3 Polygon Triangulation

Guarding an Art Gallery

Works of famous painters are not only popular among art lovers, but also among criminals. They are very valuable, easy to transport, and apparently not so difficult to sell. Art galleries therefore have to guard their collections carefully.

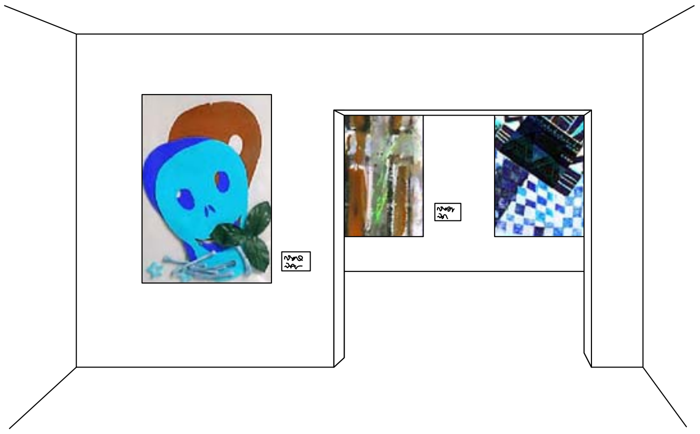
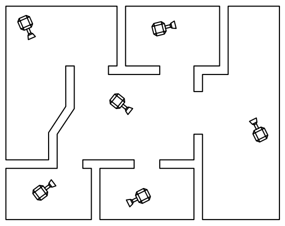
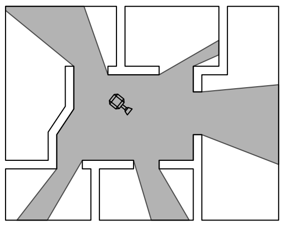


Figure 3.1
An art gallery

During the day the attendants can keep a look-out, but at night this has to be done by video cameras. These cameras are usually hung from the ceiling and they rotate about a vertical axis. The images from the cameras are sent to TV screens in the office of the night watch. Because it is easier to keep an eye on few TV screens rather than on many, the number of cameras should be as small as possible. An additional advantage of a small number of cameras is that the cost of the security system will be lower. On the other hand we cannot have too few cameras, because every part of the gallery must be visible to at least one of them. So we should place the cameras at strategic positions, such that each of them guards a large part of the gallery. This gives rise to what is usually referred to as the *Art Gallery Problem*: how many cameras do we need to guard a given gallery and how do we decide where to place them?



3.1 Guarding and Triangulations



If we want to define the art gallery problem more precisely, we should first formalize the notion of gallery. A gallery is, of course, a 3-dimensional space, but a floor plan gives us enough information to place the cameras. Therefore we model a gallery as a polygonal region in the plane. We further restrict ourselves to regions that are *simple polygons*, that is, regions enclosed by a single closed polygonal chain that does not intersect itself. Thus we do not allow regions with holes. A camera position in the gallery corresponds to a point in the polygon. A camera sees those points in the polygon to which it can be connected with an open segment that lies in the interior of the polygon.

How many cameras do we need to guard a simple polygon? This clearly depends on the polygon at hand: the more complex the polygon, the more cameras are required. We shall therefore express the bound on the number of cameras needed in terms of n, the number of vertices of the polygon. But even when two polygons have the same number of vertices, one can be easier to guard than the other. A convex polygon, for example, can always be guarded with one camera. To be on the safe side we shall look at the worst-case scenario, that is, we shall give a bound that is good for any simple polygon with n vertices. (It would be nice if we could find the minimum number of cameras for the specific polygon we are given, not just a worst-case bound. Unfortunately, the problem of finding the minimum number of cameras for a given polygon is NP-hard.)

Let $\mathcal{P}$ be a simple polygon with n vertices. Because $\mathcal{P}$ may be a complicated shape, it seems difficult to say anything about the number of cameras we need to guard $\mathcal{P}$. Hence, we first decompose $\mathcal{P}$ into pieces that are easy to guard, namely triangles. We do this by drawing *diagonals* between pairs of vertices.

