned}$$

and again the coordinates c_i form an arithmetic sequence symmetric about zero. The substitution $c_i = D((d+1)/2 - i)$ gives

$$\begin{aligned} r_{in} &= (1 + D(d-1)/2)\frac{d}{\sqrt{d^2-d+\alpha^2 d}}, \\ r_{in} &= D/\sqrt{2}, \end{aligned}$$

which has the solution $r = \frac{d}{\sqrt{d^2-d+\alpha^2 d} - \sqrt{1/2}d(d-1)}$.

Having determined the inradius for the α-compressed R-simplex, it remains to calculate the circumradius. The vertices $\boldsymbol{w}_j = M_d(\alpha)\boldsymbol{v}_j$ satisfy

$$\boldsymbol{w}_{ji} = (1 - 1/\alpha)(1 - 2j/d) + (\text{if } (i \le j) \text{ then } 1 \text{ else } -1),$$

The circumcenter $\boldsymbol{c}$ is given by $c_i = (1 - \frac{1}{\alpha^2})(1 - \frac{2i-1}{d})$ and circumradius by $\sqrt{d + (d^2-1)(\alpha^2-1)^2/3\alpha^2 d}/\alpha$.

The application of calculus and algebra to these formulas reveals that the choice $\alpha = \sqrt{1+d}$ minimizes the aspect ratio of the simplex, producing the *P-simplices* [Cox34] and yielding aspect ratio $\sqrt{(d+1)(d+2)/6d}$. Although aspect ratios of P-simplices increase as the dimension grows, the aspect ratios for the first few dimensions are good.

Smith calls P-tetrahedra *BCC tetrahedra* [Smi88], and van der Laan and Talman call a packing of 3-space with P-tetrahedra the α^*-triangulation [vdLT80]. It is the optimal packing of space with tetrahedra under several measures, including, according to Smith, aspect ratio; the aspect ratio of a P-tetrahedron is $\sqrt{10}/3 \approx$

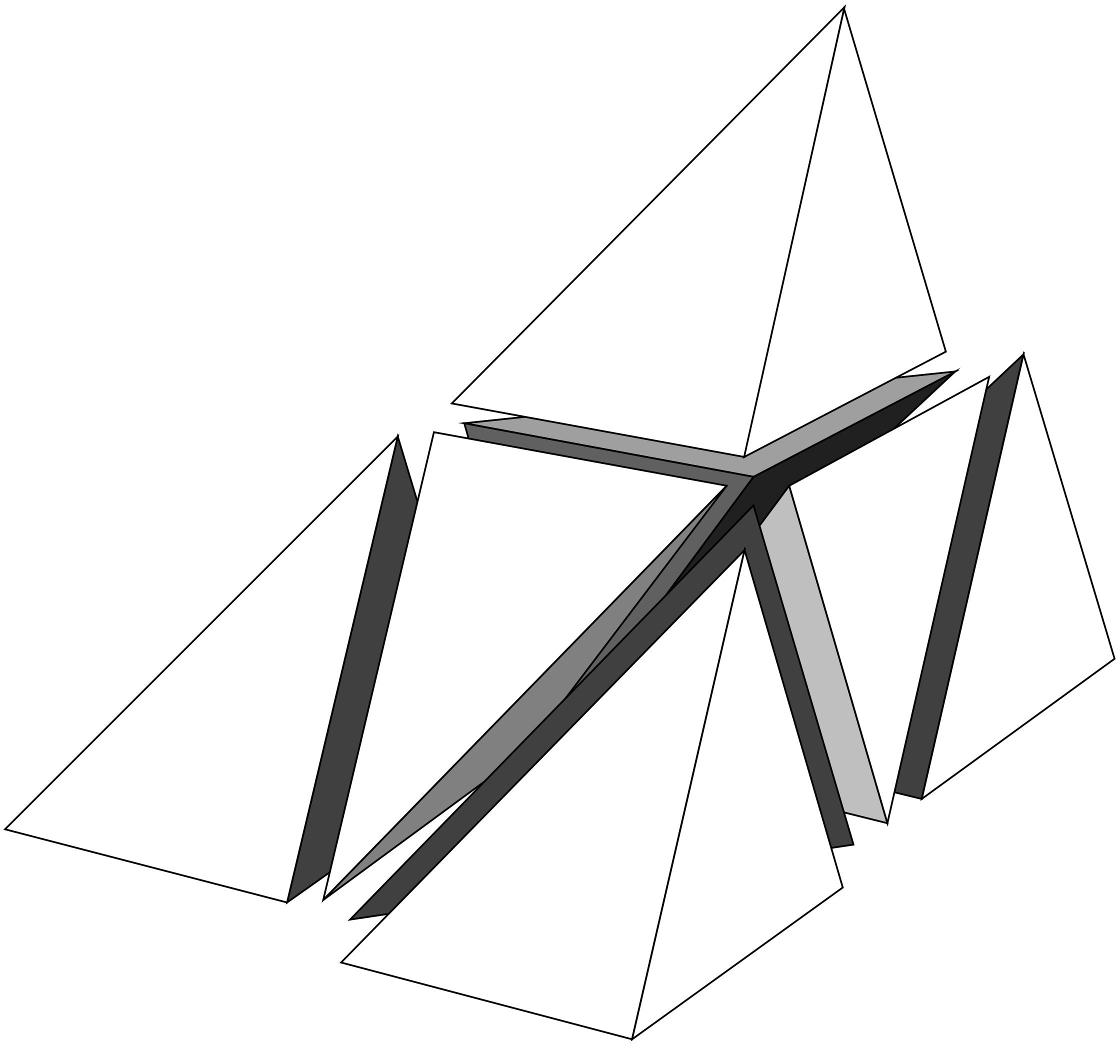


Figure 2.9: A P-tetrahedron

1.054 (see figure 2.9). To understand the α^*-triangulation in three dimensions, consider first a cubic grid. Pick two adjacent cubes, and two adjacent vertices on their common face. The convex hull of those two vertices and the cube centers is a P-tetrahedron. Each edge of the cubic grid is thus surrounded by four P-tetrahedra, and this collection of tetrahedra fill space. Because the vertices of the α^*-triangulation can be made to lie on a subset of an integer lattice in three dimensions, computing with the triangulation is particularly convenient.

The next chapter uses symmetric subdivision to build hierarchical data structures and adaptive triangulations. Although any initial simplex can be subdivided by the rules for symmetric subdivision, applications that can work with R-simplices will often produce more pleasing results.

Chapter 3

Adaptive triangulations

The results of the previous chapter lead immediately to methods for generating uniform simplicial triangulations. Beginning with the consistently ordered triangulation of a region, subdividing once each simplex of the triangulation produces a more refined, consistently ordered triangulation of the region. Such uniform triangulations have value in some computer applications, but more often adaptive triangulations are necessary. This chapter develops techniques that use simplicial subdivision to build adaptive triangulations. It also presents new results relating to the size of adaptive triangulations in two dimensions.

3.1 Quadtrees

3.1.1 Building an adaptive quadtree

A *quadtree* is a hierarchical spatial data structure. Originally, the term applied to data structures that represent square regions in the plane. A node in a quadtree represents a square region, and if it has four children, then they represent the four quadrants of the square region. A node with no children is homogeneous. The term has now come to encompass many different structures, and the original quadtree is now called a *two-dimensional region quadtree* [Sam90].

```
void Quadtree::refine
(const Element& E, // this single-node quadtree represents element E.
 bool (*accurate)(const Element&)) // refine to make all leaf nodes "accurate"
{
  if (accurate(E)) return;
  createChildren();
  for (int k = 0; k < nChildren(); ++k)
     child[k].refine(E.subElement(k), accurate);
}
```

Figure 3.1: Algorithm for adaptive refinement

Definition A *d-dimensional simplicial region quadtree* is a hierarchical data structure that represents an ordered d-simplex. A node in the quadtree represents an ordered d-simplex, and if it has 2^d children, then they represent the 2^d subsimplices of the d-simplex. Unless otherwise specified, the symmetric subdivision method produces the subsimplices.

It is possible to build simplicial quadtrees using either recursive or symmetric subdivision, but the results that appear below apply to symmetric subdivision. Where the ordering of the children of a node in a quadtree matters, the orderings described in the previous chapter are used.

The generic quadtree construction algorithm is presented in figure 3.1. The idea is simple. Given a subdividable element and an accuracy predicate on those elements, apply the predicate to the element. If the element satisfies the predicate, then the quadtree for that element consists of a single node. Otherwise, the quadtree for that element is a node with several children that are the quadtrees for its subelements. The specific meaning of the predicate is determined by the application. In general, the application seeks a simple approximation of a complex object, and the predicate checks that a simple object accurately models the part of the complex object defined over the element.

The example running through this chapter and the next is from an application that seeks piecewise linear approximations to plane curves. In that application, the elements are equilateral triangles and the accuracy predicate measures the nearness of the the curve to a line segment over the element. For a curve defined over a single initial element, figure 3.2 illustrates refinement of the initial triangle, and figure 3.3 shows the resulting data structure and final refinement. Each figure uses shading to distinguish opposite sides of the curve. The two figures represent the same idea in different ways; in the following, "quadtree" describes illustrations of either kind.

For the quadtree construction algorithm to work, the accuracy predicate must be *satisfiable* for the initial element. An accuracy predicate is satisfiable for an element if the element satisfies the accuracy predicate or if the predicate is satisfiable for each subelement of the element. If the accuracy predicate is not satisfiable for the initial element, the quadtree construction algorithm fails to finish, but instead extends some path in the tree without bound. For the algorithms of the next section to work, the predicate must also be *hereditary*. An accuracy predicate is hereditary if any element contained in an accurate element is also accurate.

Given an element and a predicate satisfiable for that element, the quadtree construction algorithm produces the smallest quadtree that represents the element and has leaves that each satisfy the accuracy predicate. The set of leaf elements is unlikely to represent a triangulation of the original element, however, and in many applications a triangulation is necessary. The rest of the chapter describes techniques for making a simplicial quadtree into a triangulation in which each triangle satisfies the accuracy predicate.

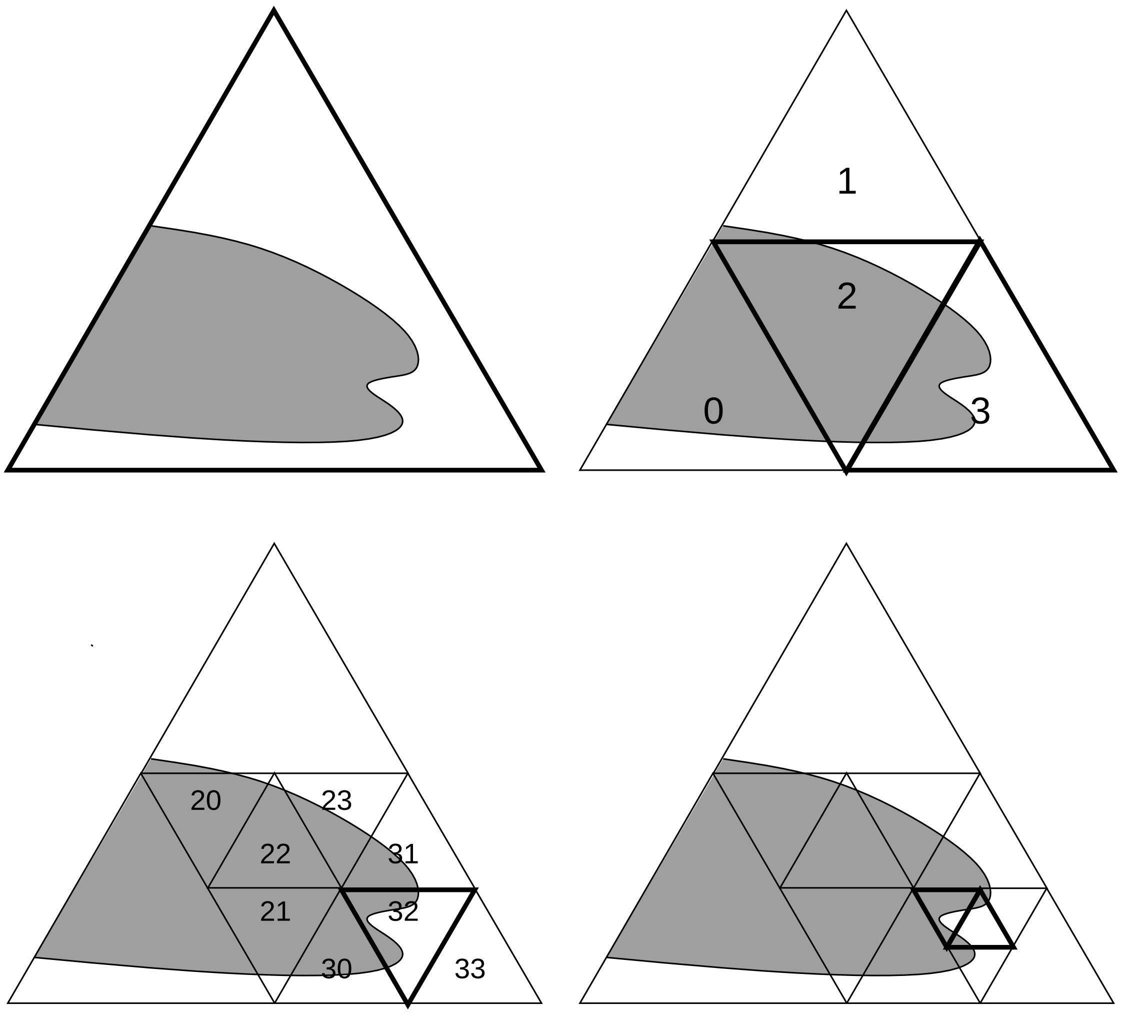


Figure 3.2: Adaptive subdivision of a triangle

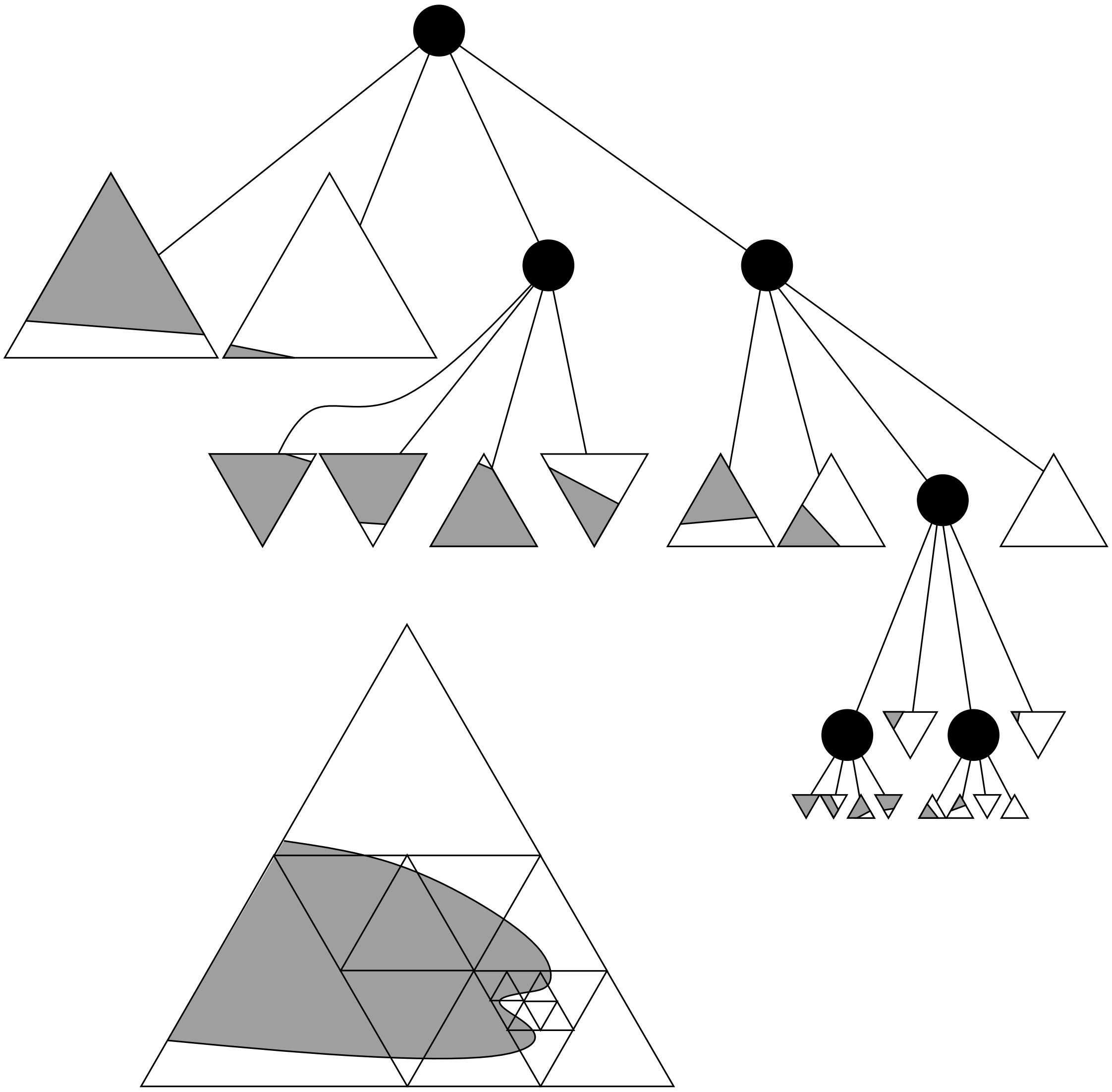


Figure 3.3: Defining a quadtree

3.1.2 Direct quadtree triangulation

Two elements at the same level in a quadtree are *neighbors* if their intersection contains a common edge, and *extended neighbors* if their intersection contains a common vertex. A quadtree fails to be a triangulation where only one of a pair of neighboring elements is split. These definitions generalize the definitions given for square quadtrees by Bern et al. [BEG90]. There are several known techniques for producing a triangulation by dividing only those leaf elements of the tree that have split neighbors [HW90].

Projection is the basis for one such technique. Suppose that a triangular element is surrounded by a collection of split and unsplit neighbors. Introduce a new vertex in the center of the element, and project from the intersection of any leaf with the element to the central vertex. Each projection creates a subtriangle, and the set of such triangles fill the element. The left panel of figure 3.4 illustrates the application of this technique to the quadtree of figure 3.2. The generalization of the projection technique to d dimensions requires that all the facets of an element be triangulated first, by projection, before dividing the volume itself. An algorithm might work by triangulating first all the edges of all elements, then all the faces, then all the 3-cells, and so on.

The projection technique is not minimal; that is, it creates more than the fewest possible elements required to restore the triangulation. The right panel of figure 3.4 illustrates a minimal triangulation for the example of figure 3.2. One algorithm for minimal planar triangulation works on each element by projecting each intersection of one particular edge with a leaf to to the vertex opposite that edge, rather than to a central vertex, then triangulating the newly created elements. The higher-dimensional version is similar, and assumes that the facets of an element are triangulated before the element itself. This method is a slight variation on a common technique in combinatorial geometry described, for example, by Clarkson [Cla88].

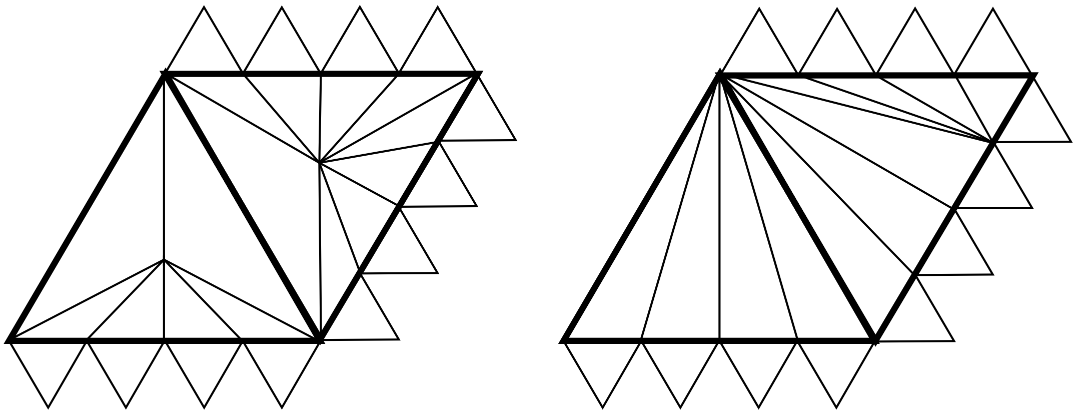


Figure 3.4: Two direct triangulations

These methods share a common weakness; the aspect ratios of the elements they produce can be arbitrarily bad. When the difference in depth of adjacent leaves is great, splitting the shallow element creates many splinter elements. The examples of figure 3.4 illustrates this. Two neighboring triangles, with highly split neighbors along one and two edges, respectively, split to produce nearly degenerate triangles. A solution is to prevent the tree from having two leaves at greatly different depths that share part of an edge.

3.2 Balancing an adaptive quadtree

3.2.1 Theoretical limits – smooth quadtrees

Definition In a *balanced* quadtree, any pair of neighbors differs in quadtree depth by at most one. In a *smooth* tree, any pair of extended neighbors differs in depth by at most one.

A balance condition prevents abrupt changes in element size. Any of the techniques of the previous section, when applied to balanced simplicial quadtrees, produce simplicial elements with bounded aspect ratios. Section 3.3 presents a particularly simple algorithm for triangulating a balanced simplicial quadtree.

Bern et al. define balance for square quadtrees [BEG90], and Von Herzen and Barr call balanced square quadtrees *restricted* quadtrees [VB87]. Bank, Sherman and Weiser call balanced square and triangular quadtrees *1-irregular*, and Weiser has shown that splitting the elements of an arbitrary quadtree to produce a balanced square or triangular quadtree increases the size of the tree by a factor of no more than nine or thirteen, respectively [Wei81,BSW83]. A similar result holds for general smooth quadtrees, and follows from a similar proof.

Theorem 7 *A quadtree with n nodes can be refined to form a smooth quadtree with no more than Cn nodes, where C is the maximum number of extended neighbors an element has, counting itself.*

The proof follows from the following lemma.

Lemma 1 *If a leaf in the quadtree has no split extended neighbors, then the leaf is not split to attain smoothness.*

The proof of the lemma is by induction on the levels of the quadtree. No leaf at the deepest level of the tree splits, since splitting for smoothness only occurs when an element has a split extended neighbor. Suppose that a leaf at level l has no split extended neighbors. When one of its extended neighbors splits, the offspring are of depth $l + 1$, and the offspring that are extended neighbors of the level l leaf have, at the moment of their creation, no split extended neighbors. By induction, those offspring do not split to achieve smoothness. Thus, no extended neighbor of the level l leaf can produce an adjacent descendent of depth greater than $l+1$, and there can never be a violation of the smoothness condition at any of the vertices of the leaf.

From the converse of the lemma, that an element may split only if it has a split extended neighbor, comes the proof of the theorem.

