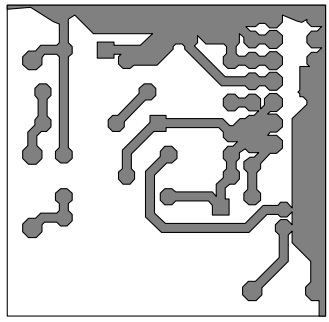
14 Quadtrees

Non-Uniform Mesh Generation

Almost all electrical devices, from shavers and telephones to televisions and computers, contain some electronic circuitry to control their functioning. This circuitry—VLSI circuits, resistors, capacitors, and other electric components—is placed on a printed circuit board. To design printed circuit boards one has to decide where to place the components, and how to connect them. This raises a number of interesting geometric problems, of which this chapter tackles one: mesh generation.



Many components on a printed circuit board emit heat during operation. In order for the board to function properly, the emission of heat should be below a certain threshold. It is difficult to predict in advance whether the heat emission will cause problems, since this depends on the relative positions of the components and the connections. In former days one therefore made a prototype of the board to determine the heat emission experimentally. If it turned out to be too high, an alternative layout had to be designed. Today the experiments can often be simulated. Because the design is largely automated, a computer model of the board is readily available and simulation is a lot faster than building a prototype. Simulation also allows testing during the initial stages of the design phase, so that faulty designs can be rejected as early as possible.

The heat transfer between different materials on the printed circuit board is a quite complicated process. To simulate the heat processes on the board one therefore has to resort to approximation using *finite element methods*. Such methods first divide the board into many small regions, or *elements*. The elements are usually triangles or quadrilaterals. The heat that each element emits by itself is assumed to be known. It is also assumed to be known how neighboring elements influence each other. This leads to a big system of equations, which is then solved numerically.

The accuracy of finite element methods depends heavily on the mesh: the finer the mesh, the better the solution. The other side of the coin is that the computation time for the numerical process increases drastically when the number of elements increases. So we would like to use a fine mesh only where necessary. Often this is at the border between regions of different material. It is also important that the mesh elements respect the borders, that is, that any

mesh element is contained in only one region. Finally, the shape of the mesh elements plays an important role: irregularly shaped elements such as very thin triangles often lead to a slower convergence of the numerical process.

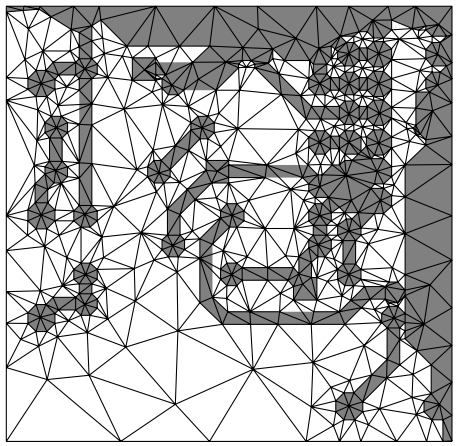
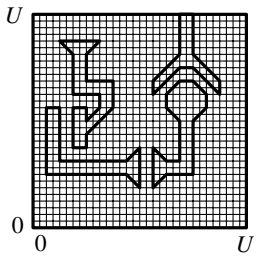
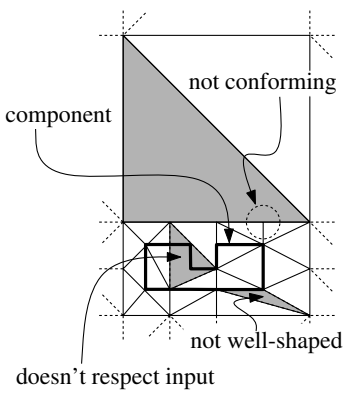


Figure 14.1
Triangular mesh of a part of a printed circuit board





14.1 Uniform and Non-Uniform Meshes

We'll study the following variant of the mesh generation problem. The input is a square—the printed circuit board—with a number of disjoint polygonal components inside it. The square together with the components is sometimes called the *domain* of the mesh. The vertices of the square are at $(0,0)$, $(0,U)$, $(U,0)$, (U,U), where $U = 2^j$ for a positive integer j. The coordinates of the vertices of the components are assumed to be integers between 0 and U. We make one more assumption, which is satisfied in a number of applications: the edges of the components have only four different orientations. In particular, the angle that an edge makes with the x-axis is either $0°$, $45°$, $90°$, or $135°$.

Our goal is to compute a *triangular mesh* of the square, that is, a subdivision of the square into triangles. We require the mesh to have the following properties:

- The mesh must be *conforming*: a triangle is not allowed to have a vertex of another triangle in the interior of one of its edges.
- The mesh must *respect the input*: the edges of the components must be contained in the union of the edges of the mesh triangles.
- The mesh triangles must be *well-shaped*: the angles of any mesh triangle should not be too large nor too small. In particular, we require them to be in the range from $45°$ to $90°$.

Finally, we would like the mesh to be fine only where necessary. Where this is depends on the application. The property we shall require is as follows.

- The mesh must be *non-uniform*: it should be fine near the edges of the components and coarse far away from the edges.

We have already seen triangulations earlier in this book. Chapter 3 presented an algorithm for triangulating a simple polygon, and Chapter 9 presented an algorithm for triangulating a point set. The latter algorithm computes the Delaunay triangulation, a triangulation that maximizes the minimum angle over all possible triangulations. Given our restriction on the angles of the mesh triangles this seems quite useful, but there are two problems.