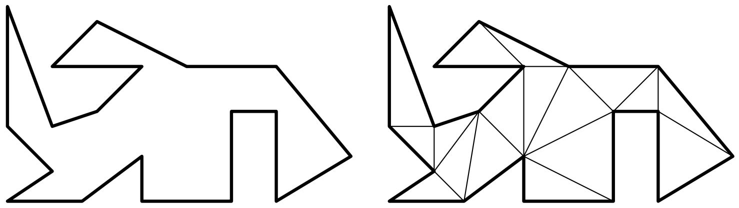
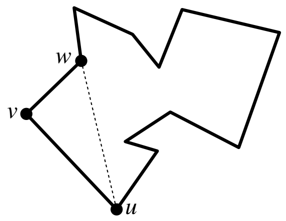


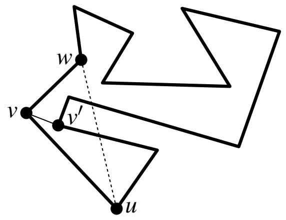
Figure 3.2
A simple polygon and a possible triangulation of it

A diagonal is an open line segment that connects two vertices of $\mathcal{P}$ and lies in the interior of $\mathcal{P}$. A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals is called a *triangulation* of the polygon—see Figure 3.2. (We require that the set of non-intersecting diagonals be maximal to ensure that no triangle has a polygon vertex in the interior of one of its edges. This could happen if the polygon has three consecutive collinear vertices.) Triangulations are usually not unique; the polygon in Figure 3.2, for example, can be triangulated in many different ways. We can guard $\mathcal{P}$ by placing a camera in every triangle of a triangulation $\mathcal{T}_{\mathcal{P}}$ of $\mathcal{P}$. But does a triangulation always exist? And how many triangles can there be in a triangulation? The following theorem answers these questions.

Theorem 3.1 *Every simple polygon admits a triangulation, and any triangulation of a simple polygon with n vertices consists of exactly $n-2$ triangles.*

Proof. We prove this theorem by induction on n. When $n=3$ the polygon itself is a triangle and the theorem is trivially true. Let $n>3$, and assume that the theorem is true for all $m<n$. Let $\mathcal{P}$ be a polygon with n vertices. We first prove the existence of a diagonal in $\mathcal{P}$. Let v be the leftmost vertex of $\mathcal{P}$. (In case of ties, we take the lowest leftmost vertex.) Let u and w be the two neighboring vertices of v on the boundary of $\mathcal{P}$. If the open segment $\overline{uw}$ lies in the interior of $\mathcal{P}$, we have found a diagonal. Otherwise, there are one or more vertices inside the triangle defined by u, v, and w, or on the diagonal $\overline{uw}$. Of those vertices, let v' be the one farthest from the line through u and w. The segment connecting v' to v cannot intersect an edge of $\mathcal{P}$, because such an edge would have an endpoint inside the triangle that is farther from the line through u and w, contradicting the definition of v'. Hence, $\overline{vv'}$ is a diagonal.



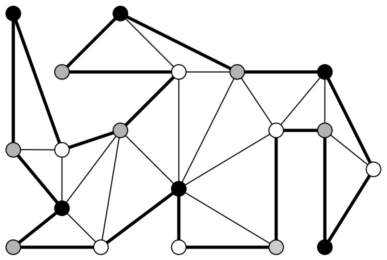


So a diagonal exists. Any diagonal cuts $\mathcal{P}$ into two simple subpolygons $\mathcal{P}_1$ and $\mathcal{P}_2$. Let m_1 be the number of vertices of $\mathcal{P}_1$ and m_2 the number of vertices of $\mathcal{P}_2$. Both m_1 and m_2 must be smaller than n, so by induction $\mathcal{P}_1$ and $\mathcal{P}_2$ can be triangulated. Hence, $\mathcal{P}$ can be triangulated as well.

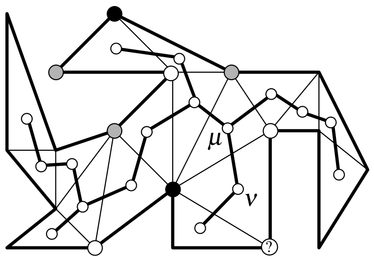
It remains to prove that any triangulation of $\mathcal{P}$ consists of $n-2$ triangles. To this end, consider an arbitrary diagonal in some triangulation $\mathcal{T}_{\mathcal{P}}$. This diagonal cuts $\mathcal{P}$ into two subpolygons with m_1 and m_2 vertices, respectively. Every vertex of $\mathcal{P}$ occurs in exactly one of the two subpolygons, except for the vertices defining the diagonal, which occur in both subpolygons. Hence, $m_1+m_2=n+2$. By induction, any triangulation of $\mathcal{P}_i$ consists of m_i-2 triangles, which implies that $\mathcal{T}_{\mathcal{P}}$ consists of $(m_1-2)+(m_2-2)=n-2$ triangles. $\square$