Proof When an element at level l splits, creating a new internal node at level l and new leaves at level $l+1$, assign the "blame" for the splitting to an initially-split

dimension	growth factor
1	3
2	13
3	71
4	461
5	3447
6	29093
7	273343
8	2829325
9	31998903

Table 3.1: Growth factor bounds for smoothed simplicial quadtrees

neighbor of the newly-split element. By the lemma, such a neighbor exists. No internal node can be blamed for more subdivisions than it has extended neighbors. □

For d-dimensional box quadtrees, $C = 3^d$, and for d-dimensional simplicial quadtrees, $C \approx d(d+1)!$; its precise values for dimensions one to nine have been computed by a brute force technique, and appear in table 3.1. For square quadtrees, the bound is tight, as can be seen by considering one family of quadtrees. Let T_0 denote the quadtree with no internal nodes representing the unit square. Build T_i from T_{i-1} by subdividing the leaf of T_{i-1} that contains $(0.4, 0.3)$. For sufficiently large i, T_i requires eight more element splits to achieve smoothness than does T_{i-1}. The elements split in successive levels are chosen in a cyclic order, first the northwest corner, then southwest, then southeast, and so on.

Conjecture A generalization of this construction to higher dimensions, using a Gray code, and to simplicial meshes is possible, and shows the bound to be tight in every case.

3.2.2 Theoretical limits – balanced quadtrees

Trivially, the upper bound on quadtree growth for smoothing holds also for balancing, since any smooth quadtree is balanced. The upper bound is not tight, however.

Theorem 8 *An arbitrary square quadtree with n nodes can be refined to form a balanced square quadtree with no more than $8n$ nodes.*

Like the proof for the smoothness bound above, the construction that leads to this upper bound begins by finding a *barrier*, or configuration of elements, around a designated element sufficient to guarantee that the designated element never splits to help achieve balance. The previous section describes a barrier that protects a square element from splitting either for balance or for smoothness. That barrier is sufficient, but not necessary, to protect a square element from splitting for balance. To build a weaker sufficient barrier, take the four neighbors of an element and the twelve elements that are both children of extended neighbors and adjacent to neighbors of the designated element. Such a barrier is illustrated in the left half of figure 3.5. If a series of element splits can produce such a barrier in which none of the elements of the barrier is split, then the designated element is *safe.*

Lemma 2 *A safe element need never split to achieve balance.*

The proof of the lemma is by induction on the levels of the quadtree. No leaf at the deepest level of the tree splits, since splitting for balance only occurs when an element has a split neighbor. Suppose that a designated leaf at level l is safe. Splitting every element on the boundary of the barrier to improve balance creates a collection of safe elements that are adjacent to the designated element; one of these elements and its barrier is highlighted in the right half of figure 3.5. By induction on the depth l, none of these elements splits to achieve balance. Since

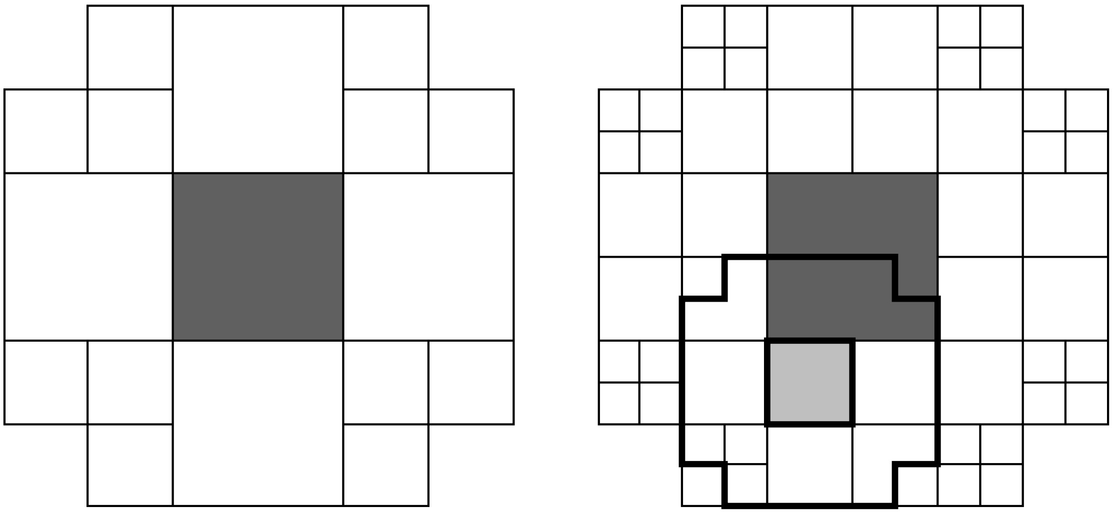


Figure 3.5: A square barrier

the designated element shares edges only with elements at one deeper level that never split, the designated element never splits.

From the lemma, and an accounting scheme that assigns the cost of an element split appropriately, the proof of the theorem follows.

Proof If a designated element splits to achieve balance, it must not be safe. Thus, one or more of the sixteen elements that could have comprised its barrier is split and fails to protect it. Choose one of those split elements and assign it the blame for the splitting of the element for balance. A split element may accept the blame for the splitting of its four neighbors (elements 1-4 in figure 3.6). It may also accept the blame for the splitting of three elements that are extended neighbors, but not neighbors, of its parent (elements 5-7). Since a split element can be blamed only for seven other split elements, balancing the quadtree can increase its number of nodes by only a factor of eight. □

The upper bound of the previous theorem is essentially tight, as demonstrated by the family of quadtrees depicted in figure 3.7. The first member of the family ($k = 1$) is the quadtree defined by the solid lines inside the innermost white square of that figure. That quadtree consists of eleven internal nodes, and balancing that

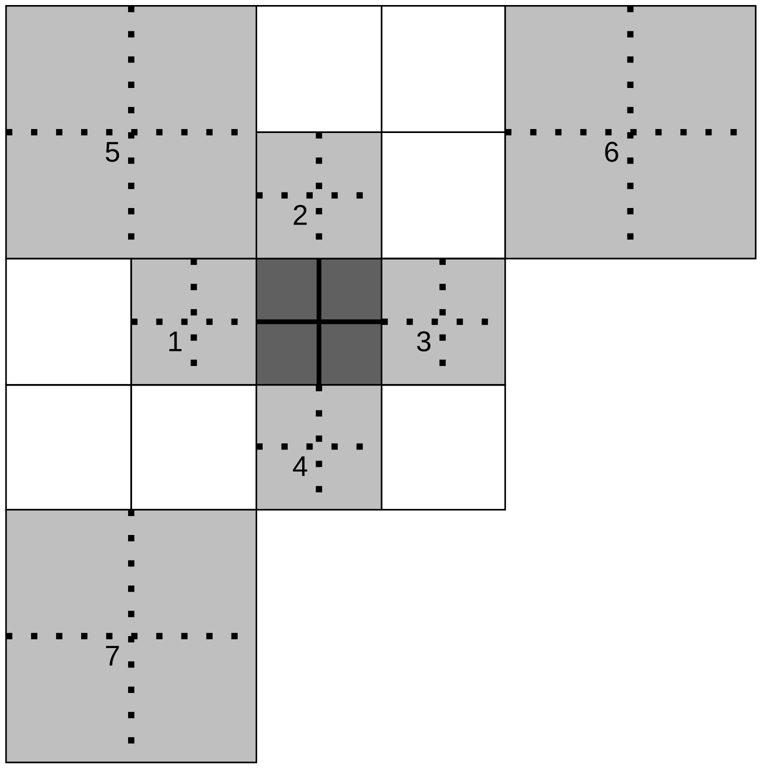


Figure 3.6: A scheme that charges up to 7 splits to a split square element

quadtree produces another quadtree, represented by the solid and dashed lines, with thirty internal nodes. In general, to build the kth quadtree in this family, begin with a simple quadtree with an internal node and four leaves. The *current* leaf is the unsplit element that has the central vertex of the quadtree at its northwest corner. Split the current leaf k times, then split the current leaf, its children, and its four most distant grandchildren. The gray square and the solid lines of various thicknesses within it represent the second quadtree in the sequence, and the dashed lines indicate additional elements in that quadtree split for balance. Similarly, the largest square represents the third quadtree in the sequence.

Every quadtree in the sequence contains one more internal node than its predecessor, and balancing it requires seven more element splits than were necessary to balance its predecessor. Thus, the quadtree in this family with n internal nodes yields, when balanced, a quadtree with $8n - 58$ nodes.

Weiser's upper bound on triangular quadtree growth is also an overestimate, as demonstrated by the following theorem.

Theorem 9 *An arbitrary triangular quadtree with n nodes can be refined to form a balanced triangular quadtree with no more than $10n$ nodes.*

A barrier for a triangular element is depicted in the left half of figure 3.8. The barrier consists of all the neighbors of the element, and all the elements that are children of extended neighbors and adjacent to neighbors of the element.

If a series of element splits can produce such a barrier in which none of the barrier elements is split, then the designated element is safe. The proof by induction that a safe triangular element never splits to achieve balance is similar to the proof in the square case. The right half of figure 3.8 demonstrates the result of splitting every element on the boundary of the barrier to improve balance, then splitting other elements of the barrier as necessary to maintain balance locally. This splitting creates a collection of safe elements that are adjacent to the designated element; one of these elements and its barrier is highlighted in the figure. By induction

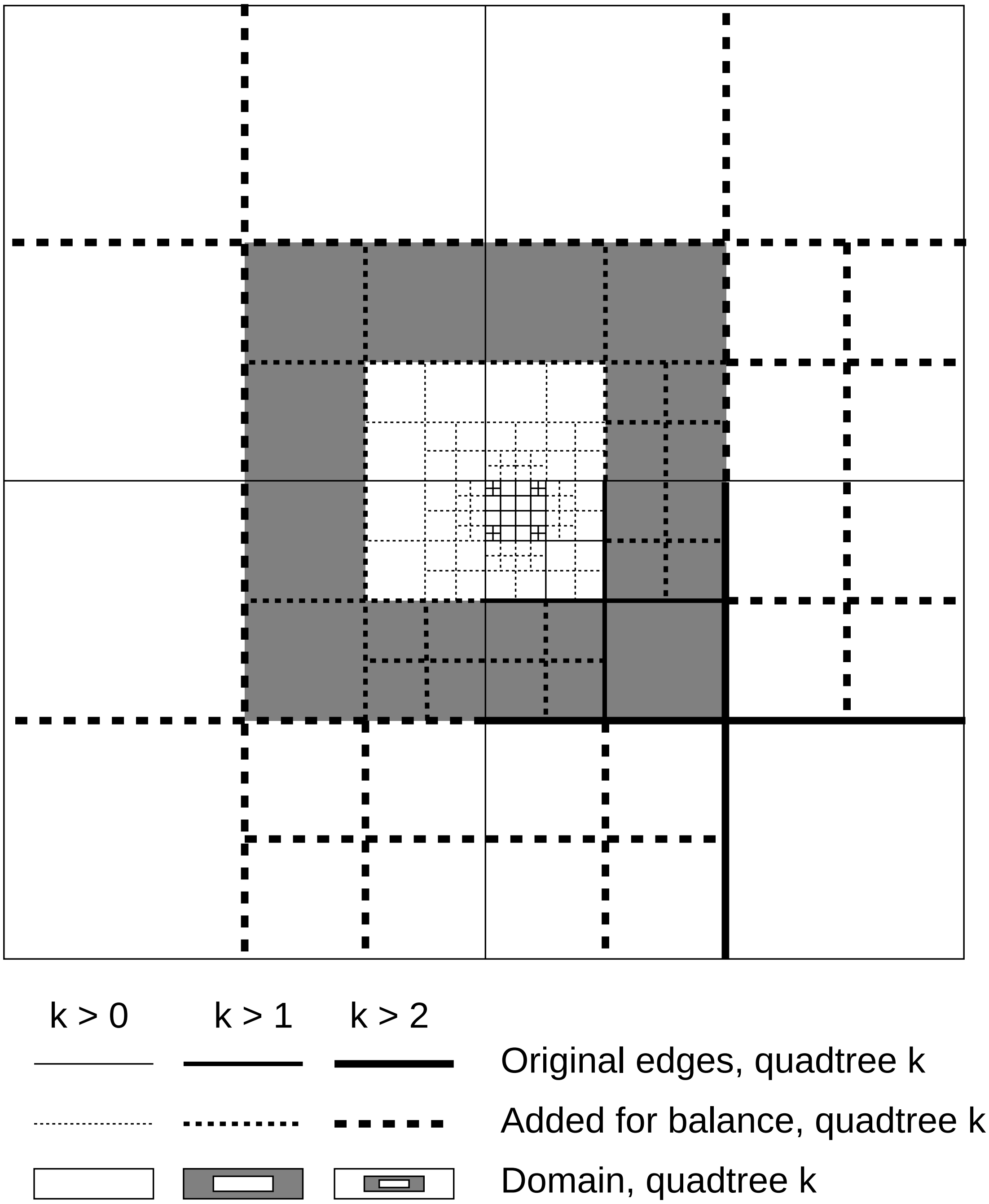


Figure 3.7: Balancing square quadtrees can make them eight times larger

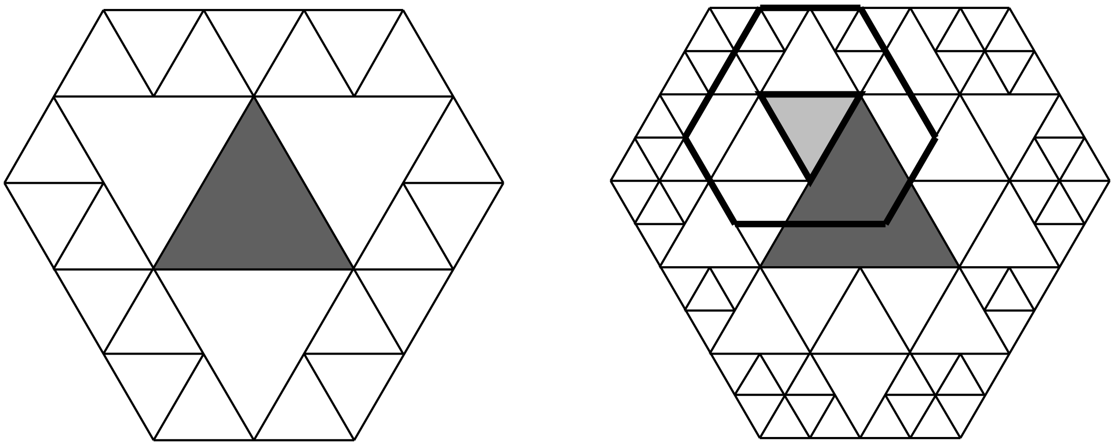


Figure 3.8: A triangular barrier

on the depth of the designated element, none of these adjacent elements splits to achieve balance. Since the designated element shares edges only with elements at one deeper level that never split, the designated element never splits.

Thus, by assigning the costs of element splits appropriately, the proof of the theorem follows.

Proof If a triangular element splits to achieve balance, it must not have a barrier. Thus, one or more of the twenty-four elements that would have comprised a barrier is split. Choose one of those split elements and assign it the cost of splitting the element for balance. This scheme can charge any split element in the quadtree for the splitting of its three neighbors (elements 1-3 in figure 3.9). A middle element, one that does not share a vertex with its parent, can pay for the splitting of six elements that are extended neighbors, but not neighbors, of its parent (elements 4a-9a). An outer element, one that does share a vertex with its parent, can pay for the splitting of five such larger elements (elements 4b-8b). Since no split element can be charged for more than nine other split elements, balancing the quadtree can increase its number of nodes by only a factor of ten. □

A precise growth bound for triangular quadtrees remains unknown. A more careful analysis of the barrier above, or the construction of a more complex barrier,

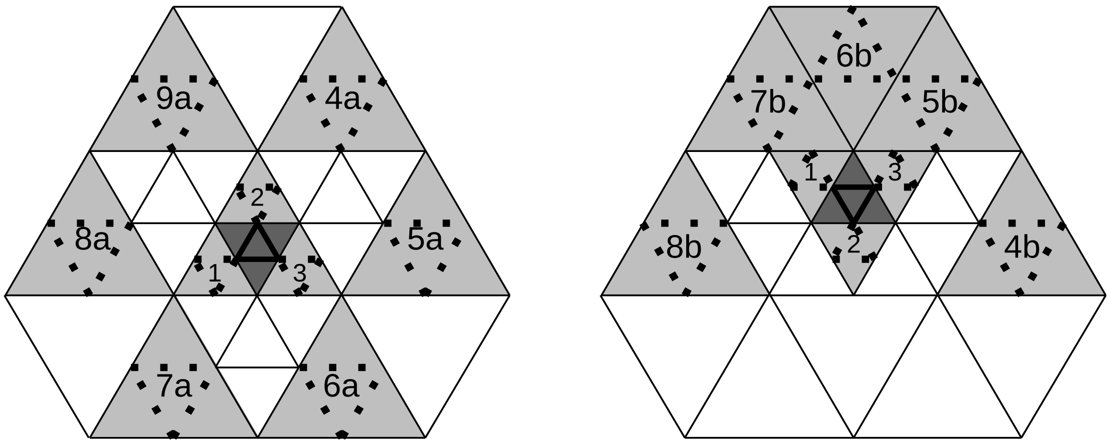


Figure 3.9: A scheme that charges up to 10 splits to a split triangular element

could reduce the upper bound. Highest lower growth bound found so far is six, but this bound can probably be raised. More work on these bounds, and on the growth of quadtrees of both kinds in higher dimensions, remains to be done.

3.2.3 Buffering to maintain smoothness

Samet and Von Herzen-Barr incompletely describe an algorithm for producing a balanced quadtree. Their algorithm subdivides any element that is adjacent to the grandchild of a neighbor, until the quadtree has no such elements. The organization of the search for these elements is not specified. The REFINE procedure of Bank et al. [BSW83] produces balanced quadtrees with a simple algorithm, but requires maintenance of a data structure that can quickly locate the unsplit neighbors of an element and its ancestors.

This section and the next each describe a way to organize a quadtree smoothing algorithm, one integrated into the quadtree construction algorithm that subdivides more elements than necessary for smoothness, and one that begins with a quadtree and efficiently produces the smallest smooth refinement of it. Neither algorithm requires neighbor finding, but instead relies on a data structure that associates global data with the vertices of elements.

The *buffer method* builds a quadtree by breadth-first, rather than depth-first, search. The method begins with a single inaccurate element and a single noded tree. The process that builds the next level of the tree from the current level maintains two collections of leaf elements. The *core* set contains all elements that must split because they fail the accuracy predicate. The *buffer* set includes the extended neighbors of nodes in the core set, and possibly other elements. The remaining inactive leaf elements are at current and previous levels of the quadtree and play no role in the remainder of the algorithm.

In the processing of a single level, each element of the core set is split, and its children tested for accuracy. Those elements deemed insufficiently accurate are added to the core set for the next iteration, and their vertices are accumulated into a set to permit the fast recognition of extended neighbors of the core. The remaining elements are added to the buffer set for the next level.

Then, each element of the buffer set is examined, and if its vertices include a vertex of an element in the next core set, then it is split and its offspring are put into the buffer for the next level. The rest of the buffer elements are set aside. The algorithm of figure 3.10 presents a more structured description of the algorithm.

Aside from the operations on the data structure that maintains the vertex set, and the time spent testing element accuracy, the algorithm takes time proportional to the size of the quadtree it produces, because each element is examined only a few times.

Figure 3.11 shows an example of buffered subdivision applied to the example of figure 3.2. In each frame, the bold elements lie within the core set, the dotted elements are the completely processed, inactive elements, and the remaining elements form the buffer. Core elements split to produce the next set of core elements, and part of the next set of buffer elements. Of the remaining elements, those that share vertices with elements of the next core split and added to the next buffer, and the rest are thrown away.

```
void Quadtree::refine (const Element& E, bool (*accurate)(const Element&))
{ Queue core, buffer, next_core, next_buffer;
  core.enqueue(*this, E);
  while (!core.empty()) {
      PointSet vtxSet;
      while (!core.empty()) {
          core.dequeue(node, E);
          node.createChildren();
          for (int k = 0; k < nChildren(); ++k) {
            Element offspring = E.subElement(k);
            if (accurate(offspring))
              next_buffer.enqueue(node.child[k], offspring);
            else {
              vtxSet.add(offspring.allVtxs());
              next_core.enqueue(node.child[k], offspring);
        } } }
      core.append(next_core); // leaves next_core empty

      while (!buffer.empty()) {
          buffer.dequeue(node, E);
          if (vtxSet.intersects(E.allVtxs())) {
            node.createChildren();
            for (int k = 0; k < nChildren(); ++k)
              next_buffer.enqueue(node.child[k], E.subElement(k));
        } }
      buffer.append(next_buffer);
} }
```

Figure 3.10: Algorithm for adaptive refinement with buffering

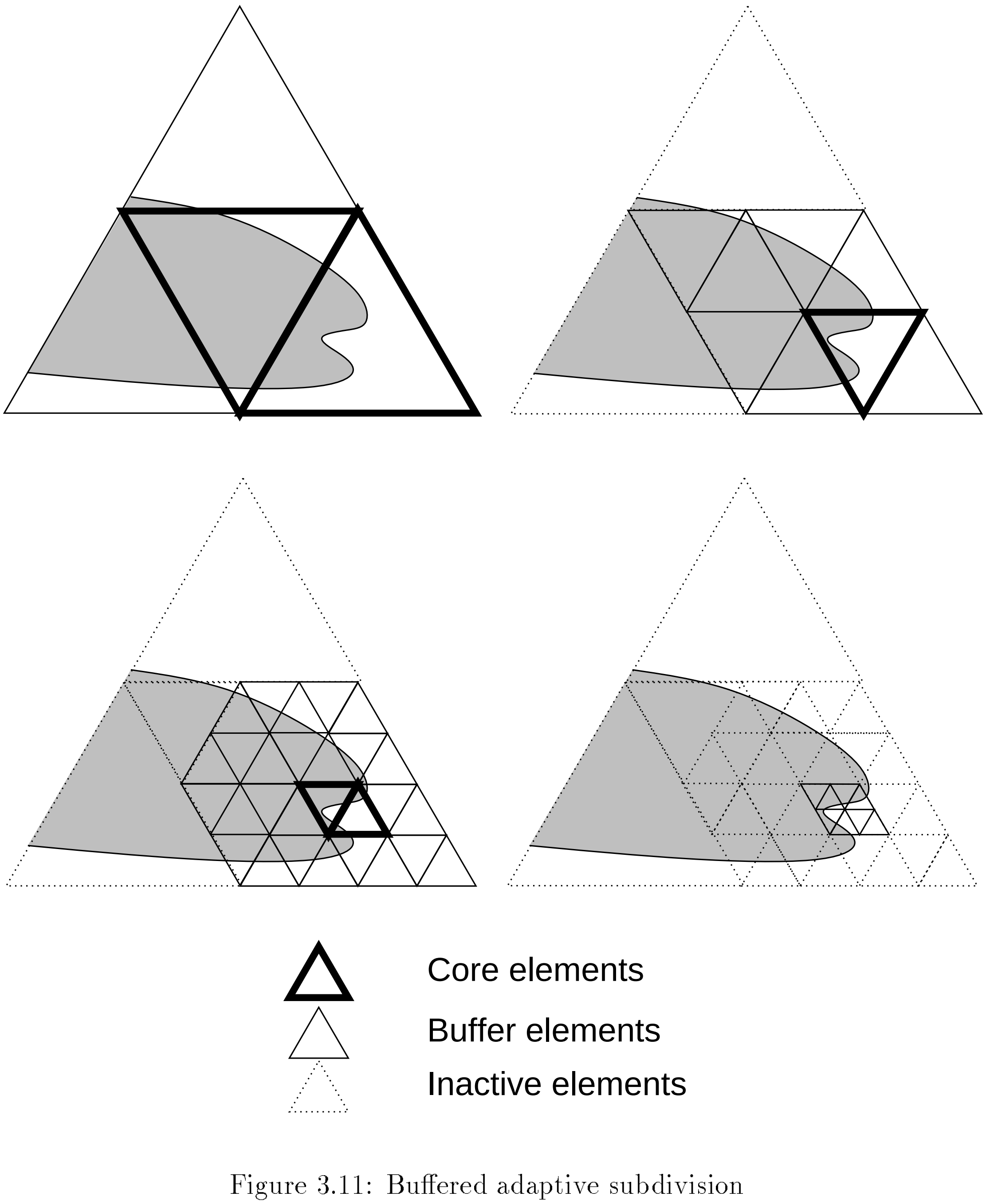


Figure 3.11: Buffered adaptive subdivision

Theorem 10 *If the accuracy predicate is both satisfiable and hereditary, the buffer method produces a smooth quadtree of accurate elements.*

Proof The proof is by induction on the height of the tree.