First of all, a triangulation of the vertices of the components need not respect the edges of the components. Even if it did, there is a second problem: there can still be angles that are too small. Suppose that the input is a square with side length 16 and one component, which is a small square with side length 1 placed in the top left corner at distance 1 from the left and the top side of the square. Then the Delaunay triangulation contains triangles that have an angle of less than 5°. Since the Delaunay triangulation maximizes the minimum angle, it seems impossible to generate a mesh with only well-shaped triangles. But there is a catch: unlike in a triangulation, the triangles in a mesh are not required to have their vertices at the input points. We are allowed to add extra points, called *Steiner points*, to help us obtain well-shaped triangles. A triangulation that uses Steiner points is sometimes called a *Steiner triangulation*. In our example, if we add a Steiner point at every grid point inside the square, then we can easily obtain a mesh consisting only of triangles with two 45° angles and one 90° angle—see the mesh shown on the left in Figure 14.2. Unfortunately, this

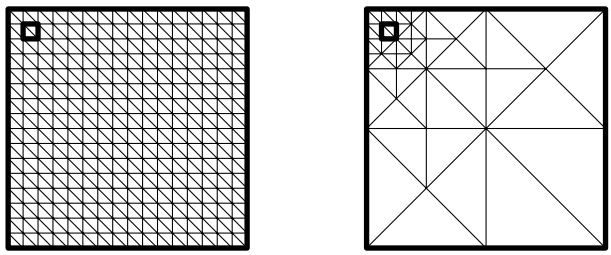


Figure 14.2
A uniform and a non-uniform mesh

mesh suffers from another problem: it uses small triangles everywhere, not only near the edges of the input, so it is a *uniform* mesh. As a result, it has many triangles. We cannot simply replace all the triangles in, say, the bottom right quarter of the square by two big triangles, because then the mesh would no longer be conforming. Nevertheless, if we gradually increase the size of the triangles when we get farther away from top left corner, then it is possible to get a conforming mesh with only well-shaped triangles, as shown on the right in Figure 14.2. This leads to a significantly smaller number of triangles: the uniform mesh has 512 triangles, whereas the non-uniform one only has 52.

14.2 Quadtrees for Point Sets

The non-uniform mesh generation method we shall describe in the next section is based on *quadtrees*. A quadtree is a rooted tree in which every internal node has four children. Every node in the quadtree corresponds to a square. If a

node v has children, then their corresponding squares are the four quadrants of the square of v—hence the name of the tree. This implies that the squares of the leaves together form a subdivision of the square of the root. We call this subdivision the *quadtree subdivision*. Figure 14.3 gives an example of a quadtree and the corresponding subdivision. The children of the root are

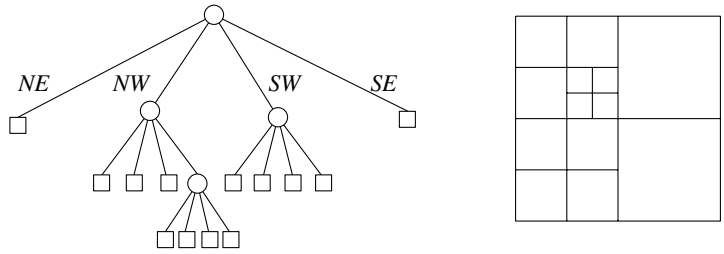
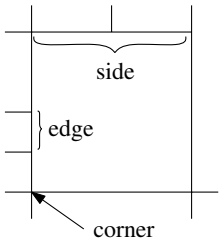


Figure 14.3
A quadtree and the corresponding subdivision

labelled NE, NW, SW, and SE to indicate to which quadrant they correspond; NE stands for the north-east quadrant, NW for the north-west quadrant, and so on.

Before we continue, we introduce some terminology related to quadtree subdivisions. The faces in a quadtree subdivision have the shape of a square. Although they can have more than four vertices, we shall call them squares anyway. The four vertices at the corners of the square are called *corner vertices*, or *corners* for short. The line segments connecting consecutive corners are the *sides of the square*. The edges of the quadtree subdivision that are contained in the boundary of a square are called the *edges of the square*. Hence, a side contains at least one, but possibly many more, edges. We say that two squares are *neighbors* if they share an edge.



Quadtrees can be used to store different types of data. We will describe the variant that stores a set of points in the plane. In this case the recursive splitting of squares continues as long as there is more than one point in a square. So the definition of a quadtree for a set P of points inside a square σ is as follows. Let $\sigma := [x_\sigma : x'_\sigma] \times [y_\sigma : y'_\sigma]$.

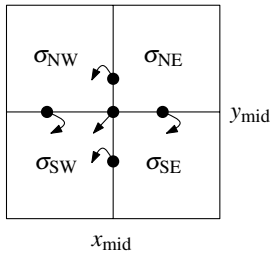
- If $\text{card}(P) \leqslant 1$ then the quadtree consists of a single leaf where the set P and the square σ are stored.
- Otherwise, let σ_{NE}, σ_{NW}, σ_{SW}, and σ_{SE} denote the four quadrants of σ. Let $x_{\text{mid}} := (x_\sigma + x'_\sigma)/2$ and $y_{\text{mid}} := (y_\sigma + y'_\sigma)/2$, and define

$$\begin{array}{lcl} P_{\text{NE}} & := & \{p \in P \,:\, p_x > x_{\text{mid}} \text{ and } p_y > y_{\text{mid}}\}, \\ P_{\text{NW}} & := & \{p \in P \,:\, p_x \leqslant x_{\text{mid}} \text{ and } p_y > y_{\text{mid}}\}, \\ P_{\text{SW}} & := & \{p \in P \,:\, p_x \leqslant x_{\text{mid}} \text{ and } p_y \leqslant y_{\text{mid}}\}, \\ P_{\text{SE}} & := & \{p \in P \,:\, p_x > x_{\text{mid}} \text{ and } p_y \leqslant y_{\text{mid}}\}. \end{array}$$