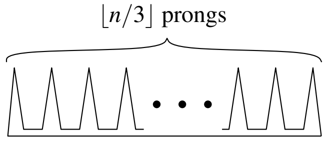
Theorem 3.1 implies that any simple polygon with n vertices can be guarded with $n-2$ cameras. But placing a camera inside every triangle seems overkill. A camera placed on a diagonal, for example, will guard two triangles, so by placing the cameras on well-chosen diagonals we might be able to reduce the number of cameras to roughly $n/2$. Placing cameras at vertices seems even better, because a vertex can be incident to many triangles, and a camera at that vertex guards all of them. This suggests the following approach.

Let $\mathcal{T}_{\mathcal{P}}$ be a triangulation of $\mathcal{P}$. Select a subset of the vertices of $\mathcal{P}$, such that any triangle in $\mathcal{T}_{\mathcal{P}}$ has at least one selected vertex, and place the cameras at the selected vertices. To find such a subset we assign each vertex of $\mathcal{P}$ a color: white, gray, or black. The coloring will be such that any two vertices connected by an edge or a diagonal have different colors. This is called a *3-coloring* of a triangulated polygon. In a 3-coloring of a triangulated polygon, every triangle has a white, a gray, and a black vertex. Hence, if we place cameras at all gray vertices, say, we have guarded the whole polygon. By choosing the smallest color class to place the cameras, we can guard $\mathcal{P}$ using at most $\lfloor n/3 \rfloor$ cameras.



But does a 3-coloring always exist? The answer is yes. To see this, we look at what is called the *dual graph* of $\mathcal{T}_{\mathcal{P}}$. This graph $\mathcal{G}(\mathcal{T}_{\mathcal{P}})$ has a node for every triangle in $\mathcal{T}_{\mathcal{P}}$. We denote the triangle corresponding to a node ν by $t(\nu)$. There is an arc between two nodes ν and μ if $t(\nu)$ and $t(\mu)$ share a diagonal. The arcs in $\mathcal{G}(\mathcal{T}_\mathcal{P})$ correspond to diagonals in $\mathcal{T}_\mathcal{P}$. Because any diagonal cuts $\mathcal{P}$ into two, the removal of an edge from $\mathcal{G}(\mathcal{T}_\mathcal{P})$ splits the graph into two. Hence, $\mathcal{G}(\mathcal{T}_\mathcal{P})$ is a tree. (Notice that this is not true for a polygon with holes.) This means that we can find a 3-coloring using a simple graph traversal, such as depth first search. Next we describe how to do this. While we do the depth first search, we maintain the following invariant: all vertices of the already encountered triangles have been colored white, gray, or black, and no two connected vertices have received the same color. The invariant implies that we have computed a valid 3-coloring when all triangles have been encountered. The depth first search can be started from any node of $\mathcal{G}(\mathcal{T}_\mathcal{P})$; the three vertices of the corresponding triangle are colored white, gray, and black. Now suppose that we reach a node ν in $\mathcal{G}$, coming from node μ. Hence, $t(\nu)$ and $t(\mu)$ share a diagonal. Since the vertices of $t(\mu)$ have already been colored, only one vertex of $t(\nu)$ remains to be colored. There is one color left for this vertex, namely the color that is not used for the vertices of the diagonal between $t(\nu)$ and $t(\mu)$. Because $\mathcal{G}(\mathcal{T}_\mathcal{P})$ is a tree, the other nodes adjacent to ν have not been visited yet, and we still have the freedom to give the vertex the remaining color.





We conclude that a triangulated simple polygon can always be 3-colored. As a result, any simple polygon can be guarded with $\lfloor n/3 \rfloor$ cameras. But perhaps we can do even better. After all, a camera placed at a vertex may guard more than just the incident triangles. Unfortunately, for any n there are simple polygons that require $\lfloor n/3 \rfloor$ cameras. An example is a comb-shaped polygon with a long horizontal base edge and $\lfloor n/3 \rfloor$ prongs made of two edges each. The prongs are connected by horizontal edges. The construction can be made such that there is no position in the polygon from which a camera can look into two prongs of the comb simultaneously. So we cannot hope for a strategy that always produces less than $\lfloor n/3 \rfloor$ cameras. In other words, the 3-coloring approach is optimal in the worst case.