Let B and C denote the buffer and core elements at one level of the tree, and let D denote the other elements. Let A denote the extended neighbors in B of elements in C. Finally, let A',B',C' and D' denote the corresponding sets for the next level of the tree. The inductive claim is that at any step of the algorithm, the quadtree is smooth, and that no element of A shares a vertex with an element in D. Initially, take the core C as the single element of the tree. Then the tree is smooth, and the inductive claim holds.

In one refinement step of the buffer method, some elements of $A \cup C$ subdivide. No element that splits shares a vertex with an element in D, by the induction hypothesis, so smoothness is preserved. Because the accuracy predicate is hereditary, the inaccurate elements at the next level are all children of inaccurate elements at the current level. Call the set of inaccurate children of core elements C' to preserve the invariant that the core contains only inaccurate elements. From the algorithm, $B' \subseteq \text{children}(C \cup A) - C'$. An element $E \in A$ can split to form smaller elements, but none of the smaller elements is both an extended neighbor of a child of an element of C and adjacent to an element of $B - A$. The elements of $B - A$ become elements of D' at the next level, so none of the smaller elements is adjacent both to an element of C' and to an element of D'. $\Box$

One primary advantage of the buffer method is that is local; An element splits based only on information about itself and its extended neighbors. This property can be useful for a parallel implementation, since it limits interprocessor communication. Moreover, once all elements at one level have been adaptively split the elements at higher levels are never subdivided again. This allows finite element computations, for example, to begin on coarser elements before the subdivision of the finest elements is complete. In a computer graphics application such as a surface renderer, the portions of the quadtree at higher levels can be processed and discarded immediately.

The method has serious shortcomings as well. By splitting extra elements to maintain the buffer, the method can produce more elements. Experience with implementations of the buffer method indicates that, in graphics applications, the method typically creates $40-50\%$ more elements than required to achieve smoothness.

3.2.4 A heuristic to improve smoothness

To produce a smooth quadtree of minimum size, an algorithm must follow the general approach that Von Herzen-Barr and Samet describe, and split only elements that must be split. One simple way to organize an algorithm to split at least some of those elements is to use a quadtree traversal algorithm to visit all the leaf elements in some order. A *traversal* is a simple recursive procedure to visit all the nodes in a tree. A *forward* traversal visits each leaf in a quadtree by visiting the root and, if it is not a leaf, visiting each of its subtrees in the natural order associated with subdivision of an element. A *backward* traversal is the same, but visits the subtrees in the reverse of the order associated with subdivision.

The heuristic *semismoothing* algorithm improves the smoothness of the tree by making either a forward or a backward traversal of it. When a leaf is visited in the traversal, its depth in the quadtree is compared to the depths of the deepest elements with which it shares a vertex. If an element shares a vertex with an element two levels deeper, the element must split, and the heuristic visits the children in the appropriate order. Figure 3.12 presents the major components of the heuristic more formally.

The procedure uses an auxiliary data structure to store, for each vertex of any element, the depth of the deepest element containing the vertex. Abstractly, the data structure is an associative array. Concretely, the implementation may be a

```
void Quadtree::semi_smooth
(int depth, // depth of current node
 const Element& E, // element corresponding to this tree
 DepthMap vtxDepth) // vtxDepth.contains(v) iff v is an element vertex
{ // vtxDepth[v] is the depth of the deepest element at v
  if (isLeaf())
    if (hasDeeperNeighbors(depth, E, vtxDepth))
      {createChildren(); updateDepthInfo(depth+1, E, vtxDepth);}
    else return;

  for (int k = firstChild; k ≠ beyondChildren; k += toNextChild)
    child[k].semi_smooth(depth+1, E.subElement(k), vtxDepth);
}

bool hasDeeperNeighbors(int depth, const Element& E, const DepthMap&
vtxDepth)
{
  for (int j = 0; j < E.nVtxs(); ++j)
    if (vtxDepth[E.vtx(j)] > depth+1)
      return TRUE;
  return FALSE;
}
```

Figure 3.12: Heuristic to partially smooth quadtrees

large multidimensional array, since each vertex in the quadtree can be mapped to a point that lies in a subset of the integer grid. Practically, however, the implementation should be a hash table, for expected constant time queries, or an augmentation of the quadtree to associate each vertex of every element with a location in a global table, where information about that vertex is stored.

From the algorithmic considerations of the preceding paragraphs, the next theorem follows.

Theorem 11 *To produce an n-node quadtree, semismoothing can take $O(vn)$ time, where v denotes the number of vertices of an element.*

Figure 3.13 illustrates the result of applying the procedure to the quadtree of figure 3.2. The upper panel illustrates the initial partition, with labels that show vertex depths. The left and right panels illustrate the result of one traversal in the forward order (left, up, middle, right), then another in the reverse order, respectively.

As the previous example demonstrates, semismoothing is not a smoothing algorithm. A one-dimensional example demonstrates the same phenomenon. The four panels of figure 3.14 reflect repeated applications of semismoothing to the initial quadtree, labeled A. Panels B, C, and D portray the state of the tree after one, two, and three applications of semismoothing from left to right. They also portray the state of the tree during a single application of semismoothing from right to left. Consideration of this example leads to the hypothesis, easily proven, that two applications of semismoothing applied in opposite directions, smoothes the tree.

From the definition of smoothness, the number of applications of semismoothing needed to smooth the quadtree completely is bounded by a constant multiple of the quadtree size. For one-dimensional trees, the number of traversals required is only two, if they are in opposite directions. For two-dimensional trees of both the square and triangular types, one semismoothing traversal in each direction is usually enough to smooth the quadtree, but is occasionally insufficient.

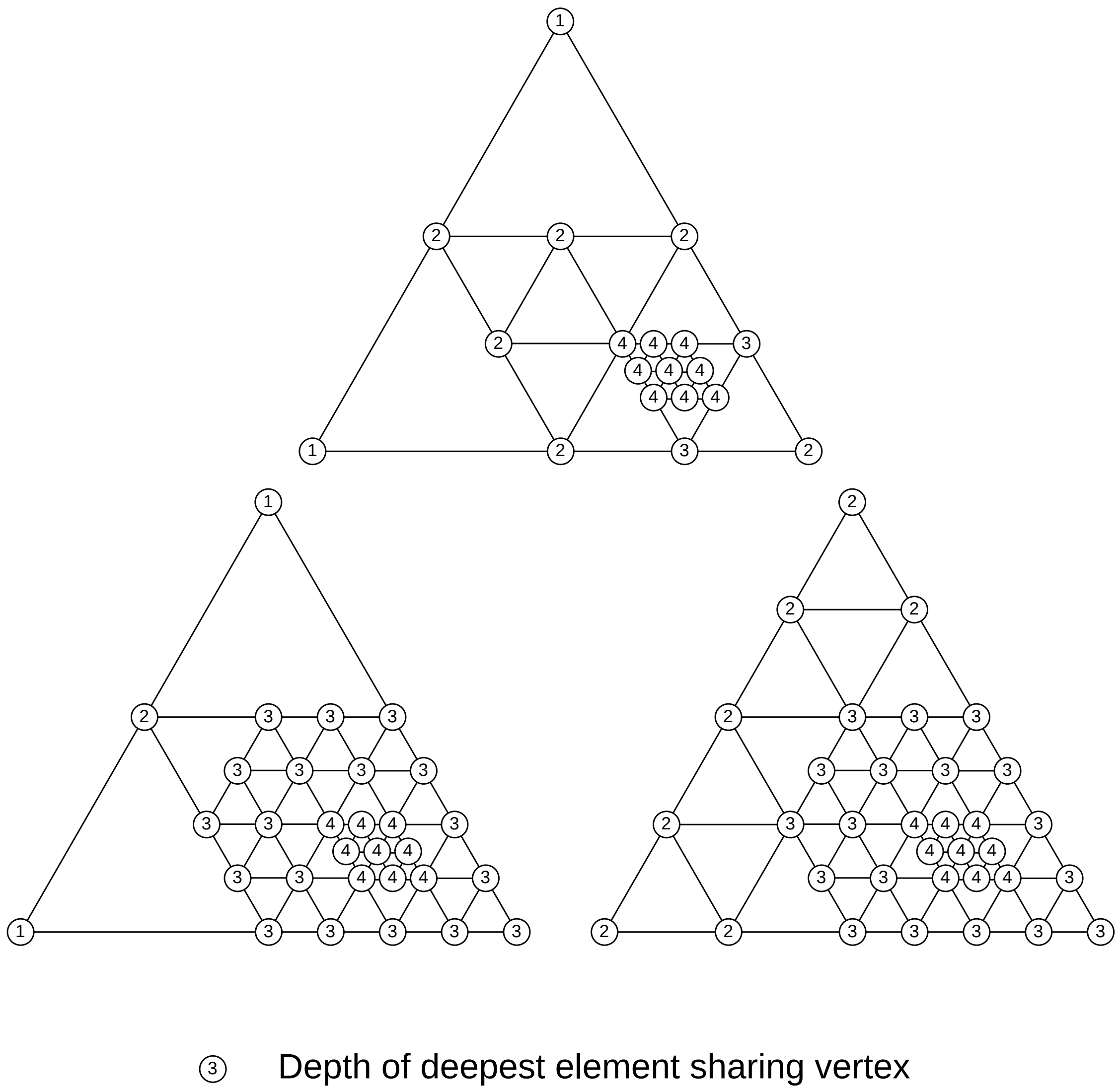


Figure 3.13: Smoothing a quadtree.

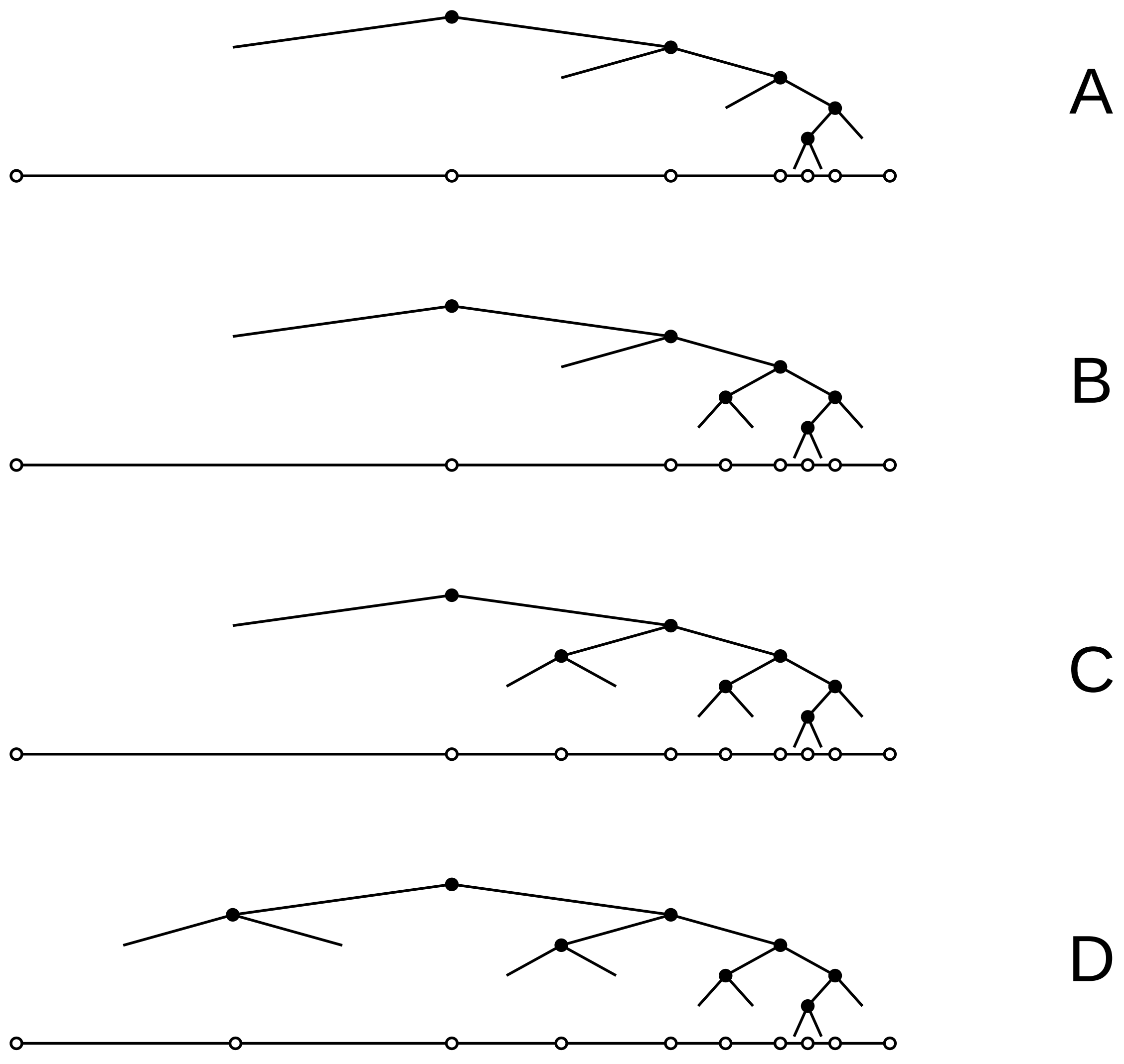


Figure 3.14: Smoothing a one-dimensional quadtree

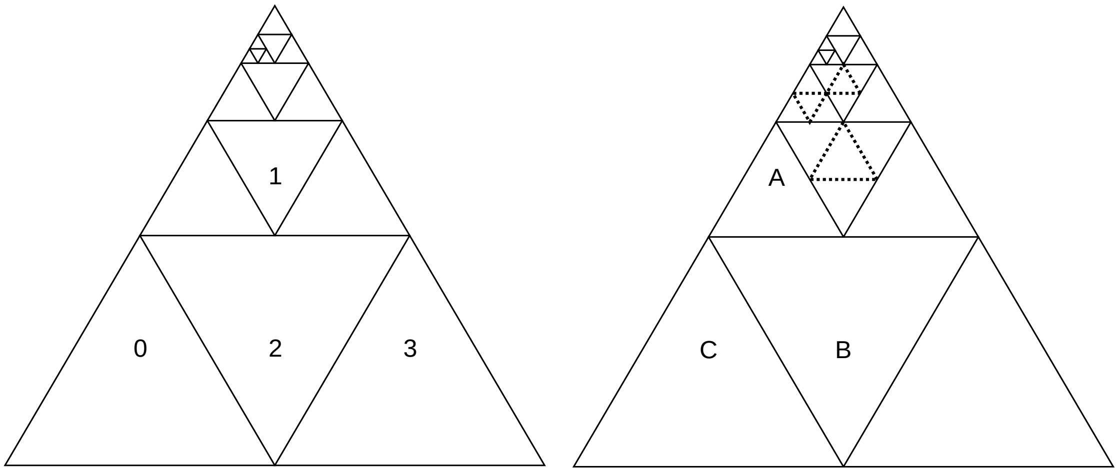


Figure 3.15: Semismoothing twice fails to smooth a triangular quadtree

Figure 3.15 gives an example. In the left part of the figure, the numbers indicate the ordering of the children of the root of the quadtree. In the right part of the figure, dashed triangles identify the elements split in a forward semismoothing traversal of the quadtree. The triangles labeled A and C split for smoothness on the backward semismoothing traversal, leaving A unsmooth with respect to B.

Figure 3.16 gives an example of the failure of back-and-forth semismoothings for square quadtrees; it is essentially the simplest possible example. The backward traversal, which visits the children of a node in the order southeast, southwest, northeast, northwest, splits the element labeled A and its right neighbor. The forward traversal splits the element B only, and element C is left unsmooth with respect to B.

The next section demonstrates that three semismoothing traversals are sufficient to smooth a square quadtree. A constant number of semismoothings may provide a general smoothing algorithm, where the constant depends on the kind of quadtree smoothed. More work in this area remains to be done.

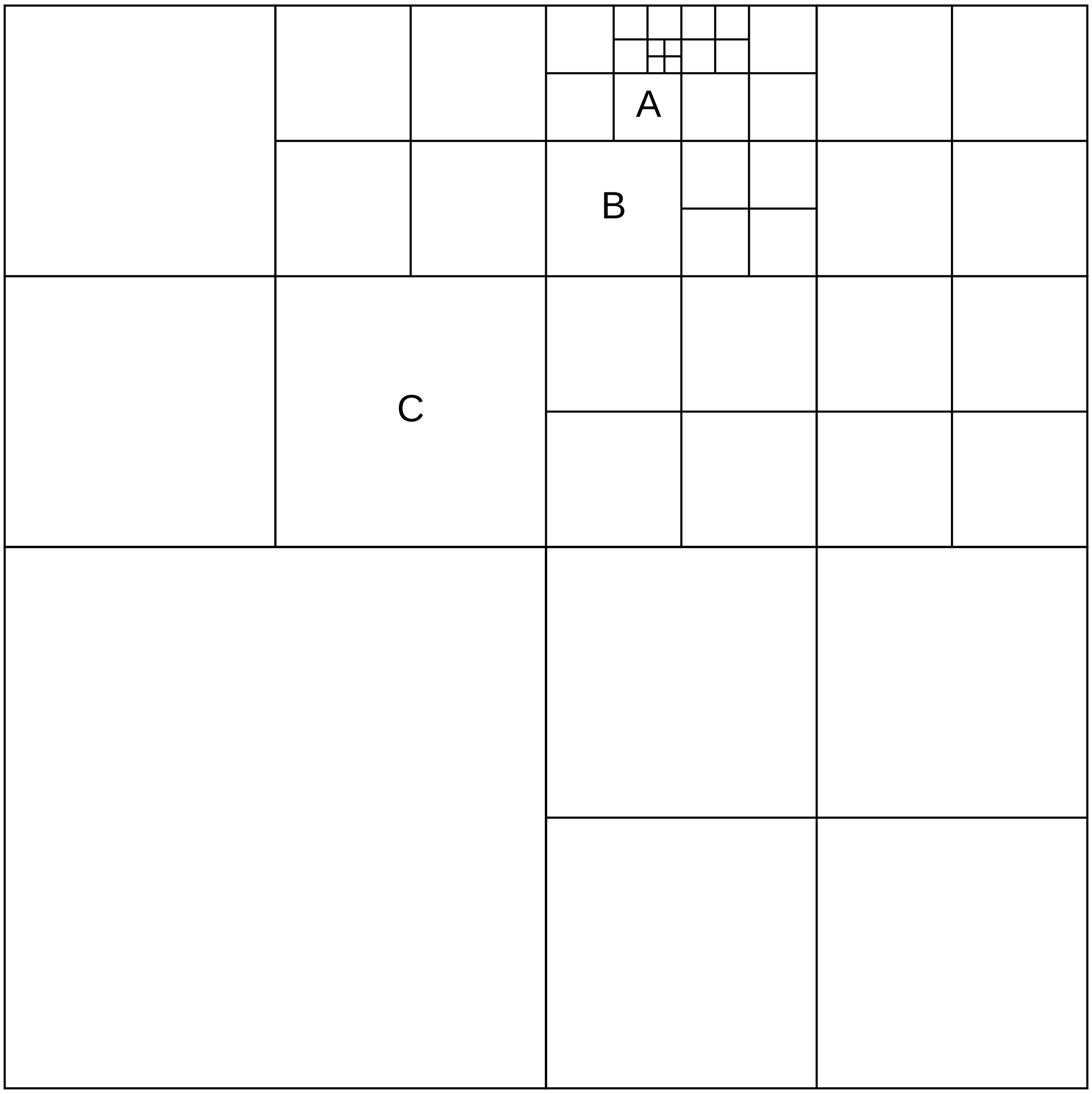


Figure 3.16: Semismoothing twice fails to smooth a square quadtree

3.2.5 Smoothing square quadtrees

Given a square quadtree, a simple algorithm can restore smoothness to the tree by splitting the fewest possible elements. The complete smoothing algorithm initializes the associative array of vertex depths and semismoothes three times, with a backward traversal, a forward traversal, and another backward traversal. The ordering of the children of a square quadtree node is "row-major", from top to bottom and from left to right within a row. Figure 3.17 presents the algorithm and minor auxiliary functions.

Theorem 12 *The algorithm of figure 3.17 smoothes a square quadtree and splits the fewest possible elements necessary to do so.*

Proof The second part of the claim is easily verified, as no element is split unless it shares a vertex with an element two levels deeper.

To establish the primary claim, consider the set of leaf elements as they exist after the first application of semismoothing. Assume that after the backward traversal, a forward traversal of the partially smoothed tree would visit the elements in order E_0, E_1, ...

Lemma 3 *After backward semismoothing, no unsplit element E_r shares a vertex with an element E_s, $r < s$, two levels deeper in the tree.*

In the backward traversal, E_r is visited after E_s. If E_s were two levels deeper than E_r, the procedure would have split E_r.

Lemma 4 *After backward semismoothing, no unsplit element E_s shares a vertex with an element E_r, $r < s$, three levels deeper in the tree.*

Assume, conversely, that E_r is three levels deeper than E_s, that E_r and E_s have a common vertex, and that E_s is the first such element examined in the backward traversal. After the backward traversal visited E_s, E_s shared no vertex with an

```
void Quadtree::smooth(const Element& E)
{ // element corresponding to this tree
  Depthmap vtxDepth; // initially empty
  calcDepths(0, E, vtxDepth);

  int firstChild = nChildren() - 1, beyondChildren = -1, toNextChild = -1;
  semi_smooth(0, E, vtxDepth); // walk tree backward

  firstChild = 0, beyondChildren = nChildren(), toNextChild = 1;
  semi_smooth(0, E, vtxDepth); // walk tree forward

  firstChild = nChildren() - 1, beyondChildren = -1, toNextChild = -1;
  semi_smooth(0, E, vtxDepth); // walk tree backward again
}

void Quadtree::calcDepths(int depth, const Element& E, DepthMap&
vtxDepth)
{
  if (isLeaf())
    for (int j = 0; j < nVtxs(); ++j)
      // "a >?= b" means "a = max(a,b)"
      vtxDepth[E.vtx(j)] >?= depth;
  else
    for (int k = 0; k < nChildren(); ++k)
      child[k].calcDepths(depth+1, E.subElement(k), vtxDepth);
}
```

Figure 3.17: Algorithm for square quadtree smoothing

element two levels deeper. Thus, the extended neighbor E_r must be the result of the splitting of an element P_r that shares an element with E_s and is two levels deeper. In turn, P_r must be the result of the splitting of some child of an extended neighbor of E_s. However, when that child split to create P_r, P_r had no split extended neighbors. By lemma 1, an element with no split extended neighbors never splits to achieve smoothness, so P_r is not split and E_r cannot be created.