  The quadtree now consists of a root node v where the square σ is stored. Below we shall denote the square stored at v by $\sigma(v)$. Furthermore, v has four children:

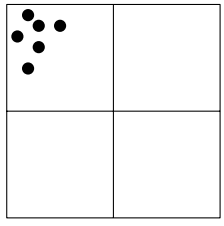
- the NE-child is the root of a quadtree for the set P_{NE} inside the square σ_{NE},
- the NW-child is the root of a quadtree for the set P_{NW} inside the square σ_{NW},
- the SW-child is the root of a quadtree for the set P_{SW} inside the square σ_{SW},
- the SE-child is the root of a quadtree for the set P_{SE} inside the square σ_{SE}.

The choice of using less-than-or-equal-to and greater-than in the definition of the sets P_{NE}, P_{NW}, P_{SW}, and P_{SE} means that we define the vertical splitting line to belong to the left two quadrants, and the horizontal splitting line to the lower two quadrants.



Every node v of the quadtree stores its corresponding square $\sigma(v)$. This is not necessary; we could store only the square of the root of the tree. When we walk down the tree we would then have to maintain the square of the current node. This alternative uses less storage at the expense of extra computations that have to be done when the quadtree is queried.

The recursive definition of a quadtree immediately translates into a recursive algorithm: split the current square into four quadrants, partition the point set accordingly, and recursively construct quadtrees for each quadrant with its associated point set. The recursion stops when the point set contains less than two points. The only detail that does not follow from the recursive definition is how to find the square with which to start the construction. Sometimes this square will be given as a part of the input. If this is not the case, then we compute a smallest enclosing square for the set of points. This can be done in linear time by computing the extreme points in the x- and y-directions.



At every step of the quadtree construction the square containing the points is split into four smaller squares. This does not mean that the point set is split as well: it can happen that all the points lie in the same quadrant. Therefore a quadtree can be quite unbalanced, and it is not possible to express the size and depth of a quadtree as a function of the number of points it stores. However, the depth of a quadtree is related to the distance between the points and the size of the initial square. This is made precise in the following lemma.

Lemma 14.1 *The depth of a quadtree for a set P of points in the plane is at most $\log(s/c) + \frac{3}{2}$, where c is the smallest distance between any two points in P and s is the side length of the initial square that contains P.*

Proof. When we descend from a node to one of its children, the size of the corresponding square halves. Hence, the side length of the square of a node at depth i is $s/2^i$. The maximum distance between two points inside a square is given by the length of its diagonal, which is $s\sqrt{2}/2^i$ for the square of a node at depth i. Since an internal node of a quadtree has at least two points in its associated square, and the minimum distance between two points is c, the depth i of an internal node must satisfy

$$s\sqrt{2}/2^i \geqslant c,$$

which implies

$$i \leqslant \log \frac{s\sqrt{2}}{c} = \log(s/c) + \frac{1}{2}.$$

The lemma now follows from the fact that the depth of a quadtree is exactly one more than the maximum depth of any internal node. □

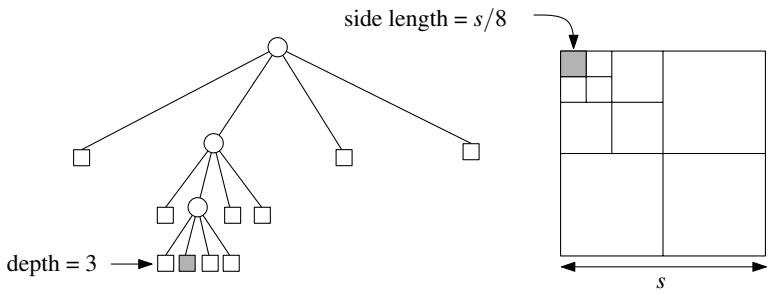


Figure 14.4
A node at depth i corresponds to a square of side length $s/2^i$

The size of a quadtree and the construction time are a function of the depth of the quadtree and the number of points in P.

Theorem 14.2 *A quadtree of depth d storing a set of n points has $O((d+1)n)$ nodes and can be constructed in $O((d+1)n)$ time.*

Proof. Every internal node in a quadtree has four children, so the total number of leaves is one plus three times the number of internal nodes. Hence, it suffices to bound the number of internal nodes.

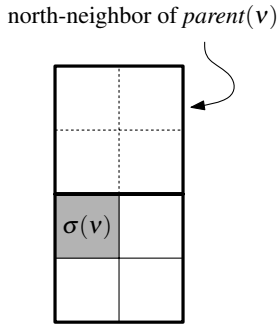
Any internal node has one or more points inside its associated square. Moreover, the squares of the nodes at the same depth in the quadtree are disjoint and exactly cover the initial square. This means that the total number of internal nodes at any given depth is at most n. The bound on the size of the quadtree follows.