We just proved the Art Gallery Theorem, a classical result from combinatorial geometry.

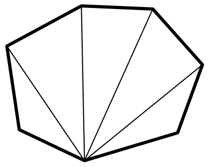
Theorem 3.2 (Art Gallery Theorem) *For a simple polygon with n vertices, $\lfloor n/3 \rfloor$ cameras are occasionally necessary and always sufficient to have every point in the polygon visible from at least one of the cameras.*

Now we know that $\lfloor n/3 \rfloor$ cameras are always sufficient. But we don't have an efficient algorithm to compute the camera positions yet. What we need is a fast algorithm for triangulating a simple polygon. The algorithm should deliver a suitable representation of the triangulation—a doubly-connected edge list, for instance—so that we can step in constant time from a triangle to its neighbors. Given such a representation, we can compute a set of at most $\lfloor n/3 \rfloor$ camera positions in linear time with the method described above: use depth first search on the dual graph to compute a 3-coloring and take the smallest color class to place the cameras. In the coming sections we describe how to compute a triangulation in $O(n \log n)$ time. Anticipating this, we already state the final result about guarding a polygon.

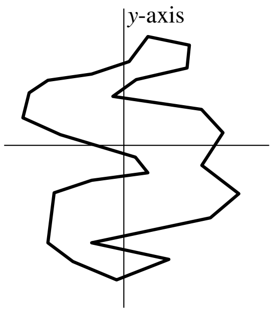
Theorem 3.3 *Let $\mathcal{P}$ be a simple polygon with n vertices. A set of $\lfloor n/3 \rfloor$ camera positions in $\mathcal{P}$ such that any point inside $\mathcal{P}$ is visible from at least one of the cameras can be computed in $O(n \log n)$ time.*

3.2 Partitioning a Polygon into Monotone Pieces

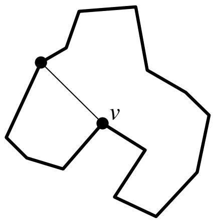
Let $\mathcal{P}$ be a simple polygon with n vertices. We saw in Theorem 3.1 that a triangulation of $\mathcal{P}$ always exists. The proof of that theorem is constructive and leads to a recursive triangulation algorithm: find a diagonal and triangulate the two resulting subpolygons recursively. To find the diagonal we take the leftmost vertex of $\mathcal{P}$ and try to connect its two neighbors u and w; if this fails we connect v to the vertex farthest from $\overline{uw}$ inside the triangle defined by u, v, and w. This way it takes linear time to find a diagonal. This diagonal may split $\mathcal{P}$ into a triangle and a polygon with $n-1$ vertices. Indeed, if we succeed to connect u and w this will always be the case. As a consequence, the triangulation algorithm will take quadratic time in the worst case. Can we do better? For some classes of polygons we surely can. Convex polygons, for instance, are easy: Pick one vertex of the polygon and draw diagonals from this vertex to all other vertices except its neighbors. This takes only linear time. So a possible approach to triangulate a non-convex polygon would be to first decompose $\mathcal{P}$ into convex pieces, and then triangulate the pieces. Unfortunately, it is as difficult to partition a polygon into convex pieces as it is to triangulate it. Therefore we shall decompose $\mathcal{P}$ into so-called monotone pieces, which turns out to be a lot easier.



A simple polygon is called *monotone with respect to a line ℓ* if for any line ℓ' perpendicular to ℓ the intersection of the polygon with ℓ' is connected. In other words, the intersection should be a line segment, a point, or empty. A polygon that is monotone with respect to the y-axis is called *y-monotone*. The following property is characteristic for y-monotone polygons: if we walk from a topmost to a bottommost vertex along the left (or the right) boundary chain, then we always move downwards or horizontally, never upwards.



Our strategy to triangulate the polygon $\mathcal{P}$ is to first partition $\mathcal{P}$ into y-monotone pieces, and then triangulate the pieces. We can partition a polygon into monotone pieces as follows. Imagine walking from the topmost vertex of $\mathcal{P}$ to the bottommost vertex on the left or right boundary chain. A vertex where the direction in which we walk switches from downward to upward or from upward to downward is called a *turn vertex*. To partition $\mathcal{P}$ into y-monotone pieces we should get rid of these turn vertices. This can be done by adding diagonals. If at a turn vertex v both incident edges go down and the interior of the polygon locally lies above v, then we must choose a diagonal that goes up from v. The diagonal splits the polygon into two. The vertex v will appear in both pieces. Moreover, in both pieces v has an edge going down (namely on original edge of $\mathcal{P}$) and an edge going up (the diagonal). Hence, v cannot be a turn vertex anymore in either of them. If both incident edges of a turn vertex go up and



the interior locally lies below it, we have to choose a diagonal that goes down. Apparently there are different types of turn vertices. Let's make this more precise.