Thus, after the backward traversal of the quadtree, the only violation of smoothness criteria occurs when an element E_s shares a vertex with an unsplit element E_r two levels deeper and $r < s$.

Lemma 5 *If, in the forward traversal, a leaf element E_s is encountered that shares a vertex with an unsplit element E_r, $r < s$, two levels deeper in the quadtree, and another vertex with an unsplit element E_t one level higher in the quadtree, then E_r is a descendent of P_s, the parent of E_s.*

If an edge of E_s connects a vertex of E_r with a vertex of E_t, then by the choice of E_s, P_s is the first element of its depth containing that edge to be visited. Since $r < s$, E_r must be a descendent of that element. If no edge of E_s connects a vertex of E_r to a vertex of E_t, then E_r must lie strictly within the interior of P_s. Thus, E_r is a descendent of P_s.

Lemma 6 *If, in the forward traversal, a leaf element E_s is encountered that shares a vertex with an unsplit element E_r, $r < s$, two levels deeper in the quadtree, and another vertex with an unsplit element E_t one level higher in the quadtree, then either $r < s < t$ or E_t is a lower right extended neighbor of P_s and $t < r < s$.*

Let E_s be the first leaf element encountered in the forward traversal with extended neighbors E_r and E_t that satisfy the preconditions of the lemma, and for which $t < r < s$. Elements E_r and E_t do not share a common vertex, since they differ in depth by three.

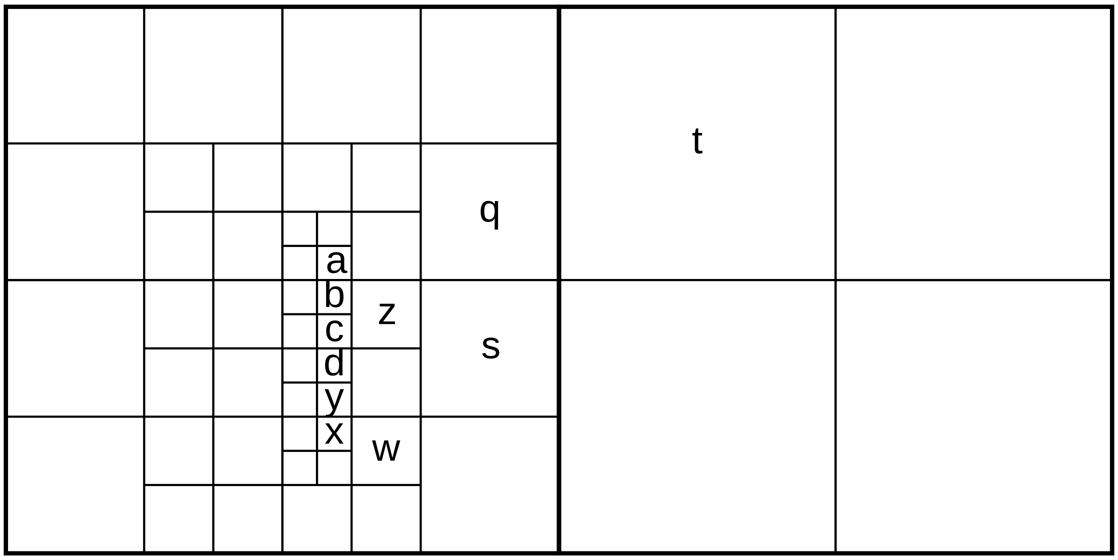


Figure 3.18: Splitting during the forward traversal

Consider the relationship of P_s, the parent of E_s, to E_t. Square quadtrees with the specified ordering of children have the property that in a forward traversal a node is visited before its right and down neighbors, and after its left and up neighbors. If P_s and E_t are neighbors, then P_s is right of or below E_t. Without loss of generality, assume that P_s is right of E_t. Element E_s must be the lower left child of P_s, and its up neighbor is visited after E_r on the backward traversal. When that neighbor is visited, it is split, and one of its vertices is a vertex of E_t. Therefore E_t splits on the backward traversal as well, contradicting the initial conditions of the lemma. Similarly, if P_s is below E_t, then E_s is a upper right child of P_s and again E_t splits on the backward traversal. Therefore, P_s is not a neighbor of E_t.

If P_s is a lower right extended neighbor of E_t, then E_s is an upper left child of P_s. Necessarily, then, $s < r$.

Because the remainder of the analysis is complex, figure 3.18 depicts the next case and labels several elements to simplify the discussion.

If P_s is a lower left extended neighbor of E_t, then E_s is an upper right child of P_s and E_r is a grandchild of the upper left child of P_s. After E_s is visited on the backward traversal, a split creates E_r. Since E_s did not split on the backward traversal, element 'w' is not split after the backward traversal. Therefore, neither element 'x' nor element 'y' is split initially. The split of an element to create E_r must be because one of elements 'a', 'b', 'c' or 'd' is split by the time 'z' is visited. In any of those four cases, element 'z' is split on the backward traversal. When element 'q' is later visited on the backwards traversal, it is split because 'z' is split. Then, when E_t is visited on the forward traversal, it is split because 'q' is split.

Therefore, the only possible relationship of P_s to E_t when $t < r < s$ is the one in which P_s is an upper right extended neighbor of E_t. This relationship is possible, and is the key to the failure of the simple two-pass algorithm.

Lemma 7 *After the forward traversal, the only elements for which smoothness criteria are not satisfied are elements that are right children and which have upper right extended neighbors that are split and have split lower left children split in the forward traversal.*

The splitting of elements that occurs on the forward traversal to achieve smoothness can cause additional elements to have to split only when one of the elements that splits has an unsplit extended neighbor one level higher in the quadtree. Either that unsplit extended neighbor will be visited and split later in the forward traversal or it is a right child of its parent and the lower left extended neighbor of the parent of the element that splits.

All the lemmas that describe the situation before and during the forward traversal apply equally well before and during the second backward traversal with suitable interchanges of up and down, left and right, forward and backward, greater than and less than. In particular, the following version of the last lemma holds.

Lemma 8 *After the second backward traversal, the only elements that do not satisfy smoothness criteria are elements that are left children and which have lower* *left extended neighbors that are split and have split upper right children split in the second backward traversal.*

Thus, an element split in the third traversal must have a split upper right extended neighbor with a lower left child that split in the second traversal. An element that still needs to be split after the third traversal is one with a split lower left extended neighbor that has an upper right child that split in the third traversal. Such an element must have already been split on the second traversal, though, else the splitting on the third traversal would not have been necessary.

Therefore, three traversals are sufficient to smooth the quadtree.

□

3.2.6 Restoring balance

In practice, smoothing can create many more vertices than balancing. The next section presents triangulation procedures that require only balanced quadtrees, so more complex procedures to achieve only balance are worth considering. For example, figure 3.19 presents modifications to the algorithm of figure 3.12 to produce a *semibalancing* algorithm.

Strictly speaking, the procedure does not need to store the depths; recording the presence or absence of a point in the set of element vertices is sufficient. However, the recording of depths often allows a single test to replace the examination of several edges. When the examination of vertex depths consistently fails to eliminate all the edges incident to a vertex, the decision procedure can take time proportional to the number of edges in an element to check each of the points one quarter of the way along each edge. If any vertex is found at one of these points, the element must split to improve balance.

Theorem 13 *To produce an n-node quadtree, semibalancing can take $O(en)$ time, where e denotes the number of edges of an element.*

```
bool hasDeeperNeighbors(int depth, const Element& E, const DepthMap&
vtxDepth)
{
  for (int j = 0; j < E.nVtxs(); ++j)
    if (vtxDepth[E.vtx(j)] > depth+1 &&
        hasDeeperEdgeNeighbors(E.vtxDegree(), E.vtx(j), vtxDepth))
      return TRUE;
  return FALSE;
}

bool hasDeeperEdgeNeighbors(int vtxDegree, const Vertex& v,
                            const DepthMap& vtxDepth)
{
  for (int j = 0; j < vtxDegree; ++j) // find deep edge neighbors.
    // Find vertices 1/4 of the way from v to adjacent vertices.
    if (vtxDepth.contains
        (convex_combination(v, v.adjacentVtx(j), .25)))
      return TRUE;
  return FALSE;
}
```

Figure 3.19: Algorithm to detect imbalance in quadtrees

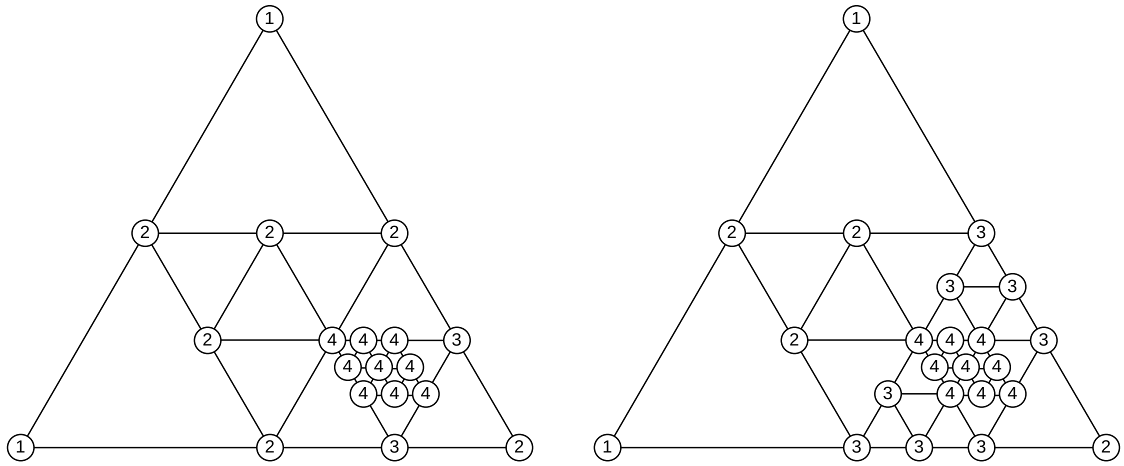


Figure 3.20: Balancing can produce much smaller quadtrees

Figure 3.20, which illustrates the result of semibalancing to the example of figure 3.2, demonstrates that balancing can create a much smaller quadtree than smoothing. The left panel represents, again, the initial quadtree with labels to show vertex depths, and the right panel represents the quadtree that results from one semibalancing traversal in the forward direction. Here, one traversal produces a balanced quadtree.

One forward, followed by one backward, application of semibalancing provides only a semblance of a balancing algorithm. The sequence of trees $\{T_i\}$ constructed above to prove the tightness of an upper bound includes trees that require two back-and-forth semibalancing traversals to achieve balance.

Conjecture The number of passes is likely a function of the maximum difference in depths of extended neighbors in a balanced quadtree.

3.3 Triangulating a balanced quadtree

The direct triangulation methods of section 3.1.2, when applied to balanced simplicial meshes, produce only a small set of shapes. For any simplex, each edge is either split, because some deeper neighbor shares the edge, or it is not, but split edges are split only into two parts, not arbitrarily many. The resulting bound on the number of possible shapes bounds also the aspect ratio of the simplices of the triangulation.

Each method has serious shortcomings. The projection method can introduce new vertices at the centers of k-faces ($k \in \{2 \ldots d\}$) of simplices. In the worst case, a simplex with all split edges is decomposed into $(d+1)!$ simplices, with $2^{d+1} - (1 + (d+1)(d+2)/2)$ new vertices. The simplices so produced tend to have poor aspect ratios. For example, even if the parent of the $(d+1)!$ simplices is regular and the interior point chosen on each face is the incenter of the face, the ratio of maximum to minimum edge length in each child simplex is $\sqrt{2d(d-1)}$. Techniques that subdivide a simplex into the fewest possible subsimplices to achieve a triangulation also tend to produce badly shaped elements.

Another disadvantage of minimal decompositions is that they are not unique. In the simplest case, consider a triangular element with two divided edges and one whole edge. There are two distinct ways to divide the triangle into three subtriangles to restore the triangulation in that region. Often, there is no reason to prefer one such triangulation to another. However, if the triangle is the face shared by two tetrahedral elements, it is important that each tetrahedron subdivide that common face the same way. For minimal triangulations, the subdivision of a facet depends not only on information about the facet itself, but on information about the entire simplex. Ensuring that consistency is maintained across a common face is thus a complication in algorithms that use a minimal technique.

3.3.1 Vertex marking for simplicial quadtrees

Warren [MW90a] has proposed a technique, *vertex marking*, to produce a triangulation from a balanced simplicial quadtree. The triangulation is a refinement of the balanced quadtree into a *conformal* quadtree. In a general quadtree of d dimensions, each non-leaf node has 2^d children, but in a conformal quadtree, a non-leaf node may have $2^{d'}$ children for any $d' \leq d$ if all children are leaves of

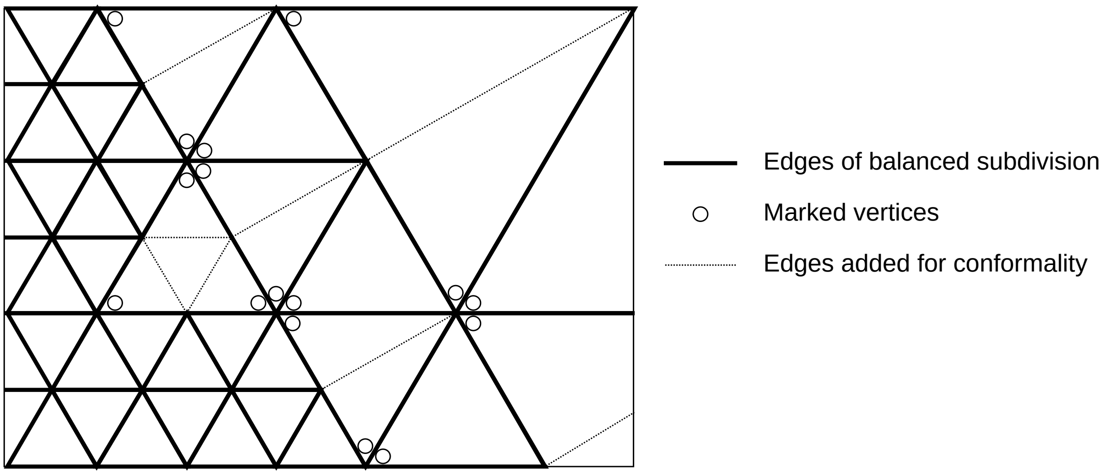


Figure 3.21: Triangulation by vertex marking

the quadtree. A simple procedure, based on symmetric subdivision, produces the conformal quadtree.

Given a balanced quadtree, mark, for each element in the tree, any vertex of the element that coincides with a vertex of a deeper element. If $d'+1$ vertices of an element E are marked, then these $d'+1$ vertices define a d'-dimensional face F of E. Next, subdivide F into $2^{d'}$ subsimplices of dimension d' by restricting the full subdivision of E to F. Finally, decompose E into $2^{d'}$ subsimplices of dimension d by projecting from the $d-d'$ unmarked vertices of E to each of the d'-dimensional subsimplices. Figure 3.21 illustrates an example of this method applied to a balanced triangular mesh. If the quadtree has been smoothed by the algorithm of the previous section, the depths recorded for the vertices provide the same information as marks would. Figure 3.22 shows the effect of vertex marking on the smoothed and balanced quadtrees of figures 3.13 and 3.20, respectively.

Theorem 14 *Given a balanced simplicial quadtree, the vertex marking method produces a triangulation.*

Proof The proof is by induction on the levels of the quadtree. We consider an implementation of the method in which all vertices of elements at the lowest level

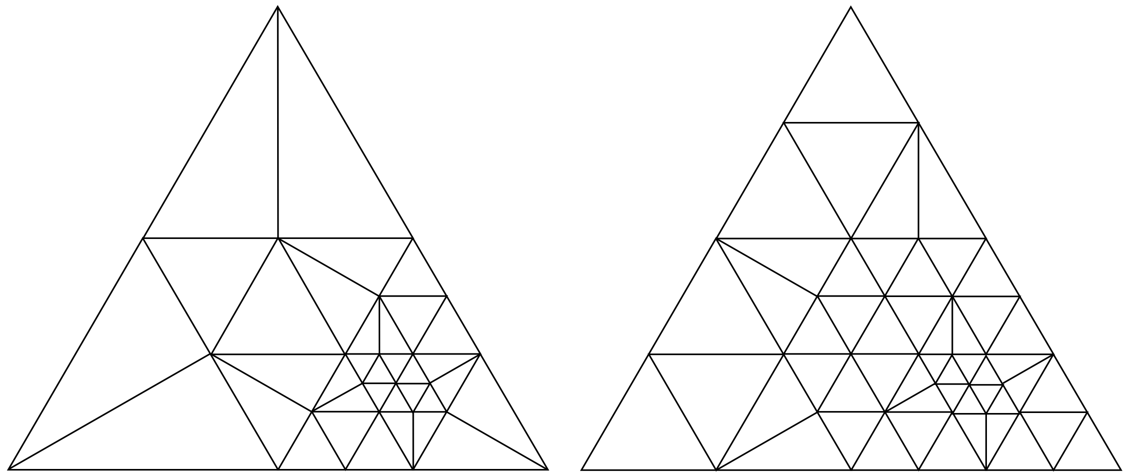


Figure 3.22: Vertex marking of smoothed and balanced trees

of the tree are marked, and elements of the next higher level are split according to those markings, then marked themselves. Initially, the subset of elements at the deepest level forms a triangulation, and all such elements have marked vertices.

Suppose that the elements at depths greater than l form a triangulation, that all vertices of those elements are marked, and that elements at depths up to l form a balanced partition. Let F be a d'-dimensional simplex that is a face of at least one element E at depth l and properly contains d'-dimensional faces of elements in a set $\mathcal{G}$ of simplices at depth $l+1$. Since all the vertices of $G \in \mathcal{G}$ are marked, all the vertices of F are marked as well. Before the elements of $\mathcal{G}$ were created, their parent contained F as a subface. Vertex marking subdivides F into $2^{d'}$ subfaces. Because subdivision produces consistently ordered triangulations, the subdivision of F alone produces the same subfaces as were produced when the elements in $\mathcal{G}$ were created. Thus, the children of E meet the corresponding elements of $\mathcal{G}$ conformally.

Certain other faces may split that do not contain faces of deeper elements. Such a face is common to a collection of unsplit elements at level l of the quadtree. The balance condition on the quadtree ensures that at most one vertex of the face is shared with a more shallow element, so the subdivision of the face and elements

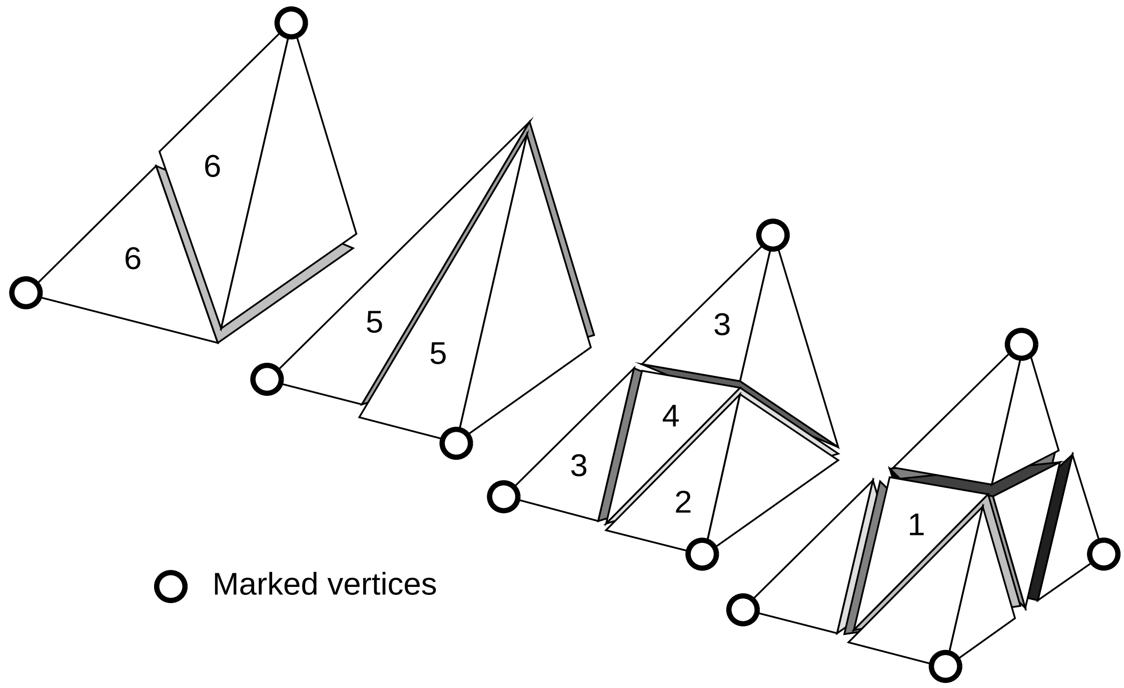


Figure 3.23: Space can be adaptively triangulated with six shapes.

that contain it cannot destroy the balance property for the first $l-1$ levels of the quadtree.

Because none of the simplices at levels 0 to $l-1$ of the partition have been subdivided yet, those levels remain balanced. Thus, after marking all the vertices of all the level $l-1$ elements, the conditions necessary to apply vertex marking to the next level of the quadtree are satisfied and the part already processed forms a triangulation. □

Vertex marking, particularly when applied to R-simplices, produces few distinct shapes for fixed dimension, as figure 3.23 illustrates in three dimensions, and the elements are of generally good aspect ratio. For a symmetrically subdividable simplex, the shortest edge that can result from applying vertex marking to the simplex is no less than half as long as the shortest edge of the simplex.