The most time-consuming task in one step of the recursive construction algorithm is the distribution of points over the quadrants of the current square. Hence, the amount of time spent at an internal node is linear in the number of points that lie in the associated square. We argued above that the total number of points associated with nodes at the same depth in the tree is at most n, from which the time bound follows. □

An operation on quadtrees that is often needed is *neighbor finding*: given a node v and a direction—north, east, south, or west—find a node v' such that $\sigma(v')$ is adjacent to $\sigma(v)$ in the given direction. Usually the given node is a leaf, and one wants the reported node to be a leaf as well. This corresponds to finding an adjacent square of a given square in the quadtree subdivision. The algorithm we shall describe is a little bit different: the given node v can also be internal, and the algorithm tries to find the node v' such that $\sigma(v')$ is adjacent to $\sigma(v)$ in the given direction and v' and v are at the same depth. If there is no such node, then it finds the deepest node whose square is adjacent. If there is no adjacent square in the given direction—this can happen if $\sigma(v)$ has an edge contained in an edge of the initial square—then the algorithm reports **nil**.

The neighbor-finding algorithm works as follows. Suppose that we want to find the north-neighbor of v. If v happens to be the SE- or SW-child of its

parent, then its north-neighbor is easy to find: it is the NE- or NW-child of the parent, respectively. If v itself is the NE- or NW-child of its parent, then we proceed as follows. We recursively find the north-neighbor, μ, of the parent of v. If μ is an internal node, then the north-neighbor of v is a child of μ; if μ is a leaf, then the north-neighbor we seek is μ itself. The pseudocode for this algorithm is as follows.



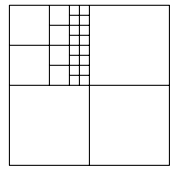
Algorithm NORTHNEIGHBOR($v, \mathcal{T}$)
Input. A node v in a quadtree $\mathcal{T}$.
Output. The deepest node v' whose depth is at most the depth of v such that $\sigma(v')$ is a north-neighbor of $\sigma(v)$, and **nil** if there is no such node.

```
1.  if v = root(T) then return nil
2.  if v = SW-child of parent(v) then return NW-child of parent(v)
3.  if v = SE-child of parent(v) then return NE-child of parent(v)
4.  μ ← NORTHNEIGHBOR(parent(v), T)
5.  if μ = nil or μ is a leaf
6.     then return μ
7.     else if v = NW-child of parent(v)
8.             then return SW-child of μ
9.             else return SE-child of μ
```

This algorithm does not necessarily report a leaf node. If we insist on finding a leaf node, then we have to walk down the quadtree from the node found by our algorithm, always proceeding to a south-child.

The algorithm spends $O(1)$ time at every recursive call. Moreover, at every call the depth of the argument node v decreases by one. Hence, the running time is linear in the depth of the quadtree. We get the following theorem:

Theorem 14.3 *Let $\mathcal{T}$ be a quadtree of depth d. The neighbor of a given node v in $\mathcal{T}$ in a given direction, as defined above, can be found in $O(d+1)$ time.*



We already observed that a quadtree can be quite unbalanced. As a result, large squares can be adjacent to many small squares. In some applications—in particular in the application to meshing—this is unwanted. Therefore we now discuss a variant of quadtrees, the balanced quadtree, that does not have this problem.

A quadtree subdivision is called *balanced* if any two neighboring squares differ at most a factor two in size. A quadtree is called balanced if its subdivision is balanced. So in a balanced quadtree any two leaves whose squares are neighbors can differ at most one in depth. Figure 14.5 shows an example of a quadtree subdivision that has been made balanced. The original subdivision is shown solid, and its refinement is dotted.

A quadtree can be made balanced with the following algorithm:

Algorithm BALANCEQUADTREE($\mathcal{T}$)
Input. A quadtree $\mathcal{T}$.
Output. A balanced version of $\mathcal{T}$.

```
1.  Insert all the leaves of T into a linear list L.
2.  while L is not empty
```

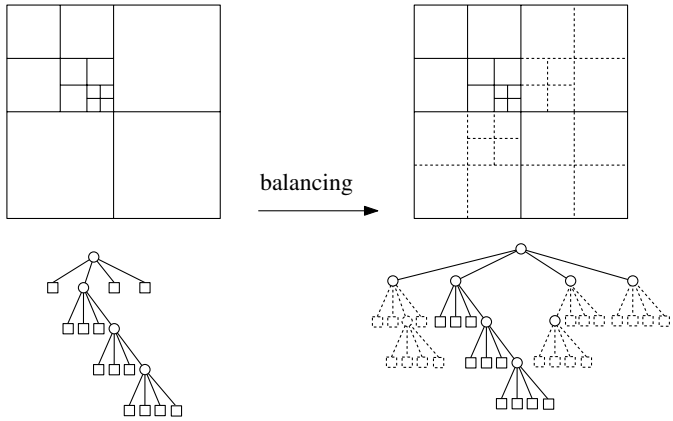


Figure 14.5
A quadtree and its balanced version

3. **do** Remove a leaf μ from $\mathcal{L}$.
4. **if** $\sigma(\mu)$ has to be split
5. **then** Make μ into an internal node with four children, which are leaves that correspond to the four quadrants of $\sigma(\mu)$. If μ stores a point, then store the point in the correct new leaf instead.
6. Insert the four new leaves into $\mathcal{L}$.
7. Check if $\sigma(\mu)$ had neighbors that now need to be split and, if so, insert them into $\mathcal{L}$.
8. **return** $\mathcal{T}$

There are two steps in the algorithm that require some explanation.

First, we have to check whether a given square $\sigma(\mu)$ needs to be split. This means that we have to check whether $\sigma(\mu)$ is adjacent to a square of less than half its size. This can be done using the neighbor-finding algorithm described earlier, as follows. Suppose that we are looking for a north-neighbor of $\sigma(\mu)$ that is less than half the size of $\sigma(\mu)$. There is such a square if and only if NORTHNEIGHBOR($\mu, \mathcal{T}$) reports a node that has a SW-child or a SE-child that is not a leaf.