If we want to define the different types of turn vertices carefully, we should pay special attention to vertices with equal y-coordinate. We do this by defining the notions of "below" and "above" as follows: a point p is below another point q if $p_y < q_y$ or $p_y = q_y$ and $p_x > q_x$, and p is above q if $p_y > q_y$ or $p_y = q_y$ and $p_x < q_x$. (You can imagine rotating the plane slightly in clockwise direction with respect to the coordinate system, such that no two points have the same y-coordinate; the above/below relation we just defined is the same as the above/below relation in this slightly rotated plane.)

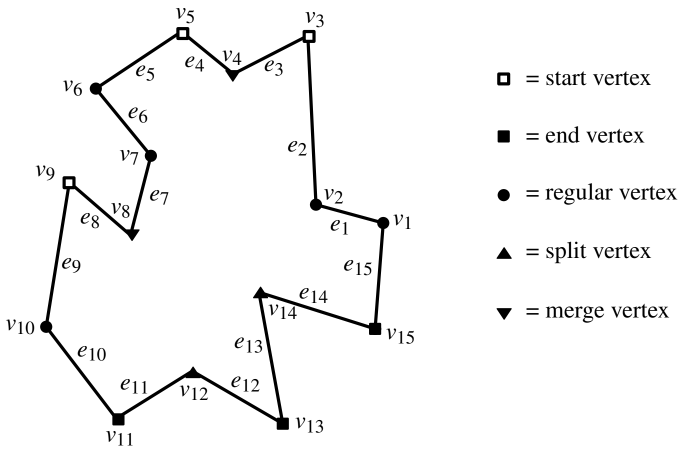


Figure 3.3
Five types of vertices

We distinguish five types of vertices in $\mathcal{P}$—see Figure 3.3. Four of these types are turn vertices: start vertices, split vertices, end vertices, and merge vertices. They are defined as follows. A vertex v is a *start vertex* if its two neighbors lie below it and the interior angle at v is less than π; if the interior angle is greater than π then v is a *split vertex*. (If both neighbors lie below v, then the interior angle cannot be exactly π.) A vertex is an *end vertex* if its two neighbors lie above it and the interior angle at v is less than π; if the interior angle is greater than π then v is a *merge vertex*. The vertices that are not turn vertices are *regular vertices*. Thus a regular vertex has one of its neighbors above it, and the other neighbor below it. These names have been chosen because the algorithm will use a downward plane sweep, maintaining the intersection of the sweep line with the polygon. When the sweep line reaches a split vertex, a component of the intersection splits, when it reaches a merge vertex, two components merge, and so on.

The split and merge vertices are sources of local non-monotonicity. The following, stronger statement is even true.

Lemma 3.4 *A polygon is y-monotone if it has no split vertices or merge vertices.*

Proof. Suppose $\mathcal{P}$ is not y-monotone. We have to prove that $\mathcal{P}$ contains a split or a merge vertex.

Since $\mathcal{P}$ is not monotone, there is a horizontal line ℓ that intersects $\mathcal{P}$ in more than one connected component. We can choose ℓ such that the leftmost component is a segment, not a single point. Let p be the left endpoint of this segment, and let q be the right endpoint. Starting at q, we follow the boundary of $\mathcal{P}$ such that $\mathcal{P}$ lies to the left of the boundary. (This means that we go up from q.) At some point, let's call it r, the boundary will intersect ℓ again. If $r \neq p$, as in Figure 3.4(a), then the highest vertex we encountered while going from q to r must be a split vertex, and we are done.