A shortcoming of vertex marking is that it divides more simplices than necessary to obtain a triangulation. Figure 3.24 illustrates one such example, in which the

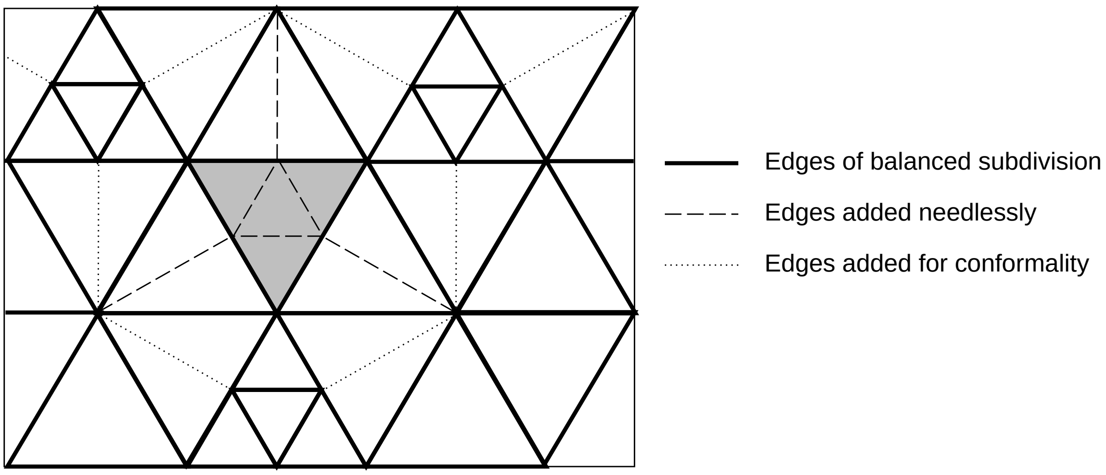


Figure 3.24: Vertex marking can subdivide unnecessarily

shaded triangle and all its neighbors split unnecessarily with vertex marking. To avoid these extra elements, an enhancement of vertex marking can check, for each edge of a particular element, whether its midpoint lies in the set of all element vertices. Both endpoints of the edge are marked and the marked vertices define a face of the element, which splits and forms subelements by projection. A vertex marked for one element is not necessarily marked for another, so some element splitting is avoided. The two-dimensional specialization of this enhancement is the triangulation method described by the "two-neighbor" and "green" rules of Bank et al. [BSW83]. The enhancement avoids the creation of some elements in regions where a few larger elements are surrounded by smaller ones. Its cost is a slight increase in the time required to examine one element, since each edge must be examined, and an examination of more elements, since the splitting of one element can establish conditions that require the splitting of nearby elements.

Chapter 4

Mesh displacement

The meshes produced by the techniques of the previous chapter consist of well-shaped elements within a regular structure. In some applications, such a partition is sufficient. However, an objective of many applications is the separation of space into a small set of classes, and the accuracy predicate is designed to reject elements not distinctly in one class. For such an application, faceted boundaries between sets of elements may not be a satisfactory approximation no matter how much the partition is refined.

The simplest example of such an application comes from graphics and solid modeling — the approximation of the interior and boundary of a smooth solid object by a collection of polyhedral and polygonal elements. A set of rigidly aligned boxes or simplices that approximates a sphere does not produce a visually realistic image of a sphere unless the partition is highly refined. Even then, techniques for simulating natural lighting are likely to produce undesirable effects because the surface normals do not converge to the true normals for a sphere.

This chapter considers the problem of producing, from a mesh of well-shaped elements, another mesh in which some vertices lie on or near the boundary of a smooth solid object. The accuracy predicate in such an application should accept an element of the mesh only if the local approximation to the boundary of the solid is good. If each element of the original partition satisfies such a predicate, then a subset of the facets of the new partition forms a good faceted approximation to the boundary of the solid. If, in addition, the accuracy predicate ensures that the intersection of the boundary of the solid with any k-face of an element is homeomorphic to a $(k-1)$-face, $0 \leq k < d$, and also that each element contains at least one vertex not on the boundary of the solid, then the techniques of this chapter produce a better approximation to the boundary with facets of arbitrary orientation.

4.1 Interpolation in the mesh

The direct approach to the problem is simply to break each element that spans the boundary into two pieces representing the inside and outside. To produce a continuous approximation to the boundary, the choice of how to break each face of an element must depend only on data associated with the face, so that all the elements sharing the face divide it similarly. When the intersection of the boundary with an element is topologically a simplex, *linear interpolation* provides a particularly simple way to break the element, by taking as the approximation the convex hull of the points where edges of the element intersect the boundary.

When the intersection is topologically a general polytope, one approach is to apply an interpolant tailored specifically to that *intersection polytope*. In simplicial meshes, for example, the intersection of the boundary with an element in general position is a simploid, for which a generalization of multilinear interpolation is well-suited. Warren has developed a theory of more general polytopal interpolants from principles of algebraic geometry [War91].

Traditional approaches are simpler; they triangulate the intersection polytope and apply linear interpolation to each resulting simplex to produce a simplicial polytope that approximates the boundary. Variants of the traditional approach have appeared as methods for surface polygonalization in computer graphics [LC87, WMW86,Blo88,HW90] and as mesh generation methods in finite-element analysis. Some of these techniques produce a new mesh of elements of the same kind as in the original mesh. Others, however, produce either a mesh of elements of several different kinds, or produce only the boundary polytope.

A solid is *embedded* in a mesh when a set of facets in the mesh, the *embedded surface*, bounds a polytope and the vertices of the polytope lie on the boundary of the solid. Warren has observed that a generalization of the vertex marking technique of the previous chapter provides one way to refine an adaptive triangulation so that a solid is embedded within it, but the description that has appeared [MW90b] is neither clear nor complete. Figure 4.1 presents Warren's method in a new way.

For every element that spans the boundary of the solid, the method marks each vertex that does not lie precisely on the boundary. Vertex marking produces subsimplices of ordered simplices, where different orders produce different decompositions. For Warren's method, order the vertices so that any vertex within the solid precedes any vertex on its boundary, and in turn any boundary vertex precedes any vertex outside the solid. As shown in chapter 2, this ordering ensures that no subsimplex contains vertices of the parent simplex that lie on opposite sides of the boundary. To split the marked face of an element, divide each of its edges that cross the boundary at the segment-boundary intersection point, rather than at the midpoint, and divide other edges at midpoints. This modified vertex marking method produces no subsimplex that has vertices on opposite sides of the boundary. The subfaces produced with all vertices on the boundary form an embedded surface.

Figure 4.2 shows the result of applying this technique to the two triangulations of figure 3.22. In these examples, almost all vertices are marked, so the figure identifies unmarked vertices with white circles, and dotted lines identify the newly introduced edges.

```
void Quadtree::embed
(const SymSimplex& S, const Solid& obj) {
  VtxSet markSet;
  markBoundaryVtxs(S, obj, markSet);
  refineEmbed(S, obj, markSet);
}

void Quadtree::markBoundaryVtxs
(const SymSimplex& S, const Solid& obj, VtxSet& markSet) {
  if (isLeaf()) {
    int nInside = 0, nOutside = 0;
    for (int j = 0; j < S.nVtxs(); ++j)
      if (! obj.boundaryContains(S.vtx(j)))
        if (obj.contains(S.vtx(j))) nInside++;
        else nOutside++;
    if (nInside == 0 || nOutside == 0) return;
    for (int j = 0; j < S.nVtxs(); ++j)
      if (! obj.boundaryContains(S.vtx(j))) markSet.add(S.vtx(j));
  }
  else
    for (int k = 0; k < nChildren(); ++k)
      child[k].markBoundaryVtxs(S.subElement(k), obj, markSet);
}
```

Figure 4.1: Algorithm for embedding a solid

```
void Quadtree::refineEmbed
(const SymSimplex& S, const Solid& obj, const VtxSet& markSet) {
  if (isLeaf()) {
    bitmask markedVtxs = 0;
    for (int j = 0; j < S.nVtxs(); ++j)
      if (markSet.contains(S.vtx(j))) setbit(markedVtxs, j);
    if (twoBitsOn(markedVtxs))
      createChildrenByInterpolation(markedVtxs, obj);
  }
  else
    for (int k = 0; k < nChildren(); ++k)
      child[k].refineEmbed(S.subElement(k), obj, markSet);
}
```

Figure 4.1: Algorithm for embedding a solid (cont.)

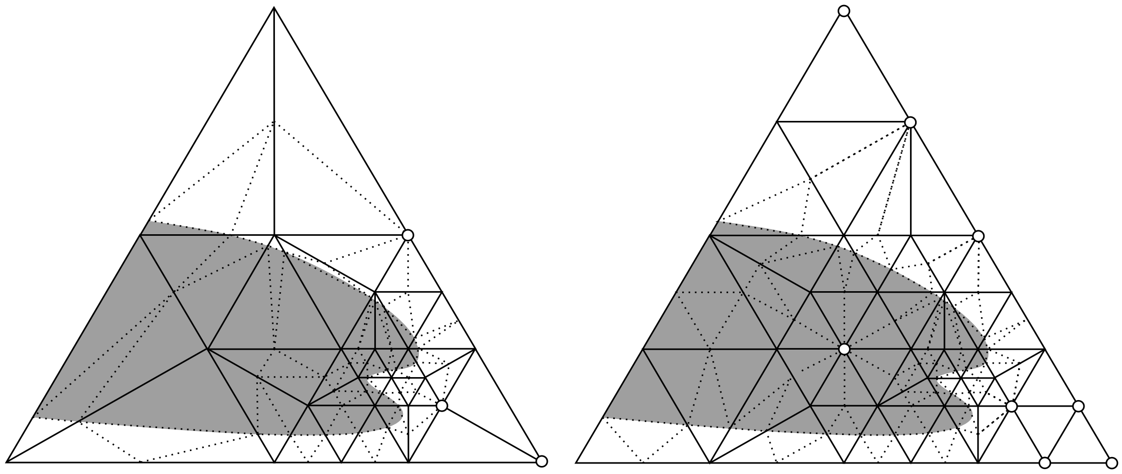


Figure 4.2: Embedding a solid in a triangulation

Clearly, the triangulations produced have many shortcomings. The elements of the mesh can have arbitrarily poor aspect ratios, particularly where the boundary passes near the vertex of an element. This is a problem inherent in all interpolation-based techniques. Moreover, the segments of the embedded curve differ considerably in length, with arbitrarily short edges possible. In three dimensions, the method produces an embedded surface in which such length differences lead to facets with arbitrarily bad aspect ratios. The overall shape quality is made worse because some of the elements in the original mesh are already slightly irregular because of the application of vertex marking to produce the triangulation. Less obviously, the irregular subdivision of irregular elements makes more complex the quadtree data structures underlying the representation of the mesh.

4.2 Displacement of the mesh

4.2.1 Description of the method

According to the accuracy predicate, each element in the original mesh is sufficiently small to provide a good local approximation, so further subdivision of mesh elements should be avoided if possible. An alternative embedding method, *mesh displacement*, requires no subdivision to embed an approximation to the solid in a suitably refined mesh. Instead, the method translates vertices of the mesh to an *interpolating surface*, such as the embedded surface constructed by the method of the previous section or a surface built from higher-order interpolants. Mesh displacement requires a mesh to have certain properties, described below, to avoid several possible kinds of degenerate behavior. The accuracy predicate that produces the initial mesh must ensure that these properties hold.

By avoiding subdivision, mesh displacement preserves the topology of the original mesh. Mesh displacement is related to the "warping" of square quadtrees as used by Bern et al. to provide guaranteed good aspect ratios for two-dimensional triangulations [BEG90], and to techniques discussed by Yerry and Shephard [YS84].

The method has two major phases. The first phase identifies a set of vertices and faces that lie close enough to the interpolating surface to allow "bending" of the mesh toward it without too much distortion of elements. This set of faces corresponds to the embedded surface in the mesh that results. The second phase performs the bending, by displacing each vertex to an appropriate point on the interpolating surface. Figure 4.3 expresses the algorithm.

Beginning with a triangulation, the algorithm examines each element. A set of vertices near the boundary of the solid is identified by examining each edge of the element that crosses the boundary, and including the endpoint nearer to the boundary along that edge. Where the boundary intersects an edge at its midpoint, the algorithm can use the one outside the solid, if known, or the lesser one in some global ordering of mesh vertices. A set of $k+1$ nearer vertices defines a k-face of the *approximating polytope*, the polytope in the original mesh that corresponds to the embedded surface in the final mesh. The accuracy predicate is assumed to reject any element for which this construction is not well-defined, as when an edge intersects the boundary of the solid in more than one point, and to reject an element that would have all its vertices included by this decision procedure.

The point where an edge of an element intersects the boundary is called a *satellite* of the near endpoint of the edge. The collection of satellites for a vertex helps identify the interpolating surface in the region of the vertex. In the second phase of the algorithm, each vertex with satellites is displaced in the mesh, in the direction of the centroid of its satellites. The centroid itself is probably a point near the interpolating surface, but the algorithm displaces the vertex along the ray extending from the vertex through the centroid. The vertex is moved to the first point where that ray intersects the interpolating surface.

Figure 4.4 shows the phases of the algorithm on a simple example. In the first phase, the piece of the interpolating surface within each element is associated with a vertex or an edge of its enclosing triangle. The result is an approximating

```
void Quadtree::displace
(const SymSimplex& S, const Solid& obj) {
  SatelliteMap centroid;
  markNearVtxs(S, obj, centroid);
  displaceVtxs(S, obj, centroid);
}

void Quadtree::markNearVtxs
(const SymSimplex& S, const Solid& obj, SatelliteMap& centroid) {
  if (isLeaf()) {
    for (int j = 0; j < S.nVtxs()-1; ++j) {
      bool containsJ = obj.contains(S.vtx(j));
      for (int k = j; k < S.nVtxs(); ++k)
        if (containsJ ≠ obj.contains(S.vtx(k))) {
          Point mid = midpoint(S.vtx(j), S.vtx(k));
          int nearer = (containsJ == obj.contains(mid))? k: j;
          Point satellite = // 1st pt of intersection along ray
            obj.intersect(S.vtx(j), S.vtx(k));
          centroid[S.vtx(nearer)].averageWith(satellite);
        }
    }
  }
  else
    for (int k = 0; k < nChildren(); ++k)
      child[k].markNearVtxs(S.subElement(k), obj, centroid);
}
```

Figure 4.3: Embedding by mesh displacement

```
void Quadtree::displaceVtxs
(SymSimplex& S, const Solid& obj, SatelliteMap& centroid)
{
  if (isLeaf())
    for (int j = 0; j < S.nVtxs(); ++j) {
      if (centroid.contains(S.vtx(j)))
        S.vtx(j) = obj.intersect(S.vtx(j), centroid[S.vtx(j)]);
    }
  else
    for (int k = 0; k < nChildren(); ++k)
      child[k].displaceVtxs(S.subElement(k), obj, centroid);
}
```

Figure 4.3: Embedding by mesh displacement (cont.)

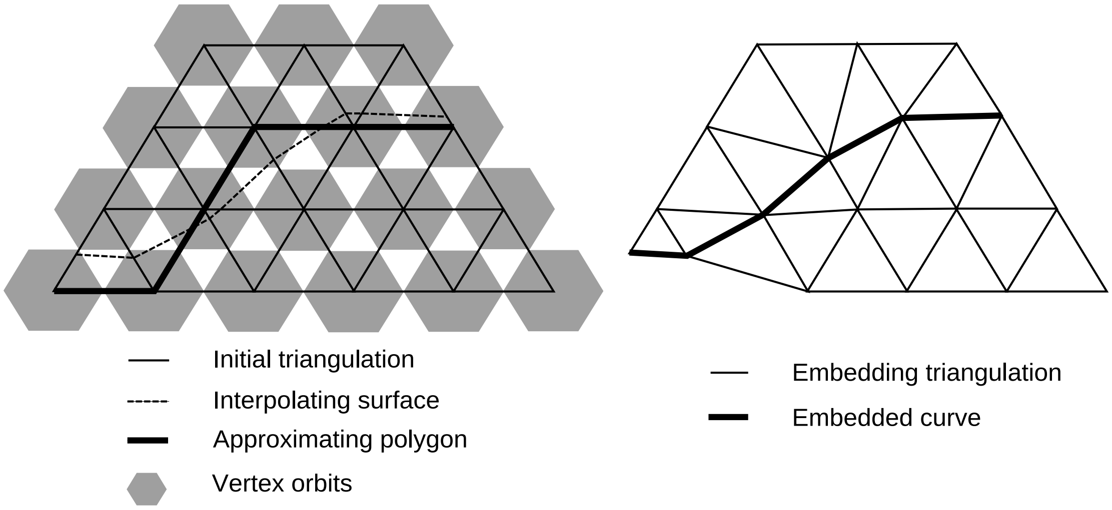


Figure 4.4: Mesh displacement applied to a simple curved boundary

polygon in the edges of the triangulation. In the second phase, the vertices of the approximating polygon move to the interpolating surface, in the direction of their satellites. As a result, the small pieces of the interpolating surface that lie within elements when it passes near a vertex are collapsed to a single vertex in the embedding mesh.

Several examples of surface meshes produced by this method appear at the end of this chapter. Given the same triangulation, mesh displacement produces a much more compact embedded surface than do linear interpolation methods. Mesh displacement also produces triangles in the embedded surface that have much better aspect ratios in practice than those that arise from linear interpolation methods. These advantages arise from the collapse of small and narrow triangles produced by linear interpolation into vertices and edges during mesh displacement.

The shaded hexagons in figure 4.4 are the key to understanding the behavior of mesh displacement. The satellites of a vertex are those vertices of the interpolating surface that lie in the shaded hexagon centered at that vertex. More generally, for a vertex v of the triangulation, let $N(v, 1)$ denote the set of all simplices containing v. $N(v, \delta)$, the size δ *vertex orbit* of v, denotes the same figure magnified by a factor δ

about v. For the mesh displacement algorithm, v is a vertex on the approximating polytope if the interpolating surface intersects the interior of $N(v, 1/2)$ and, rarely, when the linear approximation intersects a vertex of $N(v, 1/2)$.

For a face f, define $N(f, \delta)$ as the convex hull of the size δ vertex orbits of the vertices of f.

Theorem 15 *For two disjoint faces of the initial mesh, f_0 and f_1, the interiors of $N(f_0, 1/2)$ and $N(f_1, 1/2)$ are disjoint.*

Proof For any simplex S, there is a hyperplane that separates a face f_0 from its complementary face in S and passes through each point midway between a point of f_0 and a point of its complement. Consider each simplex S containing f_0; a hyperplane separates the interiors of $N(f_0, 1/2)$ and $N(S - f_0, 1/2)$. The hyperplanes define a polytope that contains $N(f_0, 1/2)$ and no part of any other $N(f_1, 1/2)$. $\square$

Thus, mesh displacement cannot move a vertex through a facet, cause a pair of skew edges to cross, or otherwise change the topology of the mesh.

4.2.2 Initial mesh properties

Several conditions must hold on the initial mesh to guarantee that mesh displacement defines an embedded surface and avoids several possible pitfalls. First, in the consideration of any element, at least one of its vertices must not be taken as belonging to a face of the approximating polytope. In the absence of this restriction, all the vertices can be taken as lying near the boundary, so that a separate d-cell is created that shares a facet with the approximating polytope. A surface defined locally by a nearly flat hyperbolic paraboloid, for example, can cross four edges of a tetrahedron in four different vertex orbits. A single extra subdivision can usually correct the problem.

Suppose, for example, that each vertex of a simplex S has a satellite arising from the consideration of one of its edges. If the boundary is nearly flat over S, the satellites must all lie close to the hyperplane through the midpoints of edges that cross the boundary. Let d denote the distance between supporting hyperplanes parallel to the middle hyperplane. Then if each satellite lies within $d/4$ of the middle hyperplane, a single subdivision of the simplex produces subsimplices in which the boundary intersects no vertex orbit of a vertex inherited from S. Thus, the children of an element that provides a good approximation, but has too many vertices with satellites, do not themselves violate the restriction.

Secondly, within each element, the interpolant must satisfy the *convex hull property*, which means that the part of the interpolating surface within an element must lie within the convex hull of the boundary points that it interpolates. In practice, this restriction can usually be relaxed and the boundary taken as the interpolating surface, so that mesh displacement produces an embedding of the solid. In the absence of the convex hull property, it is possible that the ray from a vertex in the initial mesh through the centroid of its satellites might intersect the interpolating surface outside the orbit of the vertex, which could permit mesh displacement to push a vertex through a facet.

Third, for any k-face f of the mesh, $k < d$, the intersection of $N(f, 1/2)$ with the interpolating surface must be a connected $(d-1)$-manifold. This condition rules out several interpolating surfaces that mesh displacement collapses to something of lower dimension. For example, it precludes a tiny ball around a vertex that might collapse to a point, a narrow sausage-shaped surface around an edge that might collapse to the edge, and so on.

To summarize, these three essential conditions are

- Each element must have a vertex with no satellites.

- The intersection of the boundary with an element must lie within the convex hull of the points where element edges intersect the boundary.

- The boundary must intersect the neighborhood of any face in a topological disk of appropriate dimension.

The following theorem justifies the use of the term "approximating polytope" in the description of the algorithm. It shows that the faces of the approximating polytope around any of its vertices combine to form something topologically equivalent to a hyperplane in the region of the vertex, under the conditions on the mesh given above.

Theorem 16 *If v is a vertex of the approximating polytope, then the union of the faces of the approximating polytope in the interior of $N(v, 1)$ is a connected $(d-1)$-manifold.*

Proof Consider some k-face f_k, $0 \leq k < d-1$, of the approximating polytope as defined by the algorithm. The initial conditions on the mesh require that in the region $N(f_k, 1/2)$, the interpolating surface is a connected $(d-1)$-manifold. Since the interpolating surface passes outside $N(f_k, 1/2)$, there must be some vertex w that is adjacent to an element of f_k and has a satellite. If the only such w lies in the same k-dimensional hyperplane as f_k, then let f_{k-1} denote the k vertices of f_k adjacent to w. Over $N(f_{k-1}, 1/2)$, the interpolating surface has two holes, which violates initial mesh conditions. Thus, there must exist a w affinely independent from f_k that has a satellite. By induction, a single vertex that has a satellite must be part of some $(d-1)$-face, or facet, of the approximating polytope.

The initial mesh conditions do not allow a single isolated facet, so any facet must share a vertex with some other facet. By the initial mesh conditions, the orbit of the shared face of the two facets intersects the interpolating surface in a surface that is a connected $(d-1)$-manifold. That implies that for any vertex of the shared face, the faces that share it combine to form a connected $(d-1)$-manifold.

So, starting with a vertex v of the approximating polytope, it follows that v is part of a facet of the approximating polytope. At least one vertex of the facet is shared by two facets, so it must be surrounded by facets to form, locally, a warped

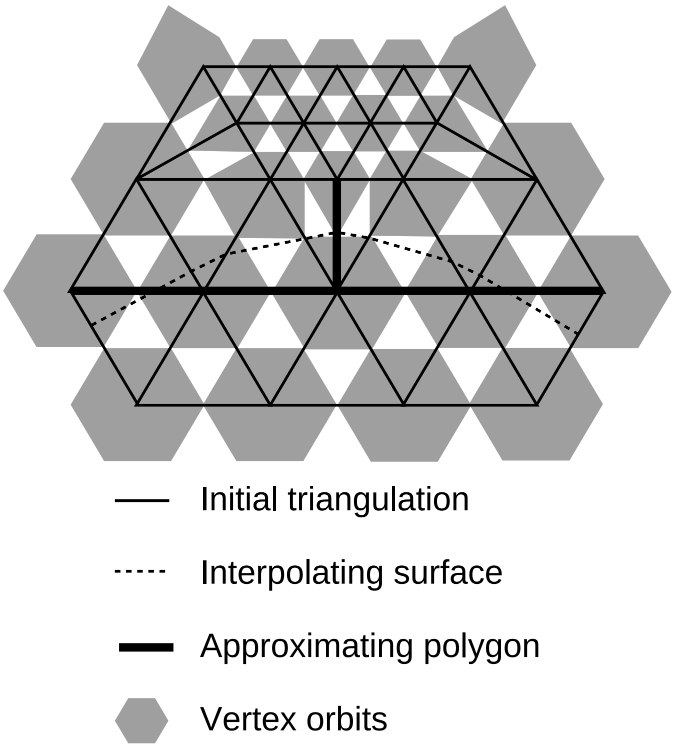


Figure 4.5: Non-manifold topology arising during mesh displacement

$(d-1)$-ball. Since one of those surrounding facets contains v, then v also must be surrounded by facets. $\square$

A more subtle topological fault ruled out by the condition above might occur when an almost flat interpolating surface intersects $N(v, 1/2)$ in two connected components. The separate parts of the surface would collapse together and produce incorrect topology. Figure 4.5 illustrates one such example.