Second, we have to check if $\sigma(\mu)$ had neighbors that now need to be split. Again, this can be done using the neighbor-finding algorithm: for example, $\sigma(\mu)$ has such a neighbor to the north if and only if NORTHNEIGHBOR($\mu, \mathcal{T}$) reports a node whose square is larger than $\sigma(\mu)$.

So now we have an algorithm to make a quadtree balanced. But before we can analyze the running time of the balancing algorithm we must answer the following question: what happens to the size of the quadtree when we make it balanced? From Figure 14.5 we may get the idea that the complexity of a balanced quadtree subdivision can be quite a lot higher than that of its unbalanced version. First of all, large squares adjacent to very small ones get split up many times. Secondly, the splitting may propagate: sometimes it seems that a square σ need not be split because its neighbors initially have the right size, but these neighbors may have to be split so that σ must be split after all. The next

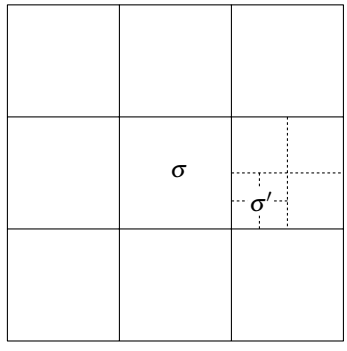
theorem shows that things are not as bad as appears at first sight, and that the balancing can be done efficiently.

Theorem 14.4 *Let $\mathcal{T}$ be a quadtree with m nodes. Then the balanced version of $\mathcal{T}$ has $O(m)$ nodes and it can be constructed in $O((d+1)m)$ time.*

Proof. We first prove the bound on the number of nodes. Denote the balanced version of $\mathcal{T}$ by $\mathcal{T}_B$. The tree $\mathcal{T}_B$ is obtained from $\mathcal{T}$ by a number of splitting operations, which replace a leaf by one internal node with four leaves. We shall prove that only $8m$ splitting operations are performed. Since a single splitting operation increases the total number of (internal and leaf) nodes by four, this proves that the number of nodes in $\mathcal{T}_B$ is $O(m)$, as claimed.

Call the squares of nodes in $\mathcal{T}$ old squares, and the squares of nodes that are in $\mathcal{T}_B$ but not in $\mathcal{T}$ new squares. Suppose that we have to split an—old or new—square σ in the balancing process. (The quadrants of σ may have to be split further, but this will be accounted for separately. Here we only need to account for the increase in the number of nodes by four due to the splitting of σ.) Below we shall prove that at least one of the eight equal-sized squares surrounding σ must be an old square. We charge the splitting of σ to one of these old square. Every old square—equivalently, every node of $\mathcal{T}$—gets charged at most eight splittings in this manner, so the total number of splits is at most $8m$, as claimed.

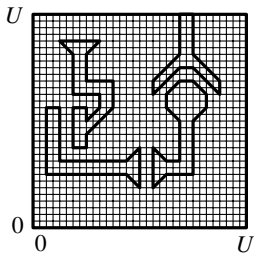
We now prove that for any square that is split in the balancing process, at least one of the eight equal-sized squares surrounding it must be an old square. Suppose there would be squares for which this claim does not hold. Let σ be the smallest such square. Since σ is split, it must have been adjacent to a small square, one of less than half its size. Let σ' be the square that has exactly half the size of σ and that contains this small square. Since σ' is contained in a new square, it is also new itself. Hence, it must have been split in the balancing process. Now we observe that all eight equal-sized squares surrounding σ' must be new, because they are either contained in one the squares surrounding σ (which are new) or in σ (which is assumed to be split in the balancing process). Hence, σ', which is smaller than σ, is a square that is split and none of whose eight surrounding squares is old. This contradicts the definition of σ, thus finishing the proof of the first part of the theorem.

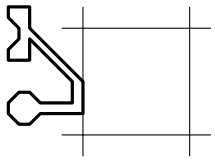


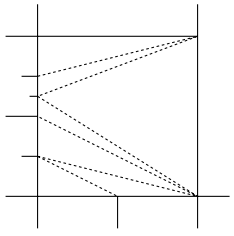
What remains is to prove that BALANCEQUADTREE takes $O((d+1)m)$ time. The time needed to handle a node μ is $O(d+1)$, because a constant number of neighbor-finding operations is performed. Since any node is handled at most once and the total number of nodes is $O(m)$, the total time is $O((d+1)m)$. □

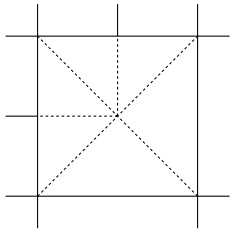
14.3 From Quadtrees to Meshes

We now return to the mesh generation problem. Recall that the input is a square $[0:U]\times[0:U]$, where $U=2^j$ for some positive integer j, with a number of disjoint polygonal components inside it. The polygon vertices have integer









coordinates, and the polygon edges have one of four possible orientations: the angle that any edge makes with the positive x-axis is 0°, 45°, 90°, or 135°. The goal is to compute a triangular mesh of the square (both outside and inside the components) that is conforming, respects the input, has well-shaped triangles, and is non-uniform.