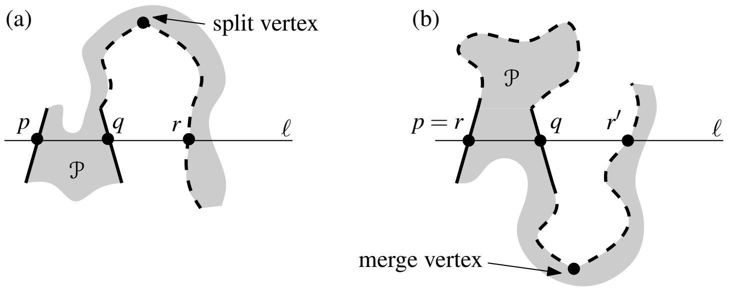


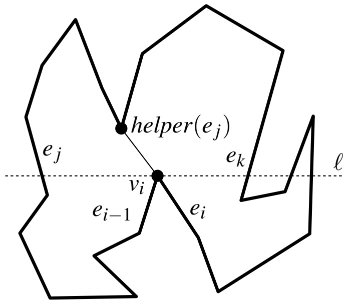
Figure 3.4
Two cases in the proof of Lemma 3.4

If $r = p$, as in Figure 3.4(b), we again follow the boundary of $\mathcal{P}$ starting at q, but this time in the other direction. As before, the boundary will intersect ℓ. Let r' be the point where this happens. We cannot have $r' = p$, because that would mean that the boundary of $\mathcal{P}$ intersects ℓ only twice, contradicting that ℓ intersects $\mathcal{P}$ in more than one component. So we have $r' \neq p$, implying that the lowest vertex we have encountered while going from q to r' must be a merge vertex. $\square$

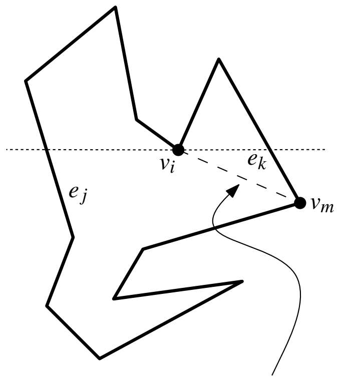
Lemma 3.4 implies that $\mathcal{P}$ has been partitioned into y-monotone pieces once we get rid of its split and merge vertices. We do this by adding a diagonal going upward from each split vertex and a diagonal going downward from each merge vertex. These diagonals should not intersect each other, of course. Once we have done this, $\mathcal{P}$ has been partitioned into y-monotone pieces.

Let's first see how we can add the diagonals for the split vertices. We use a plane sweep method for this. Let $v_1, v_2, \ldots, v_n$ be a counterclockwise enumeration of the vertices of $\mathcal{P}$. Let $e_1, \ldots, e_n$ be the set of edges of $\mathcal{P}$, where $e_i = \overline{v_i v_{i+1}}$ for $1 \leqslant i < n$ and $e_n = \overline{v_n v_1}$. The plane sweep algorithm moves an imaginary sweep line ℓ downward over the plane. The sweep line halts at certain event points. In our case these will be the vertices of $\mathcal{P}$; no new event points will be created during the sweep. The event points are stored in a event queue $\mathcal{Q}$. The event queue is a priority queue, where the priority of a vertex is its y-coordinate. If two vertices have the same y-coordinate then the leftmost one has higher priority. This way the next event to be handled can be found in $O(\log n)$ time. (Because no new events are generated during the sweep, we could also sort the vertices on y-coordinate before the sweep, and then use the sorted list to find the next event in $O(1)$ time.)

The goal of the sweep is to add diagonals from each split vertex to a vertex lying above it. Suppose that the sweep line reaches a split vertex v_i. To which vertex should we connect v_i? A good candidate is a vertex close to v_i, because we can probably connect v_i to this vertex without intersecting any edge of $\mathcal{P}$. Let's make this more precise. Let e_j be the edge immediately to the left of v_i on the sweep line, and let e_k be the edge immediately to the right of v_i on the sweep line. Then we can always connect v_i to the lowest vertex in between e_j and e_k, and above v_i. If there is no such vertex then we can connect v_i to the upper endpoint of e_j or to the upper endpoint of e_k. We call this vertex the *helper* of e_j and denote it by $\mathit{helper}(e_j)$. Formally, $\mathit{helper}(e_j)$ is defined as the lowest vertex above the sweep line such that the horizontal segment connecting the vertex to e_j lies inside $\mathcal{P}$. Note that $\mathit{helper}(e_j)$ can be the upper endpoint of e_j itself.