It is possible to make simple local adjustments to the mesh to circumvent the problem. A subset of the mesh facets that contain v forms a connected $(d-1)$-manifold that separates the two components of the interpolating surface within $N(v, 1/2)$. Replace v by two coincident vertices v_0 and v_1, and each of the separating facets f by a simplex, the convex hull of v_0, v_1, and the vertices of f, excluding v. Any other mesh vertex adjacent to v is considered adjacent to v_0 or v_1 according to which side of the separating facets it lies on. With this construction, the

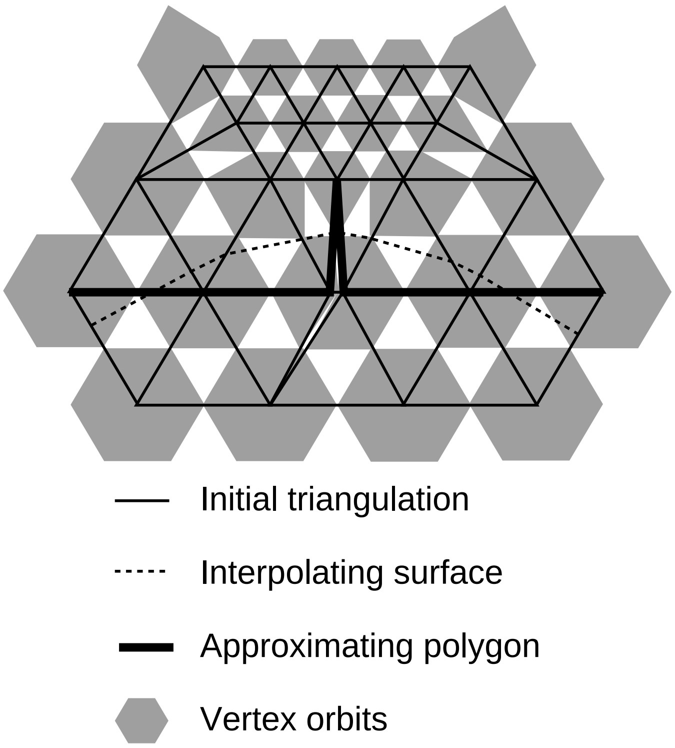


Figure 4.6: A mesh can be adjusted to avoid non-manifold topology

two copies of v are able to displace to different components and the degenerate simplices are no longer degenerate after displacement. Figure 4.6 illustrates the application of this adjustment to the mesh of figure 4.5.

The adjustment was also necessary in the application of mesh displacement to the running examples of this and the previous chapter. Figure 4.7 shows that in the lower right corner of the mesh, more refinement is necessary to satisfy the conditions for mesh displacement. Although the correction presented above can correct the problem and lead to a proper embedding, it does not provide bounds on element aspect ratio.

Conjecture [Warren] In a uniform mesh of P-tetrahedra, mesh displacement produces an embedding in which no face of the embedding polyhedron has aspect ratio greater than $\frac{11(3+\sqrt{11})}{24} = 2.895\ldots$. The conjecture comes from an enumeration of cases.

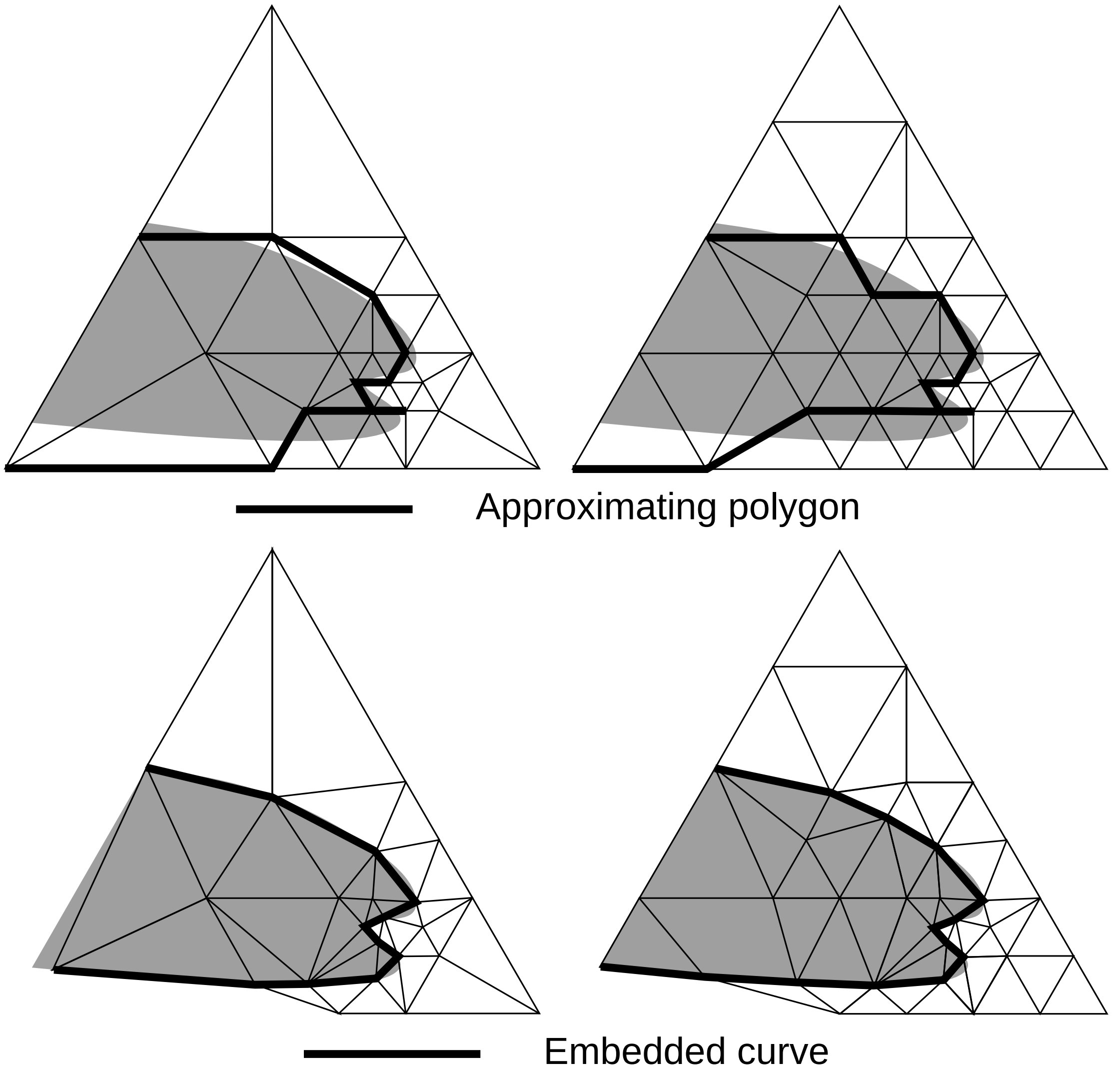


Figure 4.7: Two examples of mesh displacement

For obtaining an approximate triangulation of the boundary of the solid, mesh displacement produces reasonably well-shaped elements. A difficulty arises in creating a triangulation of the interior of the embedded polytope, however. An interpolating surface, even one that is locally a hyperplane and satisfies all initial mesh conditions, can intersect the vertex orbits of all the vertices of a simplex. After displacement, all the vertices of the simplex lie on the interpolating surface. As a result, the nearly flat simplex may have a poor aspect ratio. The next section describes a method that guarantees a final triangulation whose elements have bounded aspect ratios, and that imposes less strict conditions on the mesh than does mesh displacement.

4.3 A hybrid method

The previous section identifies two potential problems with mesh displacement. First, the mesh must satisfy some subtle conditions for mesh displacement to produce a topologically correct embedding surface. Second, the interpolating surface can intersect the vertex orbits of all vertices of a simplex. This section presents a method developed jointly with Joe Warren that avoids these two problems for locally flat interpolating surfaces.

An interpolating surface is *locally ϵ-flat* if, for every vertex v of the initial triangulation, each satellite within $N(v,1)$ lies within ϵ of some hyperplane. Often, the application that generates the initial triangulation is capable of generating an ϵ-flat mesh for any specified value of ϵ. In the approximation of algebraic sets with smooth boundaries, for example, Warren has presented an algorithm based on the Bernstein basis representation of polynomials that generates ϵ-flat surface meshes for user-specified ϵ [HW90].

For any positive ϵ, mesh displacement may produce a non-manifold approximating polytope for a locally ϵ-flat interpolating surface mesh. However, if the original interpolating surface is a manifold that is locally ϵ-flat for sufficiently small

ϵ, the method of this section, developed with Warren, produces an embedding surface that is a manifold. Moreover, the volume mesh produced by the method has elements of bounded aspect ratio.

The observation that leads to the derivation of the algorithm is that the vertex orbits of size 1/2 touch at the midpoints of edges in the initial triangulation. If the interpolating surface is a hyperplane, it can touch all orbits of the vertices of a simplex simultaneously. However, if all vertex orbits are smaller, of size 1/3 for example, then the interpolating hyperplane cannot touch all vertex orbits simultaneously within the simplex. With such small vertex orbits, it is possible for the interpolating surface to cross an edge but miss the orbits of its endpoints. To allow the surface mesh to cross the interior of the edge, the method splits the edge using Warren's vertex marking algorithm. After all such crossings are split, mesh displacement moves the remaining vertices to the interpolating surface.

To apply this hybrid technique, begin by examining every simplex, and within that simplex every edge. Classify a vertex of the simplex as lying near the boundary if it has a satellite within its size δ vertex orbit, and as inside or outside the solid otherwise. The corresponding step in Warren's method classifies vertices as strictly on the boundary of the solid, inside or outside it. Order the vertices of the simplex so that all vertices inside the solid are numbered before those near the boundary, and the vertices outside the solid are numbered last. Mark a vertex if it is far from the boundary and it is adjacent to a vertex across and far from the boundary. Use vertex marking to split the simplex. Cut each marked boundary-spanning edge where it crosses the boundary of the interpolating surface, and cut other marked edges at midpoints.

Displace the near-boundary vertices to the boundary, by mesh displacement as described previously. That is, find the centroid of the satellites of such a near-boundary vertex and move the vertex in the direction of the centroid until it meets the boundary.

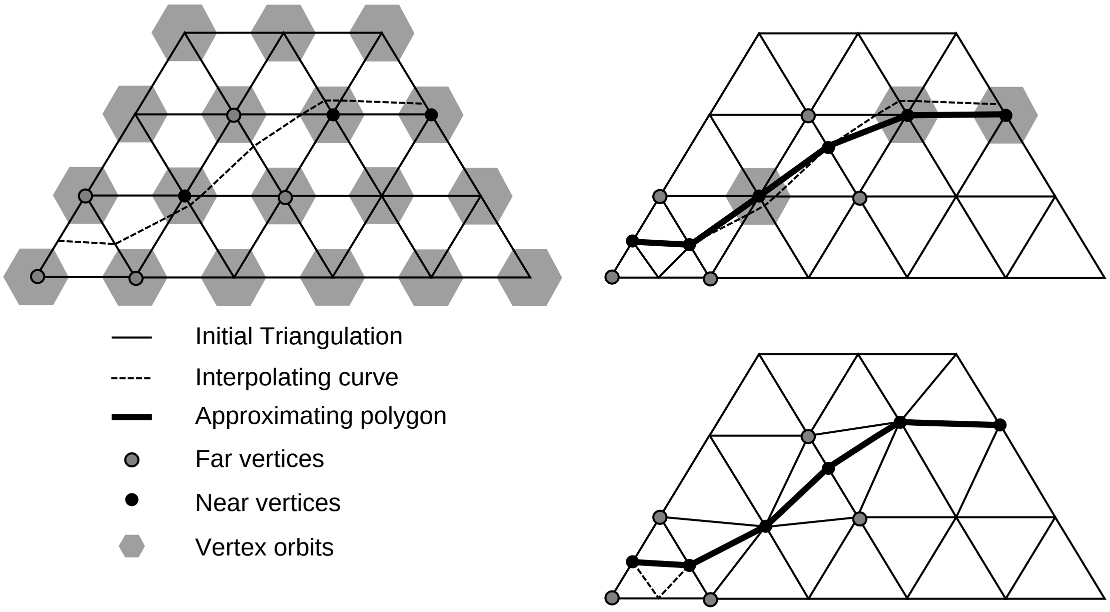


Figure 4.8: The hybrid method applied to an embedded curve

The resulting mesh is guaranteed to be a triangulation by the properties of the vertex marking algorithm. Figure 4.8 shows the algorithm applied to a two-dimensional example for $\delta = 1/3$. The upper lefthand portion shows the embedded curve and its triangulation after the classification of vertices. The upper righthand portion shows the triangulation after the application of vertex marking. Finally, the lower righthand portion shows that triangulation and resulting contour after displacement of the vertices near the surface.

This algorithm combines aspects of both Warren's triangulation method and mesh displacement. For $\delta = 0$, the vertex classification step identifies as near the surface only those vertices on the surface, so that the algorithm is Warren's algorithm. For $\delta = 1/2$, the vertex orbits for two adjacent vertices cover their entire common edge, so this hybrid technique produces the same triangulation as does mesh displacement.

The next theorem demonstrates that, for well-behaved meshes and some $\delta > 0$, this method produces a surface mesh that approximates a smooth curved surface.

Theorem 17 *Given a triangulation of P-tetrahedral elements with edge lengths* 1 *and* $\sqrt{3}/2$ *and, within it, a locally* ϵ*-flat interpolating surface that intersects* $N(v, 1)$ *in a set homeomorphic to a disk, for any vertex* v *of the triangulation, the surface embedding produced by the method above is a manifold for all* $0 \leq \delta \leq 1/4 - \epsilon$.

Proof Consider, for each vertex v of the mesh that results from application of the algorithm, the intersection of the embedded surface with the interior of $N(v, 1)$. If v is a vertex far from the boundary, the embedded surface does not intersect the interior of $N(v, 1)$.

Suppose that v is near the boundary before displacement. The 14 vertices of $N(v, 1)$ form a rhombic dodecahedron in which each rhombic face is treated as a pair of triangular faces. A number of marked vertices may appear on the exterior of $N(v, 1)$, and where a pair of adjacent marked vertices on opposite sides of the boundary appear, vertex marking is applied to create even more triangular faces. This proof is concerned with the resulting figure, which is topologically a polyhedron with 24 or more triangular faces, some pairs of them coplanar.

Some of the vertices of the polyhedron are near the boundary, and some others, created by vertex marking, are on it exactly. If these boundary vertices form a simple chordless cycle of length greater than three on the exterior of the polyhedron, where any pair of adjacent boundary vertices is connected by an edge, then each pair on the exterior forms, with v, an element of the embedded surface, and the embedded surface is locally a manifold because the cycle forms a simple closed curve. The object of the proof is to demonstrate that under the assumptions given above, the boundary vertices necessarily form a chordless cycle.

Because v is near-boundary, v and all the boundary vertices on the exterior of the polyhedron lie between a pair of parallel planes less than $1/2$ apart. A three-cycle of boundary vertices can therefore not appear on the exterior of the polyhedron, because, with v, those vertices are the vertices of a P-tetrahedron, which cannot lie between parallel planes less than $1/2$ apart.

A boundary vertex created by vertex marking must be adjacent to exactly two others, because the rules that split triangles and create new boundary vertices so specify.

A boundary vertex w chosen from the original 14 vertices must be adjacent to at least two others by the topological assumptions made on the initial mesh. Any of the several ways a vertex from the original 14 could be adjacent to three or more others requires either that the boundary be two-sheeted over the interior of the polyhedron, or that the boundary not lie between planes 1/2 apart.

Thus, every boundary vertex on the exterior of the polyhedron is adjacent to two others and there can be no three-cycles. For there to be two cycles on the exterior of the polygon requires that the boundary be locally two-sheeted, which is forbidden by the initial topological conditions on the mesh. □

Additionally, the aspect ratios of the triangles and tetrahedra in this triangulation are bounded above.

Theorem 18 *For a locally* (0.01)*-flat interpolating surface that is embedded in a P-tetrahedral triangulation, the hybrid method with* $\delta = 0.24$ *produces an embedding triangulation whose tetrahedra have a worst aspect ratio of* 6.71 *and whose triangles have a worst aspect ratio of* 2.48*.*

Proof The basis of the proof is an examination of the possible outcomes during the splitting of a tetrahedron in the application of vertex marking. In this examination of outcomes, a vertex is considered near or far from the boundary based on an examination of all edges incident to it, not just the ones in a particular tetrahedron; it is near to the boundary if it is near along any of its incident edges.

1. The tetrahedron has no vertices near the boundary, and a pair of vertices on opposite sides of the boundary, so it splits into eight tetrahedra. The three or four new vertices are introduced on the boundary, and they may lie anywhere along the middle portion of their edges.

2. The tetrahedron has one vertex near the boundary, and other vertices on opposite sites of the boundary, so it splits into four tetrahedra. Two new vertices are introduced on the boundary, and they may lie anywhere along the middle portion of their edges. The vertex near the boundary may be displaced anywhere in its size δ vertex orbit.

3. The tetrahedron has two vertices near the boundary, and the other two are on opposite sides of the boundary, so it splits into two tetrahedra. A new vertex is introduced on the boundary, and may lie anywhere along the middle portion of its edge. The two near-boundary vertices may be displaced anywhere in their size δ vertex orbits.

4. The tetrahedron has no pair of vertices distant from the boundary and on opposite sides of it. The tetrahedron does not split. Up to three vertices may be displaced within their size δ vertex orbits.

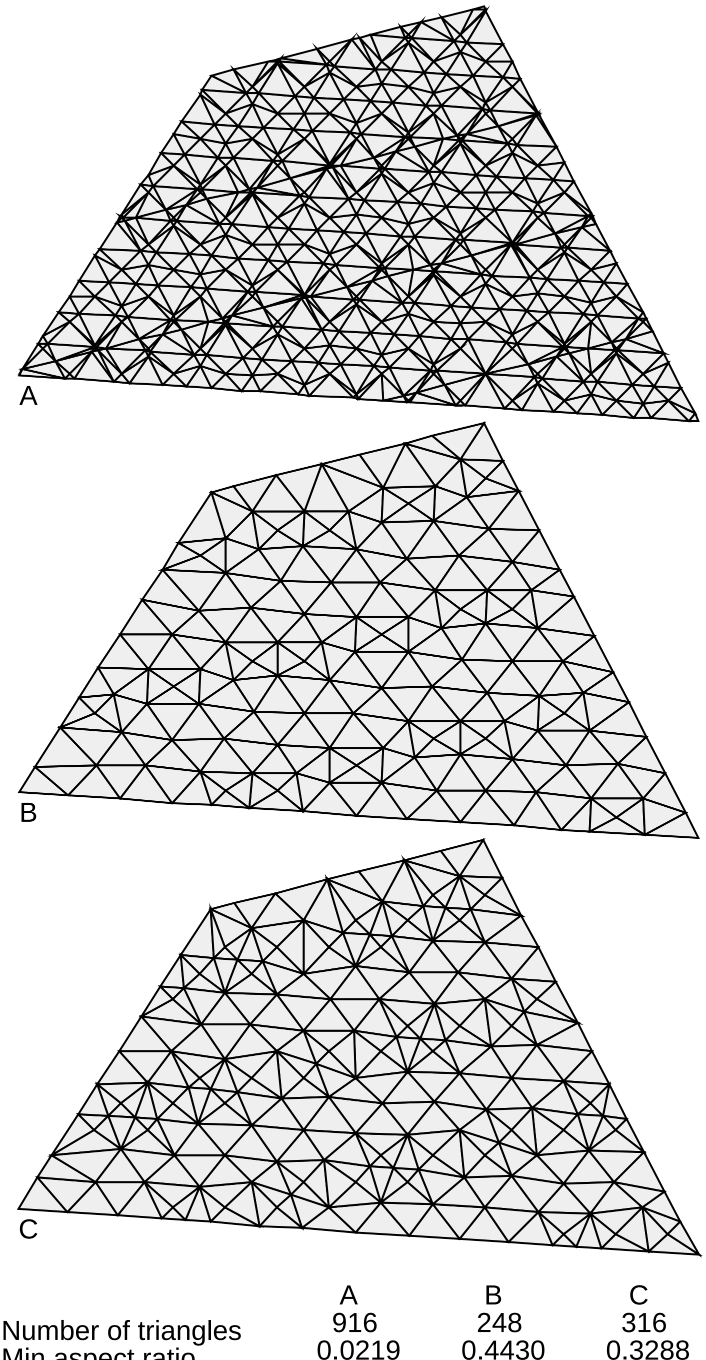
The worst possible aspect for tetrahedra produced by this method is 6.71 and arises in the second case listed above. The minimum possible aspect ratio for triangles produced by this method is 2.48 and arises in the first case. These results come from a computer program that computes worst aspect ratios for each of the cases, by placing vertices at the extreme points of their domains in all possible ways. Some of these configurations may not arise in practice. Thus, tighter lower bounds may be possible. □

4.4 Experimental Results

These embedding methods have been applied to several simple algebraic sets, examples of which appear below. Each figure contains three meshes. All meshes begin with the same triangulation. Mesh A is the original interpolating surface constructed by Warren's method. Mesh B is the product of mesh displacement applied to the interpolating surface in mesh A. Mesh C is mesh A after applying the hybrid method with $\delta = 1/3$. The statistics accompanying each figure reflect the average value and deviation of the reciprocal aspect ratios of each triangle, because averages involving the usual aspect ratio are heavily skewed by the presence of a few nearly degenerate triangles.