The idea is to use a quadtree subdivision as the first step towards a mesh. When we construct a quadtree on a point set, then we stop the recursive construction when a square contains less than two points. Since we are now dealing with polygonal input, we have to reformulate the stopping criterion. Because we want the mesh triangles to be fine near the edges of the components, we keep on splitting as long as a square is intersected by an edge. More precisely, the new stopping criterion is this: we stop splitting when the square is no longer intersected by any edge of any component, or when it has unit size. We consider the square and the edges to be closed, so that for instance a square that has an edge contained in one of its sides is intersected by that edge. This stopping criterion guarantees that the quadtree subdivision will be non-uniform: the edges of the components will be surrounded by unit size squares, and the squares will be bigger farther away from the edges.

We claim that the interior of any square in the resulting quadtree subdivision can be intersected by an edge of a component in only one way: the intersection must be a diagonal of the square. Indeed, squares whose closure is intersected have unit size, and the vertices of the components have integers coordinates. Hence, the interior of a square cannot be intersected by a horizontal or vertical edge, and the intersection with an edge of orientation 45° or 135° must be a diagonal. It seems that we only have to add diagonals to the squares whose interior is not intersected to obtain a good mesh. The resulting triangles will respect the input, they will be well-shaped, and the mesh will be non-uniform. Unfortunately, the mesh is not conforming. We can remedy this by taking the subdivision vertices on the sides of the square into account when we triangulate it, but this leads to another problem: if a square has many vertices on a side, then not all triangles we get are well-shaped.

To avoid these problems we make the quadtree subdivision balanced before we triangulate it. Once we have a balanced quadtree subdivision, we can easily generate a mesh with well-shaped triangles as follows. Squares that do not have a vertex in the interior of one of their sides (and that are not already triangulated by an edge of one of the components) are triangulated by adding a diagonal. Because the subdivision is balanced, the remaining squares have at most one vertex in the interior of each side. Moreover, this vertex must be in the middle of the side. Hence, if we add one Steiner point in the center of such a square and connect it to all vertices on the boundary, then we get only triangles with two 45° angles and one 90° angle.

Summarizing, the mesh generation algorithm is as follows. GENERATEMESH constructs the mesh in the form of a doubly-connected edge list. We omit the details related to handling the doubly-connected edge list—in particular, how to construct the doubly-connected edge list corresponding to a given quadtree.

Algorithm GENERATEMESH(S)
Input. A set S of components inside the square $[0:U]\times[0:U]$ with the properties stated at the beginning of this section.
Output. A triangular mesh $\mathcal{M}$ that is conforming, respects the input, consists of well-shaped triangles, and is non-uniform.

1. Construct a quadtree $\mathcal{T}$ on the set S inside the square $[0:U]\times[0:U]$ with the following stopping criterion: a square is split as long as it is larger than unit size and its closure intersects the boundary of some component.
2. $\mathcal{T} \leftarrow$ BALANCEQUADTREE($\mathcal{T}$)
3. Construct the doubly-connected edge list for the quadtree subdivision $\mathcal{M}$ corresponding to $\mathcal{T}$.
4. **for** each face σ of $\mathcal{M}$
5. **do if** the interior of σ is intersected by an edge of a component
6. **then** Add the intersection (which is a diagonal) as an edge to $\mathcal{M}$.
7. **else if** σ has only vertices at its corners
8. **then** Add a diagonal of σ as an edge to $\mathcal{M}$.
9. **else** Add a Steiner point in the center of σ, connect it to all vertices on the boundary of σ, and change $\mathcal{M}$ accordingly.
10. **return** $\mathcal{M}$

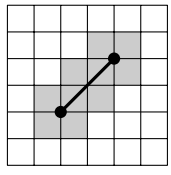
The following theorem summarizes the properties of the mesh that is constructed by our algorithm.

Theorem 14.5 *Let S be a set of disjoint polygonal components inside the square $[0:U]\times[0:U]$ with the properties stated in the beginning of this section. Then there exists an non-uniform triangular mesh for this input that is conforming, respects the input, and has only well-shaped triangles. The number of triangles is $O(p(S)\log U)$, where $p(S)$ is the sum of the perimeters of the components in S, and the mesh can be constructed in $O(p(S)\log^2 U)$ time.*

Proof. The properties of the mesh—that it is non-uniform, conforming, that it respects the input, and consists of well-shaped triangles—follow from the discussion above. What remains is to bound the size and preprocessing time of the mesh.

The mesh construction has three phases: first we construct a quadtree subdivision, then we make it balanced, and finally we triangulate it.

To bound the size of the quadtree subdivision resulting from the first phase we use the following observation. Consider the grid whose cells have unit size. Then the number of cells whose closure is intersected by a line segment of length l is at most $4+3l/\sqrt{2}$. It follows that the number of cells whose closure is intersected by an edge of any of the components is $O(p(S))$. Obviously the same is true for any grid with larger cells. Hence, the number of internal nodes of the quadtree at a fixed depth is $O(p(S))$. The depth of the quadtree is $O(\log U)$, since we stop splitting once a cell has unit size. The total number of nodes of the quadtree and, hence, the complexity of the corresponding subdivision, is therefore $O(p(S)\log U)$.



We know from Theorem 14.4 that making a quadtree subdivision balanced does not increase its complexity asymptotically. The same is true for the third phase of the mesh generation, the triangulation of the squares of the balanced subdivision. (This follows because each cell is subdivided into a constant number of triangles.) We conclude that the number of triangles of the final mesh is linear in the complexity of the quadtree subdivision resulting from the first phase, which we just proved to be $O(p(S)\log U)$.