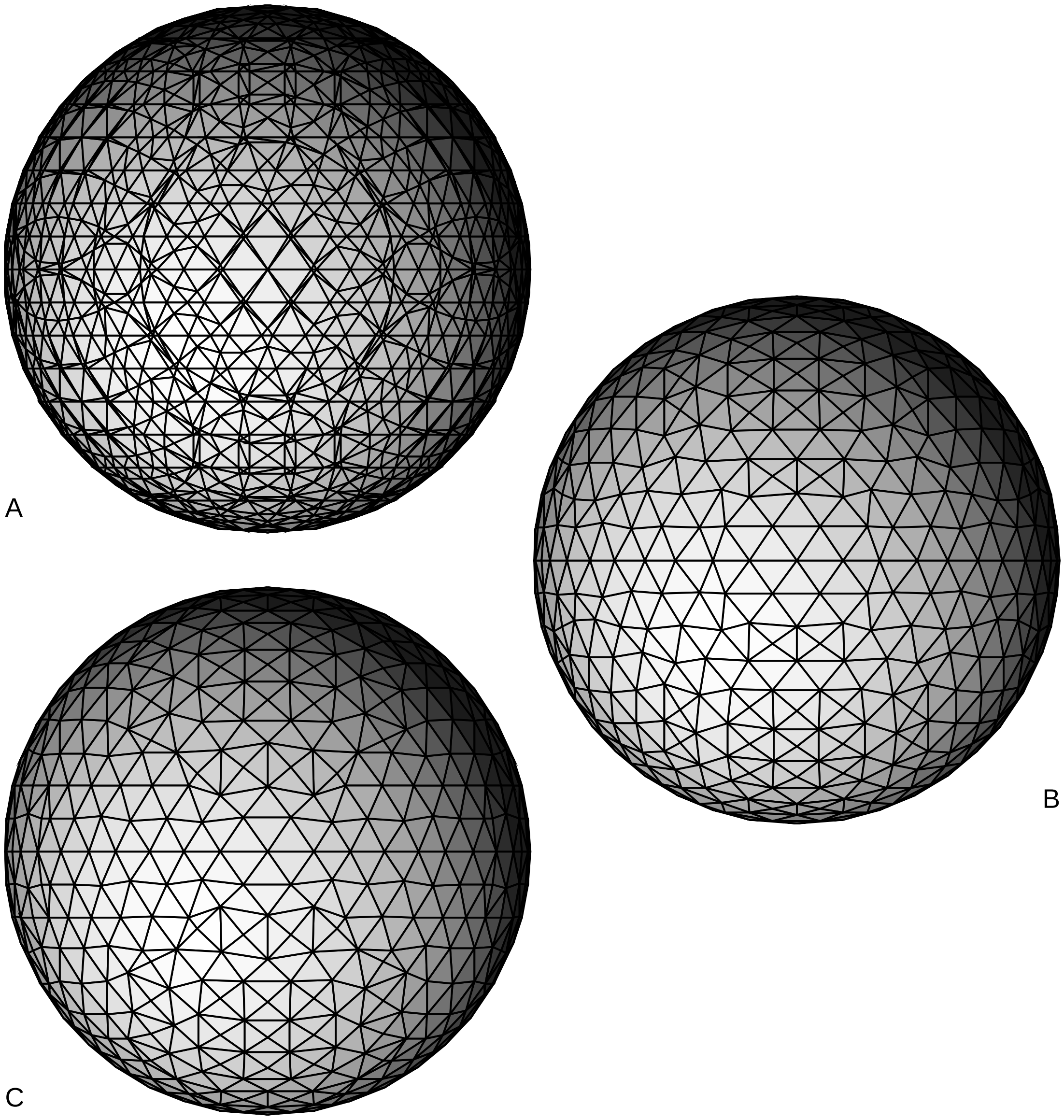
Figure 4.9 shows a plane passing through a triangulation consisting of P-tetrahedra. The triangulation has been refined several times to illustrate the irregularity of the resulting embedded surface mesh. Figure 4.10 is a sphere passing through a similar mesh. Figure 4.11 illustrates a hyperboloid of one sheet. The triangulation enclosing the hyperboloid is nonuniform, so some of its elements are not P-tetrahedra. Figure 4.12 is an elbow-shaped composite surface consisting of a pair of cylinders and cubic surface.

On average, mesh displacement appears to compress a typical interpolating surface mesh produced by Warren's method by a factor of three. Experience with the implementation has shown that the minimum and average aspect ratios of triangles in the final surface mesh tend to improve as δ is increased. This observation suggests that if only surface meshes are desired, using straightforward mesh displacement is the best course. In the examples, the worst aspect ratios for the surface meshes produced by the hybrid method do not always satisfy the bound of theorem 18. For the plane and the elbow, some elements near the boundary of these surfaces have been further distorted to conform to the planar boundaries imposed by the surface definitions; unmodified, mesh displacement might not produce a clean "cut" at the end of the cylinder. For the sphere, there were no boundary triangles and the bound was satisfied.



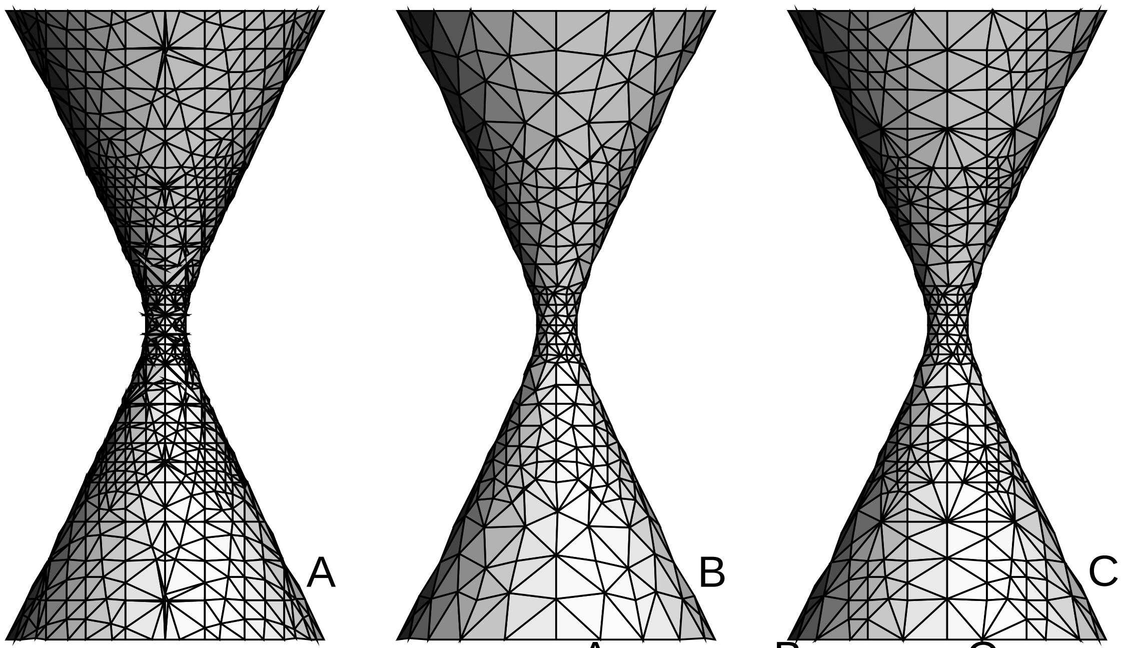
	A	B	C
Number of triangles	916	248	316
Min aspect ratio	0.0219	0.4430	0.3288
Avg aspect ratio	0.7300	0.9060	0.8399
Deviation in ratios	0.2603	0.1124	0.1194

Figure 4.9: Surface triangulation of planar quadrilateral



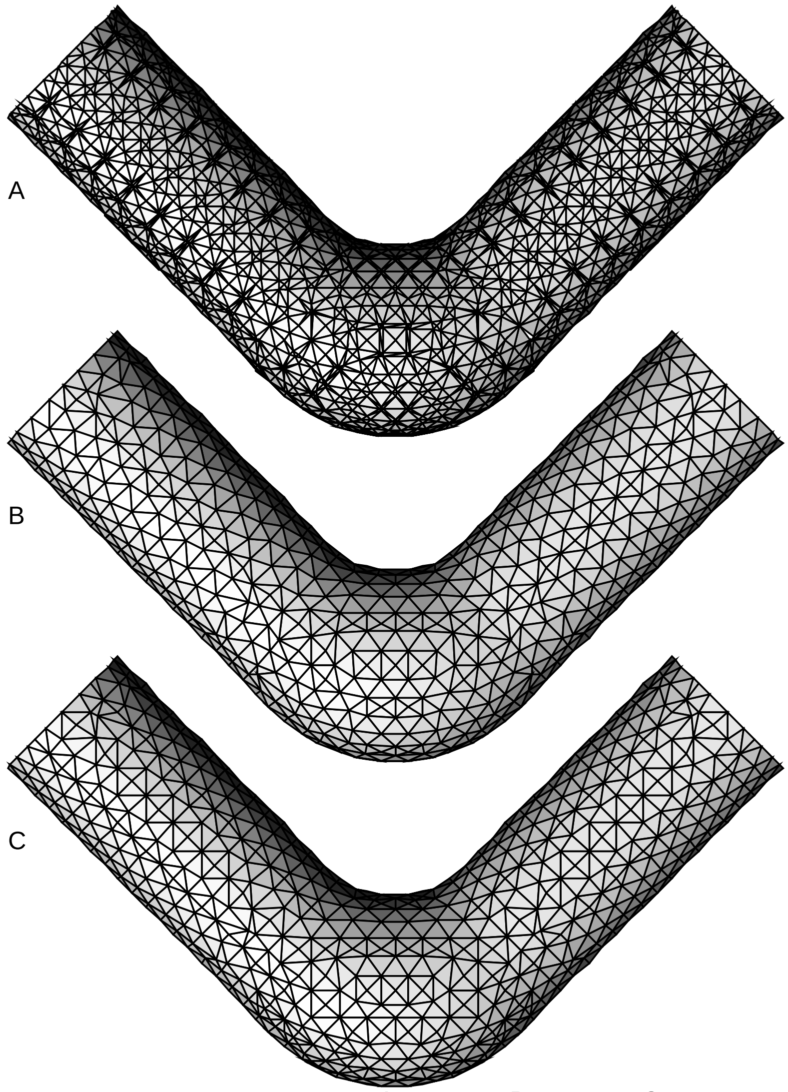
	A	B	C
Number of triangles	2424	696	744
Min aspect ratio	0.1541	0.6580	0.5408
Avg aspect ratio	0.7520	0.9216	0.8976
Deviation in ratios	0.2338	0.0873	0.1038

Figure 4.10: Surface triangulation of a sphere



	A	B	C
Number of triangles	3172	984	1296
Min aspect ratio	0.0054	0.4796	0.3591
Avg aspect ratio	0.7494	0.8781	0.8321
Deviation in ratios	0.2568	0.1399	0.1492

Figure 4.11: Surface triangulation of a hyperboloid



	A	B	C
Number of triangles	6922	1992	2240
Min aspect ratio	0.0032	0.4665	0.4554
Avg aspect ratio	0.6619	0.9271	0.8938
Deviation in ratios	0.2753	0.0948	0.1199

Figure 4.12: Surface triangulation of an elbow-shaped surface

Chapter 5

Displacement for complex objects

The previous chapter deals only with smooth objects. More difficult, and more practical, problems arise when objects are not smooth, but instead have boundaries with creases, cusps, or self-intersections. The techniques of the previous chapter can fail for such objects. This chapter considers two problems involving complex solids and discusses modifications to mesh displacement that let it provide adaptive meshes, often of well-shaped elements, that fill the interiors of the solids.

5.1 Constructive solid geometry

A principal technique for solid modeling is *constructive solid geometry (CSG)* [RV82,RV83]. A CSG system provides a small collection of primitive objects and represents more complex objects as *regularized* Boolean combinations of primitive objects. The regularized Boolean operators are union, intersection, difference and the like, slightly modified to avoid creating topological oddities like dangling edges and isolated points. Popular primitive objects in CSG systems include halfspaces, natural quadric surfaces and tori.

An appropriate set of smooth solid objects can provide the basis for a solid modeling system. A system whose primitives include all objects bounded by algebraic sets, $F(x, y, z) = 0$ for some polynomial F, is enormously flexible. Such a system provides an easy method for determining whether a point lies in a particular solid, by simply evaluating several polynomials at the point and checking the signs of the values that result. However, two fundamental problems are difficult to solve in such a representation. The question of whether two objects intersect requires advanced and computationally difficult techniques involving Gröbner bases [Buc85] or polynomial resultants [Col71]. Generating the boundary of a solid so defined, whether for graphical rendering or for some other purpose, is difficult because only rarely does an algebraic set have a parameterization to help identify points on the boundary of the solid.

By abandoning exactness and adapting the approximation techniques from the previous chapter, this section outlines a CSG system based on approximations to smooth solids. The proposed system creates objects in three phases. The first phase uses techniques from previous chapters to build a sufficiently accurate simplicial quadtree approximation to a smooth object, which can be bounded by an algebraic set or can arise from some other source. The second phase performs Boolean operations on pairs of quadtree representations to produce, ultimately, a quadtree representation of a complex object. The final phase displaces the vertices of the quadtree representation to make them lie more nearly on the object boundary, to improve the shape of boundary and interior simplices.

5.1.1 Building displaceable quadtrees

The initial processing of an object to produce a quadtree proceeds here as it has in previous chapters. Besides recording the structure of the quadtree, however, the initial processing must also produce data, attached to the quadtree, to help guide the later displacement of the mesh. For smooth solids bounded by algebraic sets, the additional information includes

- for each vertex, the function value at the vertex, to help identify the vertices near the boundary of the solid;

- for each vertex, the location of a nearby point on the boundary to which the vertex may be displaced;
- for each element, the number of solids whose boundaries intersect the element, which is necessarily zero or one; and,
- for each element, a one or two element truth table that identifies the element as inside or outside the solid, or neither.

The function value and steepest descent direction at a vertex, not the centroid of its satellites, determine the location of the nearby point. Because the quadtree may not form a triangulation, but only a sufficiently accurate piecewise approximation, a vertex near the boundary that may ultimately be displaced to the boundary can have no satellites yet. In such a case, as in the example of figure 5.1, an alternative to selecting the centroid of satellites is necessary. When data other than algebraic formulas define the smooth solid, a different technique must compute a nearby point from that data.

For a single smooth, primitive solid, the function values at the vertices of an element determine the number of solids that intersect the element, either one or none, and the truth table for the element. Explicit storage of these values is not strictly necessary. However, the combination of several smooth solids at an element requires the storage of such information, and the simple information computed explicitly for a single solid is the degenerate case.

5.1.2 Merging quadtrees

For two displaceable quadtrees defined over the same initial simplex, any CSG operation requires a single traversal, in parallel, of the two quadtrees. When the traversal reaches internal nodes of each quadtree, the merging algorithm proceeds to traverse the subtrees of the internal nodes. If the traversal reaches an internal node of one quadtree as it reaches a leaf of the other, the merging algorithm splits

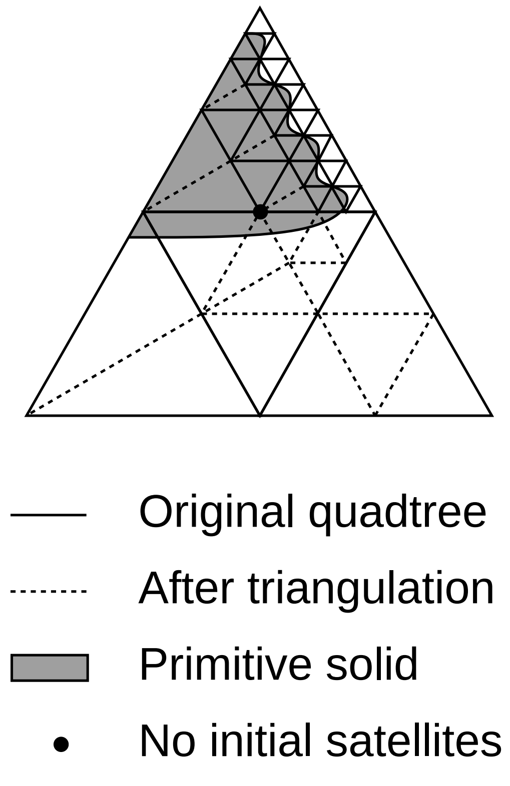


Figure 5.1: A vertex can gain satellites only after triangulation

the leaf to produce an internal node and its several descendants, then merges the corresponding internal nodes. When the traversal reaches two leaves at once, the algorithm constructs a representation of all the boundary surfaces crossing that leaf and a table to record those parts that are inside and outside the solid. Each of these cases presents some opportunity for simplification of the resulting merged quadtree.

There are two easy simplifications available when merging two leaf nodes. First, the combination of the two leaves may not depend on both of them because one leaf is homogeneous, either completely inside or outside the object, and the Boolean combination of the homogeneous leaf with the other may be identical to one of them. In the intersection operation, for example, a single leaf entirely outside the object makes the other leaf irrelevant. Second, the Boolean operation may be one that does not depend on one of its arguments. In each case, only one leaf matters. When both leaves matter, though, their representations are merged, with lists of function values and nearby points concatenated at every vertex, and the sizes of the truth tables multiplied together to form a new truth table. Figure 5.2 depicts the intersection of two leaf elements.

A third simplification is possible when functions defined over the two leaves interact in particular ways, as when no point lies in the positive-valued region of a pair of them, or when the positive-valued region of one is a subset of the positive-valued region of another. That type of simplification requires the determination of whether certain systems of linear inequalities are feasible. There are effective algorithms for making that determination. It is not clear, however, that this type of simplification is prevalent enough to justify the cost of checking for its possibility with every merger of two leaves. Therefore, that class of simplifications has been ignored in programs based on these ideas.

The merging of two internal nodes can provide an opportunity for simplification as well, when all the children of the merged node are homogeneous and identical.

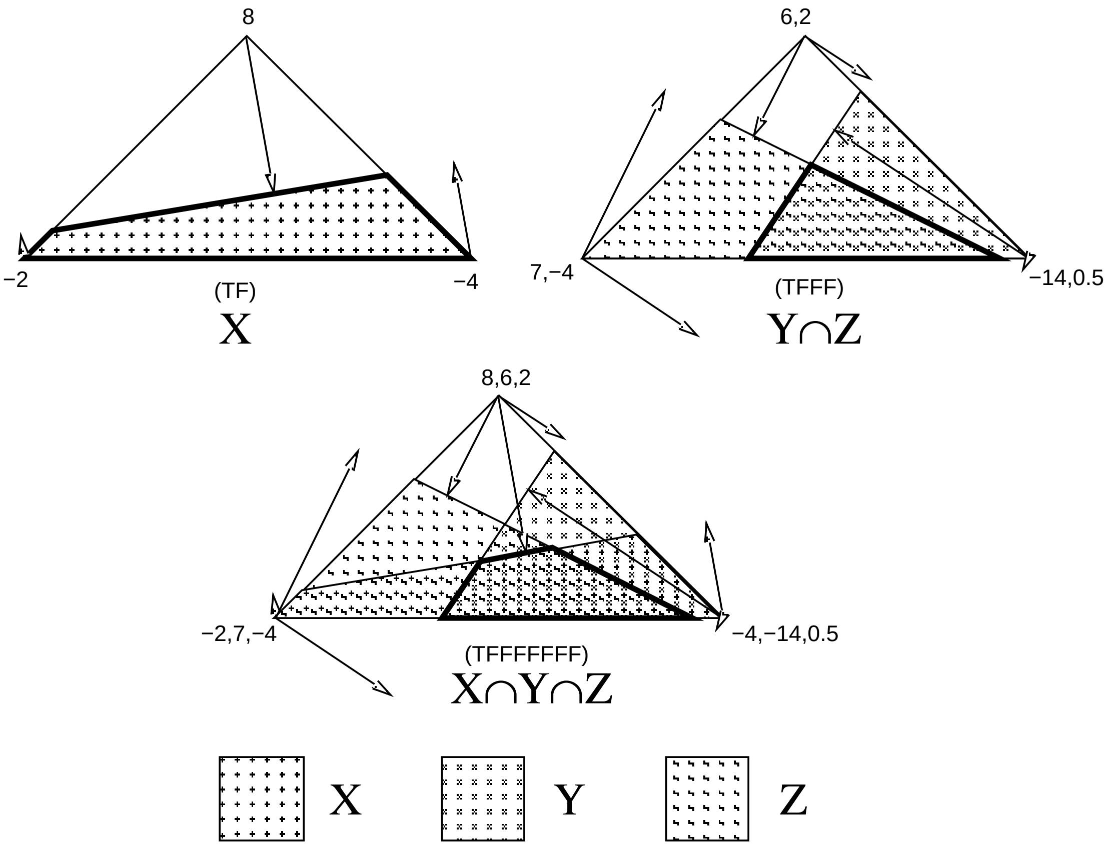


Figure 5.2: Intersecting two quadtree leaves

For example, the intersection of two disjoint objects with complex boundaries is the empty object, and should be represented as a single leaf. In the traversal of the children of an internal node, the traversal has associated with it a state, which is either empty, when no children have yet been visited, inside or outside, when all the children visited to that point are homogeneous and identical, or mixed, when either a mixed child or homogeneous children of different types have been visited. If the state at the end of the traversal is not mixed, a single node replaces the node and its children.

The splitting of a leaf, necessary when merging a leaf with an internal node, provides opportunities for simplification in general, and especially because inequality-based simplifications are not performed when merging leaves. The procedure that splits a leaf must compute, at the midpoints of the edges of the leaf, the values of the several functions defined over the leaf. Often, a particular child inherits many functions entirely positive or negative over the region it defines, and information about these functions can be discarded. Sometimes, as in the intersection of disjoint regions depicted in 5.3, a function is single-signed over a child, which allows all function data for the child to be discarded.

5.1.3 Displacing Merged Quadtrees

Producing a triangulation from a displaceable quadtree is straightforward using techniques already described. Mesh displacement, as described in the previous chapter, cannot be applied as easily. It is important not only to ensure that faces of the triangulation are displaced to coincide with surfaces, but also to ensure that edges and vertices of the triangulation are displaced to coincide with intersections of two and three object boundaries. Failure to consider such intersections specially can result in distortion or topological oddities in the displaced triangulation.