What remains is to prove the bound on the construction time. The first phase is dominated by the time needed to do the splitting steps in the recursive quadtree construction algorithm. For a given node this is linear in the number of component edges intersecting its closure. We argued above that the sum of the number of intersecting edges over all nodes at a fixed depth is $O(p(S))$. Hence, the total time needed in the first phase is $O(p(S)\log U)$. By Theorem 14.4, the time needed to do the balancing introduces an extra $O(\log U)$ factor. Constructing a doubly-connected edge list from a given quadtree, as well as triangulating the balanced subdivision, can be done within the same amount of time. The bound on the preprocessing time follows. □

14.4 Notes and Comments

Quadtrees were one of the first data structures for higher-dimensional data. They were developed by Finkel and Bentley in 1974 [177]. Since then, there have been hundreds of papers dealing with quadtrees. The surveys and books by Samet [332, 333, 334, 335] and the handbook chapter by Aluru [14] give an extensive overview of the various types of quadtrees and their applications.

As we have seen in this chapter, the size and depth of a quadtree for a set of n points cannot be bounded in terms of the number of the number of points only. The reason is that there can be many nodes with only one non-empty subtree. If one removes such nodes from the tree, then its size becomes linear. The resulting structure is called a *compressed quadtree*—see for example the survey paper by Aluru [14]. Combining the compressed quadtree with ideas from skip lists, Eppstein et al. [173] developed a quadtree variant that has not only linear size, but also allows insertions, deletions, and search operations in $O(\log n)$ time.

Mesh generation is but one application of quadtrees. They are also used in computer graphics, image analysis, geographic information systems, and many other areas. Quadtrees are often used to answer range queries. From a theoretical point of view quadtrees are not the best solution to range searching problems, because usually no sublinear bounds on the query time can be proven. Other solutions to various range searching problems are described in Chapters 5, 10, and 16. In practice, quadtrees often seem to perform well. Quadtrees have also been applied to solve tasks like hidden surface removal, ray tracing, medial axis transforms, map overlay of raster maps, and nearest neighbor finding.

Quadtrees can easily be generalized to higher dimensions, where they are usually called *octrees*.

The mesh generation problem is important in many areas and it has been studied extensively, both in the plane and in 3-dimensional space. The surveys by Bern and Eppstein [62], Bern and Plasman [65], Bern [61], and Ho-Le [215] are good starting points to look for results on a specific setting of the mesh generation problem. We shall briefly describe a few of the results.

One can distinguish *structured* and *unstructured* meshes. Structured meshes are usually (deformed) grids; unstructured meshes are often triangulations or quadrilateral meshes. We shall restrict our discussion to unstructured triangulations. Furthermore, we mainly concentrate on the case where the domain to be meshed is 2-dimensional and polygonal. In most applications there are requirements on the mesh that are similar to the ones imposed in this chapter. In particular, one usually requires the mesh to be conforming—sometimes the term "consistent" is used—and to respect the input. Furthermore, some sort of well-shapedness is important. This usually means that the triangles have to satisfy one or both of the following criteria: (i) There should be no small angles, that is, every angle should be at least some fixed (not too small) constant θ. This constant should not be larger than the smallest angle of the input domain, because we cannot avoid the input angles in the mesh. (ii) There should be *no obtuse angles*, that is, no angles larger than 90°. In the example studied in this chapter we wanted the triangles to satisfy both criteria, and we then took $\theta = 45^\circ$. Often the goal is to minimize the number of mesh elements under the given conditions. This implies some sort of non-uniformity: small triangles should be used only where needed.

We first consider minimizing the number of triangles in the mesh under the condition that there be no small angles. The number of triangles that we obtain in a mesh of a polygonal domain under this condition not only depends on the number of vertices of the domain, it also depends on the shape of the domain. To see this, we introduce a parameter that is closely related to the minimum angle of a triangle, namely the *aspect ratio* of the triangle. This is the ratio of the length of the longest side of the triangle to the height of the triangle, where the height of a triangle is the Euclidean distance of the longest edge to its opposite vertex. If the smallest angle of a triangle is θ then the aspect ratio is between $1/\sin\theta$ and $2/\sin\theta$. Now consider a rectangular domain whose shorter sides have length 1 and whose longer sides have length A. Suppose we require that the minimum angle be, say, 30°. This implies that the aspect ratio of any triangle in the mesh must be less than or equal to $2/\sin 30^\circ = 4$. Furthermore, the height of any triangle in the domain is at most one. Hence, the area of any triangle is $O(1)$. Because the total area of the rectangular domain is A, this implies that we need at least $\Omega(A)$ triangles in the mesh. Bern et al. [64] describe a method based on quadtrees that produces an asymptotically optimal number of triangles. The method described in this chapter is based on their technique. Other mesh-generation algorithms are based on the Delaunay triangulation described in Chapter 9. See for example the work by Shewchuck [356, 357] for a discussion of Delaunay-based algorithms in the plane.

If the only requirement is that the triangles in the mesh are non-obtuse then it turns out to be possible to construct a mesh for a given polygonal domain whose number of triangles only depends on the number of vertices of the domain.

More precisely, Bern and Eppstein [63] have shown that for any polygonal domain with n vertices there is a mesh consisting of $O(n^2)$ non-obtuse triangles. Bern et al. [67] improved this bound to $O(n)$.

Melissaratos and Souvaine [278] extended the approach of Bern et al. [64] for computing a mesh without small angles so that it also avoids obtuse triangles. The number of triangles in the mesh is still at most a constant factor from optimal.