For example, in the triangulation depicted in figure 5.4, each of two solids has a boundary that crosses triangle edges near their midpoints. Depending on exactly

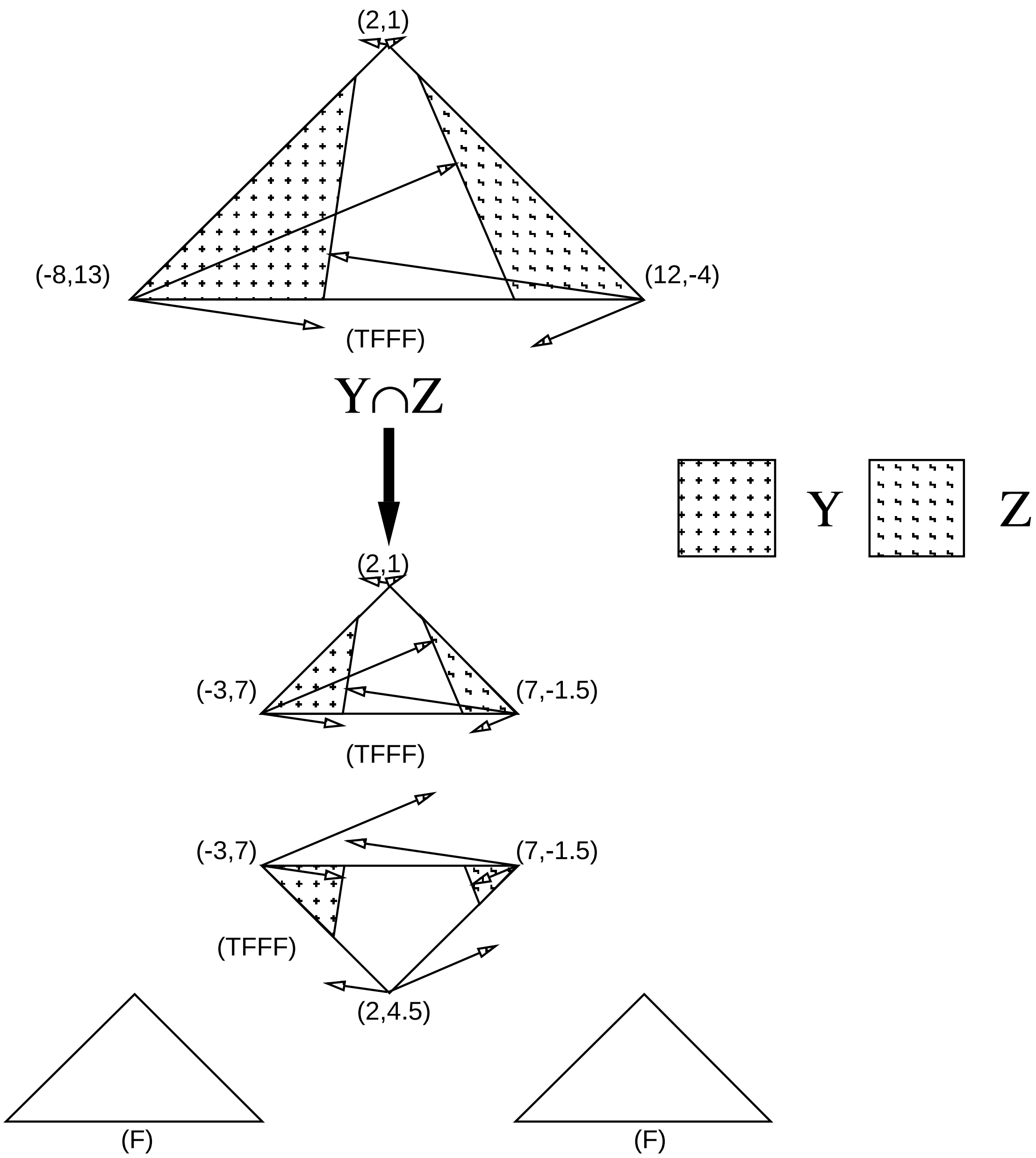


Figure 5.3: Splitting a quadtree leaf

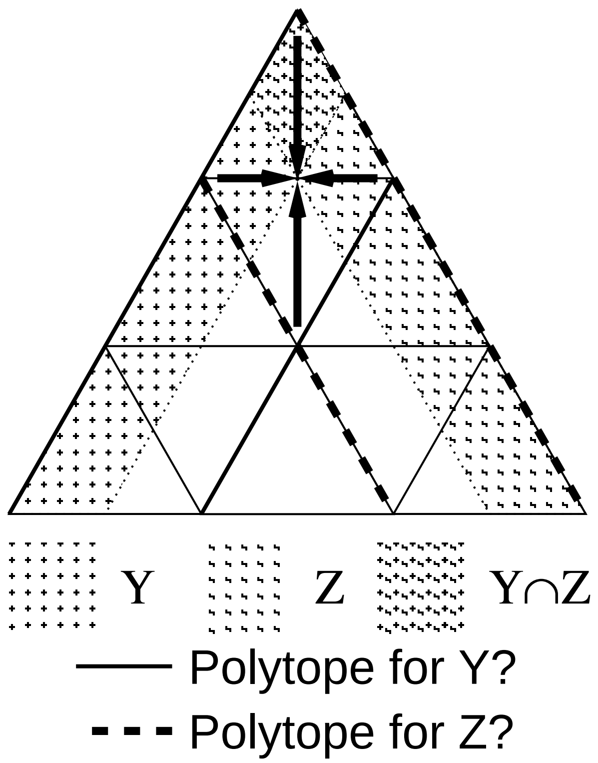


Figure 5.4: Vertices can be displaced further than edges

where the boundaries lie, either of the solid bold segments of the triangulation may be displaced to the boundary of solid Y, and either of the dashed bold segments may be displaced to the boundary of solid Z. The amount of the displacement in any case is just half the distance between parallel lines in the triangulation. The combined effect of the two edge displacements, however, is to move a particular vertex of the triangulation up to twice that distance. In the figure, displacement may move any of the vertices at the base of an arrow to the intersection of the surface boundaries. If, for example, the topmost vertex is displaced to the intersection of the boundaries, the elements that contain the vertex may be highly distorted.

Another problem can arise when two surface boundaries meet at a flat or sharp angle. In figure 5.5, for example, the intersection of two solids forms a flat angle. The approximating polytopes for the two boundaries intersect along a series of edges of the triangulation. In a careless application of mesh displacement, any of

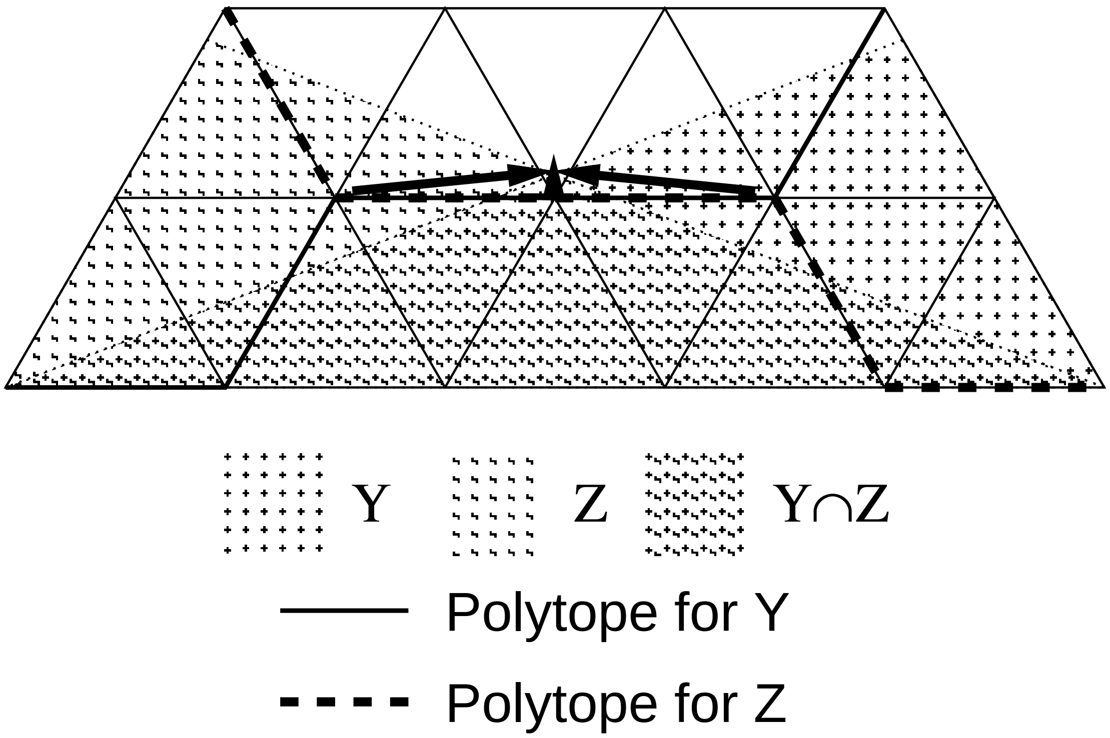


Figure 5.5: Approximating polytopes can have common boundaries

three different vertices of the triangulation might be displaced to the point where the boundaries intersect.

One solution to the vertex displacement problem, and to generalizations of it in higher dimensions, is to identify and displace vertices that are simultaneously near k solid boundaries for values of k decreasing from d, the dimension of the space, to 1. In the three-dimensional case, the proposed solution first identifies elements of the triangulation that contain the intersection of three object boundaries. For each such element, the vertex nearest the intersection is identified as a corner vertex of the composite object, and is eventually displaced to the intersection point. Next, the method identifies faces of the triangulation that contain the intersection of two object boundaries. For each such face, the vertex nearest the intersection is identified as an edge vertex of the composite object, unless some other vertex has already been marked as a corner vertex. Edge vertices are eventually displaced to the intersection curve. Finally, edges of the triangulation that cross an object boundary are examined and the nearer endpoint is identified as a vertex of the composite object, unless the other endpoint has already been so identified, and eventually displaced to the boundary.

Another problem, and one that makes impossible a bound on aspect ratio for the elements that make up composite objects, is that smooth objects may intersect at arbitrarily sharp angles. In figure 5.6, for example, the intersection of two smooth objects forms a sharp wedge, and inevitably some element must contain a sharp angle. Moreover, the approximating polygons for the two solids overlap near the boundary intersection, so that the composite solid might have a dangling edge were care not taken to avoid it. In cases such as these where two edges overlap, it is necessary to introduce a local change to the mesh, in which the overlapping edges are pulled apart by the introduction of new elements and duplicated vertices, so that mesh displacement can pull two copies of the original edge in two different directions to the distinct object boundaries. Figure 5.7 portrays such a local change for the triangulation of figure 5.6.

5.2 Algebraic singularities

Many of the ideas in this dissertation evolved during the construction of a program to produce polyhedral approximations to the boundaries of smooth algebraic sets. Not all algebraic sets are smooth, however. Cubic plane curves, for example, can have cusps, where the curve abruptly changes direction, and crossings, where one branch of the curve crosses another. In three dimensions, object boundaries may include cuspoidal points or curves, and crossing points or curves. These points and curves are examples of *singular* sets, which consist of points at which all first derivatives of the defining function vanish.

The algorithms presented for smooth solids are not likely to displace a vertex of the triangulation exactly to a cusp or crossing point, though such an outcome

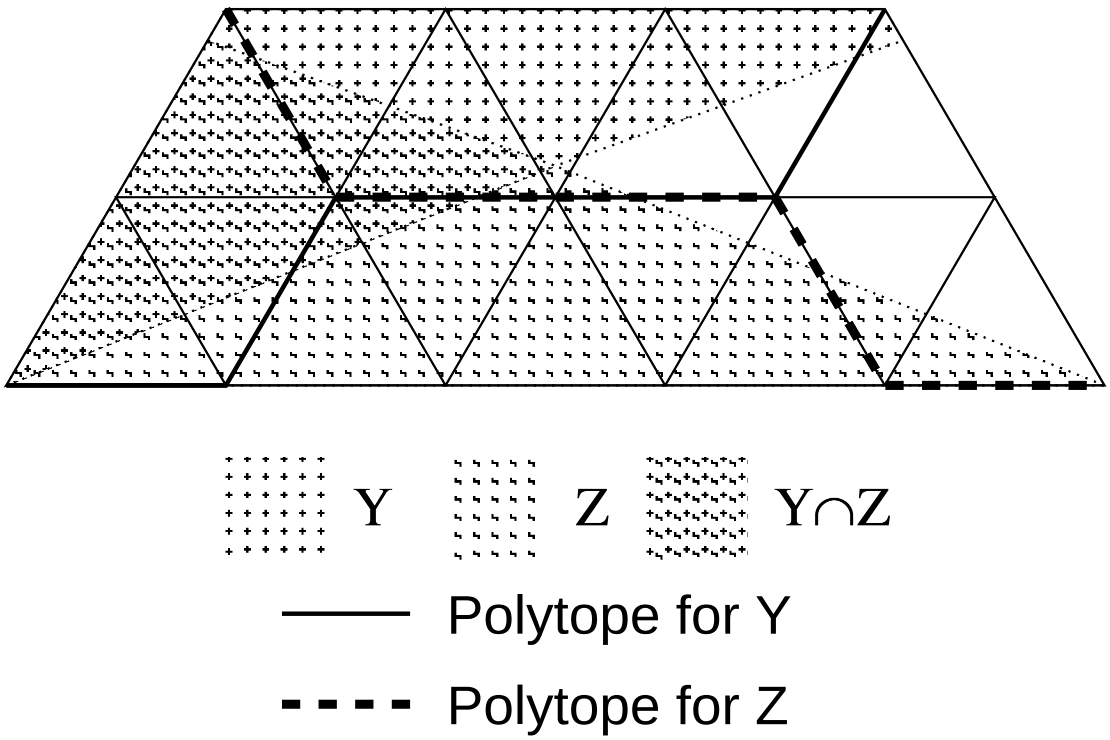


Figure 5.6: Combining smooth objects can produce bad aspect ratios

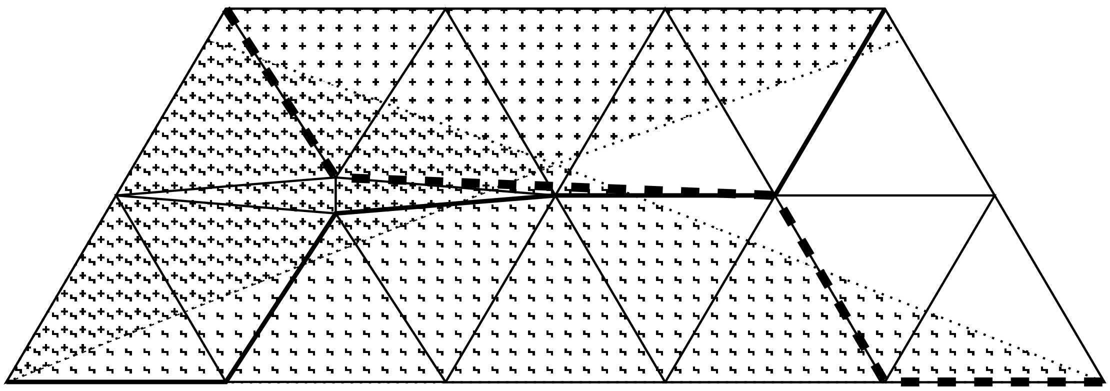


Figure 5.7: Adjusting for sharp angles

would be desirable. Indeed, displacement in a steepest descent direction is particularly unlikely to produce an appealing approximation, because points in a cluster near both branches of a curve can be displaced in two different directions. The approximation of a solid with a cuspoidal curve on its boundary, such as a keen-edged flying saucer, would likely be saucer-shaped, but with a serrated edge, the result of vertices near the singular curve being displaced somewhat haphazardly.

To better approximate solids with singular boundaries, mesh displacement must be modified to move vertices not only toward the boundary of the solid, but especially toward singular points. For a solid in three dimensions defined by $p(x, y, z) \leq 0$ for some polynomial p of degree k, the squared norm of the gradient of p, $a(x, y, z) = \|\nabla p(x, y, z)\|_2^2$, is a nonnegative polynomial of degree $2k - 2$ that vanishes exactly at singular points on the boundary of the solid. A vertex can be displaced not just to a nearby point where p vanishes, but to a nearby point where p vanishes and a is small. Two adaptations of mesh displacement are apparent.

Because a singular point may appear within the neighborhood of no vertex of the triangulation, vertices might have to be permitted to move outside their neighborhoods to align with singular points. Alternatively, more local refinement to introduce a vertex near a singular point is an option. Either scheme is likely to weaken aspect ratio bounds given for smooth solids. The high polynomial degree of a is also discouraging, since it requires almost 2^d times as long to evaluate as p in general. More efficient methods for displacement in the presence of singular points deserve more investigation.

Chapter 6

Conclusions

Simplices have advantages for computation that arise from their essential simplicity. For example:

- Given a polytope, function values at the vertices of the polytope, and an arbitrary point within the polytope, how could one generate a reasonable approximation to the function value at the point? If the polytope is a simplex, an approximation by linear interpolation is obvious and easily computed. For other polytopes, many approximation schemes are possible, some simple, but biased in some arbitrary way, others fairer, but also more complicated. Simplicial interpolants that take into account derivative information are similarly simpler than their arbitrary polytopal counterparts.

- Given a polyhedral grid of signed function values, and the responsibility for rapidly identifying a set of polygons to approximate the set of zero function values, how can one quickly identify those polygons over a particular cell? For the Marching Cubes method of Lorenson and Cline [LC87], based on a cubic grid, a table of 2^8 triangle sets, one for each possible set of negatively valued vertices, provides that computation speed. By examining the signs of the vertices and looking in a table, edges from positive to negative vertices are identified, and linear interpolation selects a point on the edge as lying on the approximate zero contour of the function. The table describes how to connect such points into triangles. With a tetrahedral grid, the table would require only 2^4 elements.

More significantly, as observed by [Dür88], the Marching Cubes algorithm fails to produce a continuous surface for some data. Its fault is that with cubes, the choice of which way to divide a square face with alternating positive and negative values depends on values other than those on the face itself. Several methods have been presented that fix this fault in the algorithm. The division of a triangular face cannot be ambiguous, so an algorithm called Marching Tetrahedra could not have the same flaw.

- Given a polytope, and an arbitrary hyperplane separating the polytope into two pieces, what shapes might the pieces and their boundary have? If the polytope is a simplex, the parts are all simploids, a class of polytopes built from simple operations on simplices, and that can easily be decomposed into simplices [DM82,Moo92]. If the polytope is arbitrary, so are the pieces. This distinction has led to the use of simplices in work in progress involving the integration of piecewise polynomial functions over polyhedral domains.

These are not abstract problems, but problems from practical programs that can be solved more easily with simplices than with boxes. The main obstacle to the use of such techniques is that data is so often measured at the vertices of a cubic grid.

In the hope that engineers, scientists, and programmers might gain a greater appreciation, or at least a tolerance, of simplicial methods, this dissertation has presented several properties of simplices, algorithms for the refinement of simplicial meshes, and general techniques for solving certain applied problems with the help of simplices. There remain several research directions to follow.

The efficient expression of simplicial mesh generation algorithms on multiprocessors is a practical direction, particularly since meshes often underlie numerical computations that scientists and engineers would like to solve on multiprocessors. The development of provably good mesh improvement techniques, similar in spirit to "Laplacian smoothing," is another practical problem. The extension of these simplicial techniques to higher order meshes of polynomially deformed elements is still another.

The development of high continuity piecewise polynomial interpolants over simplices and of multivariate splines defined over simplicial meshes lies in another, more mathematical direction. Improving the aspect ratio bounds for smooth solids and limiting the growth in balancing quadtrees of various kinds are two other interesting mathematical problems to pursue.

Some of the incidental discoveries made along the way may even lead to improved techniques for generating meshes of boxes. The application of vertex marking to triangulate balanced simplicial quadtrees has an unusual analog that permits the generation of locally adaptive square meshes from an appropriate hierarchy of squares. That analog does not obviously extend to cubes, but the question deserves further consideration.

Bibliography

[BEG90] Marshall Bern, David Eppstein, and John Gilbert. Provably good mesh generation. In *31st IEEE Symposium on Foundations of Computer Science*. IEEE, 1990.

[Blo88] Jules Bloomenthal. Polygonalization of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[BSW83] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman et al., editors, *Scientific Computing*, pages 3–17. North-Holland Publishing Company, 1983.

[Buc85] B. Buchberger. Gröbner bases: an algorithmic method in polynomial ideal theory. In N. Bose, editor, *Recent Trends in Multidimensional Systems Theory*, chapter 6. D. Reidel Publishing Company, 1985.

[CFF85] J. C. Cavendish, David Field, and W. B. Frey. An approach to automatic three dimensional finite element mesh generation. *International Journal for Numerical Methods in Engineering*, 21:329–347, 1985.

[Che89] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report TR 89-983, Cornell University, Department of Computer Science, April 1989.

[Cla88] Kenneth L. Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, 17(4):830–847, August 1988.

[Col71] George Collins. The calculation of multivariate polynomial resultants. *Journal of the ACM*, 18(4):515–522, October 1971.

[Cox34] H. S. M. Coxeter. Discrete groups generated by reflections. *Annals of Mathematics*, 6:13–29, 1934.

[Cox73] H. S. M. Coxeter. *Regular Polytopes*. Dover, New York, third edition, 1973.

[DLTW90] David P. Dobkin, Silvio V. F. Levy, William P. Thurston, and Allan R. Wilks. Contour tracing by piecewise linear approximations. *ACM Transactions on Graphics*, 9(4):389–423, October 1990.

[DM82] Wolfgang A. Dahmen and Charles A. Micchelli. On the linear independence of multivariate B-splines, I. triangulations of simploids. *SIAM Journal of Numerical Analysis*, 19(5):993–1012, October 1982.

[Dür88] M. J. Dürst. Additional reference to marching cubes. *Computer Graphics*, 22(2):72–73, 1988.

[Fie86] David A. Field. Implementing Watson's algorithm in three dimensions. In *Second ACM Symposium on Computational Geometry*, pages 246–259, Yorktown Heights, New York, 1986. ACM.

[Fle77] Wendell Fleming. *Functions of Several Variables*. Springer-Verlag, 1977.

[Fre42] H. Freudenthal. Simplizialzerlegungen von beschränkter Flachheit. *Annals of Mathematics*, 43:580–582, 1942.

[HW90] Mark Hall and Joe Warren. Adaptive tessellation of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, 10(6):33–42, November 1990.

[Kuh60] H. W. Kuhn. Some combinatorial lemmas in topology. *IBM Journal of Research and Development*, 45:518–524, 1960.

[LC87] W. Lorenson and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[Lee81] C. W. Lee. Triangulating the d-cube. manuscript, 1981.

[Moo92] Doug Moore. Understanding simploids. In David Kirk, editor, *Graphics Gems III*, chapter 5, pages 250–255. Academic Press, 1992.

[MW90a] Doug Moore and Joe Warren. Adaptive mesh generation I: Packing space. Technical Report TR 90-106, Rice University, Department of Computer Science, 1990.

[MW90b] Doug Moore and Joe Warren. Adaptive mesh generation II: Packing solids. Technical Report TR 90-139, Rice University, Department of Computer Science, 1990.

[Par91] V. N. Parthasarathy. On tetrahedron shape distortion measures. Submitted to International Journal of Numerical Methods in Engineering, 1991.

[PPW87] C. Petersen, B. Piper, and A. J. Worsey. Adaptive contouring of a trivariate interpolant. In Gerald Farin, editor, *Geometric Modeling: Algorithms and New Trends*. SIAM, 1987.

[RV82] A. A. G. Requicha and H. B. Voelcker. Solid modeling: A historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, pages 9–24, March 1982.

[RV83] A. A. G. Requicha and H. B. Voelcker. Solid modeling: Current status and research directions. *IEEE Computer Graphics and Applications*, pages 25–37, October 1983.

[Sal82] J. F. Sallee. A triangulation of the n-cube. *Discrete Mathematics*, 40:81–86, 1982.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990.

[Sen81] Marjorie Senechal. Which tetrahedra fill space? *The Mathematics Magazine*, 54(5):227–243, November 1981.

[She88] Mark S. Shephard. Approaches to the automatic generation and control of finite element meshes. *Applied Mechanics Reviews*, 41(4):169–185, April 1988.

[Smi88] W. D. Smith. *Studies in Discrete and Computational Geometry*. Ph.D. dissertation, Princeton University, 1988.

[Tod76] M. J. Todd. *The Computation of Fixed Points and Applications*, volume 124 of *Lecture Notes in Economics and Mathematical Systems*. Springer, Berlin, 1976.

[VB87] B. Von Herzen and A. Barr. Accurate triangulations of deformed intersecting surfaces. *Computer Graphics*, 21(4):103–110, 1987.

[vdLT80] G. van der Laan and A. J. J. Talman. An improvement of fixed point algorithms by using a good triangulation. *Mathematical Programming*, 18:274–285, 1980.

[War91] Joe Warren. Multi-sided rational Bézier surfaces and newton polygons. In preparation, 1991.

[Wei81] Alan Weiser. *Local-Mesh, Local-Order, Adaptive Finite Element Methods with* a Posteriori *Error Estimators for Elliptic Partial Differential Equations*. Ph.D. dissertation, Yale University, 1981.

[Whi57] H. Whitney. *Geometric Integration Theory*. Princeton University Press, 1957.

[WMW86] G. Wyvil, C. McPheeters, and B. Wyvil. Data structure for *soft* objects. *The Visual Computer*, 2:227–234, 1986.

[YS84] M. A. Yerry and M. S. Shephard. Automatic three-dimensional mesh generation by the modified octree technique. *International Journal for Numerical Methods in Engineering*, 20:1965–1990, 1984.