Minimizing the number of triangles is not always the goal of meshing algorithms. It can also be important to be able to control the mesh density, so that one can have a dense mesh in interesting areas and a coarse mesh in uninteresting areas. This is the setting studied by Chew [117]. He describes a meshing algorithm that allows the user to define a function that determines whether a triangle of the mesh is fine enough. The angles of the triangles produced by his algorithm are between 30° and 120°. Another nice aspect of his work is that the algorithm not only deals with planar regions, but also with regions on surface patches.

A technique called *edge insertion* can also give optimal triangulations for various criteria [66]. The idea is to improve a triangulation in steps by adding an edge, removing the intersected edges, and retriangulating the polygonal regions that appear optimally. It works for certain 'minmax' type of criteria, like minimizing the maximum angle in the triangulation, and minimizing the maximum slope when the triangulation represents a piecewise linear surface.

14.5 Exercises

14.1 In Figure 14.2 a uniform and a non-uniform mesh are shown for a unit square in the top left corner of square domain of size 16. Consider similar meshes in squares of larger sizes $U = 2^j$ for an integer j. Express the number of triangles in the mesh for both meshes in terms of j.

14.2 Suppose a triangular mesh is needed inside a rectangle whose sides have length 1 and length $k > 1$. Steiner points may not be used on the sides, but they may be used inside the rectangle. Also assume that all triangles must have angles between 30° and 90°. Is it always possible to create a triangular mesh with these properties? Suppose it is possible to create a mesh for a particular input, what is the minimum number of Steiner points needed?

14.3 All triangles produced by the meshing algorithm of this chapter are non-obtuse, that is, they do not have angles larger than 90°. Prove that if a triangulation of a set P of points in the plane contains only non-obtuse triangles, then it must be a Delaunay triangulation of P.

14.4 Let P be a set of point in 3-dimensional space. Describe an algorithm to construct an octree on P. (An octree is the 3-dimensional variant of the quadtree.)

14.5 It is possible to reduce the size of a quadtree of depth d for a set of points (with real coordinates) inside a square from $O((d+1)n)$ to $O(n)$. The idea is to discard any node v that has only one child under which points are stored. The node is discarded by replacing the pointer from the parent of v to v by the pointer from the parent to the only interesting child of v. Prove that the resulting tree has linear size. Can you also improve upon the $O((d+1)n)$ construction time?

14.6 In this chapter we called a quadtree balanced if two adjacent squares of the quadtree subdivision differ by no more than a factor two in size. To save a constant factor in the number of extra nodes needed to balance a quadtree, we could weaken the balance condition by allowing adjacent squares to differ by a factor of four in size. Can you still complete such a weakly balanced quadtree subdivision to a conforming mesh such that all angles are between 45° and 90° by using only $O(1)$ triangles per square?

14.7 Suppose we make the balancing condition for quadtrees more severe: we no longer allow adjacent squares to differ by a factor two in size, but we require them to have exactly the same size. Is the number of nodes in the new balanced version still linear in the number of nodes of the original quadtree? If not, can you say anything about this number?

14.8 The algorithm to construct a balanced quadtree had two phases: a normal quadtree was constructed, which was then balanced in a postprocessing step. It is also possible to construct a balanced quadtree without first constructing the unbalanced version. To this end we maintain the current quadtree subdivision in a doubly-connected edge list during the quadtree construction, and whenever we split a square we check whether any neighbors have to be split. Describe the algorithm in detail and analyze its running time.

14.9 One of the steps in Algorithm GENERATEMESH is to construct a doubly-connected edge list for the quadtree subdivision of a given quadtree. Describe an algorithm for this step, and analyze its running time.

14.10 A quadtree can also be used to store a subdivision for efficient point location. The idea is to keep splitting a bounding square of the subdivision until all leaf nodes correspond to squares that contain at most one vertex and only edges incident to that vertex, or no vertex and at most one edge.

a. Since a vertex can be incident to many edges, we need an additional data structure at the quadtree leaves storing vertices. Which data structure would you use?
b. Describe the algorithm for constructing the point location data structure in detail, and analyze its running time.
c. Describe the query algorithm in detail, and analyze its running time.

14.11 Quadtrees are often used to store pixel images. In this case the initial square is exactly the size of the image (which is assumed to be a $2^k \times 2^k$

grid for some integer k). A square is split into subsquares if not all pixels inside have the same intensity.

Prove a bound on the complexity of the quadtree subdivision. *Hint:* This is similar to the bound we proved on the size of the quadtree mesh.

14.12 Suppose we have quadtrees on pixel images I_1 and I_2 (see the previous exercise). Both images have size $2^k \times 2^k$, and contain only two intensities, 0 and 1. Give algorithms for Boolean operations on these images, that is, give algorithms to compute a quadtree for $I_1 \vee I_2$ and $I_1 \wedge I_2$. (Here $I_1 \vee I_2$ is the $2^k \times 2^k$ image where pixel (i, j) has intensity 1 if and only if (i, j) has intensity 1 in image I_1 or in image I_2. The image $I_1 \wedge I_2$ is defined similarly.)

14.13 Quadtrees can be used to perform range queries. Describe an algorithm for querying a quadtree on a set P of points with a query region R. Analyze the worst-case query time for the case where R is a rectangle, and for the case where R is a half-plane bounded by a vertical line.

14.14 In this chapter we studied quadtrees that store a set of point in the plane. In Chapter 5 we studied two other data structures for storing sets of points in the plane, the kd-tree and the range tree. Discuss the advantages and disadvantages of each of the three structures.