Now we know how to get rid of split vertices: connect them to the helper of the edge to their left. What about merge vertices? They seem more difficult to get rid of, because they need a diagonal to a vertex that is lower than they are. Since the part of $\mathcal{P}$ below the sweep line has not been explored yet, we cannot add such a diagonal when we encounter a merge vertex. Fortunately, this problem is easier than it seems at first sight. Suppose the sweep line reaches a merge vertex v_i. Let e_j and e_k be the edges immediately to the right and to the left of v_i on the sweep line, respectively. Observe that v_i becomes the new helper of e_j when we reach it. We would like to connect v_i to the highest vertex below the sweep line in between e_j and e_k. This is exactly the opposite of what we did for split vertices, which we connected to the lowest vertex above the sweep line in between e_j and e_k. This is not surprising: merge vertices are split vertices upside down. Of course we don't know the highest vertex below the sweep line when we reach v_i. But it is easy to find later on: when we reach a vertex v_m that replaces v_i as the helper of e_j, then this is the vertex we are looking for. So whenever we replace the helper of some edge, we check whether the old helper is a merge vertex and, if so, we add the diagonal between the old helper and the new one. This diagonal is always added when the new helper is a split vertex, to get rid of the split vertex. If the old helper was a merge vertex, we thus get rid of a split vertex and a merge vertex with the same diagonal. It can also happen that the helper of e_j is not replaced anymore below v_i. In this case we can connect v_i to the lower endpoint of e_j.



diagonal will be added when the sweep line reaches v_m

In the approach above, we need to find the edge to the left of each vertex. Therefore we store the edges of $\mathcal{P}$ intersecting the sweep line in the leaves of a dynamic binary search tree $\mathcal{T}$. The left-to-right order of the leaves of $\mathcal{T}$ corresponds to the left-to-right order of the edges. Because we are only interested in edges to the left of split and merge vertices we only need to store edges in $\mathcal{T}$ that have the interior of $\mathcal{P}$ to their right. With each edge in $\mathcal{T}$ we store its helper. The tree $\mathcal{T}$ and the helpers stored with the edges form the status of the sweep line algorithm. The status changes as the sweep line moves: edges start or stop intersecting the sweep line, and the helper of an edge may be replaced.

The algorithm partitions $\mathcal{P}$ into subpolygons that have to be processed further in a later stage. To have easy access to these subpolygons we shall store the subdivision induced by $\mathcal{P}$ and the added diagonals in a doubly-connected edge list $\mathcal{D}$. We assume that $\mathcal{P}$ is initially specified as a doubly-connected edge list; if $\mathcal{P}$ is given in another form—by a counterclockwise list of its vertices, for example—we first construct a doubly-connected edge list for $\mathcal{P}$. The diagonals computed for the split and merge vertices are added to the doubly-connected edge list. To access the doubly-connected edge list we use cross-pointers between the edges in the status structure and the corresponding edges in the doubly-connected edge list. Adding a diagonal can then be done in constant time with some simple pointer manipulations. The global algorithm is now as follows.

Algorithm MAKEMONOTONE($\mathcal{P}$)
Input. A simple polygon $\mathcal{P}$ stored in a doubly-connected edge list $\mathcal{D}$.
Output. A partitioning of $\mathcal{P}$ into monotone subpolygons, stored in $\mathcal{D}$.

```
1.  Construct a priority queue $\mathcal{Q}$ on the vertices of $\mathcal{P}$, using their $y$-coordinates
    as priority. If two points have the same $y$-coordinate, the one with smaller
    $x$-coordinate has higher priority.
2.  Initialize an empty binary search tree $\mathcal{T}$.
3.  while $\mathcal{Q}$ is not empty
4.     do Remove the vertex $v_i$ with the highest priority from $\mathcal{Q}$.
5.        Call the appropriate procedure to handle the vertex, depending on
          its type.
```

We next describe more precisely how to handle the event points. You should first read these algorithms without thinking about degenerate cases, and check only later that they are also correct in degenerate cases. (To this end you should give an appropriate meaning to "directly left of" in line 1 of HANDLESPLITVERTEX and line 2 of HANDLEMERGEVERTEX.) There are always two things we must do when we handle a vertex. First, we must check whether we have to add a diagonal. This is always the case for a split vertex, and also when we replace the helper of an edge and the previous helper was a merge vertex. Second, we must update the information in the status structure $\mathcal{T}$. The precise algorithms for each type of event are given below. You can use the example figure on the next page to see what happens in each of the different cases.

HANDLESTARTVERTEX(v_i)

```
1.  Insert $e_i$ in $\mathcal{T}$ and set $helper(e_i)$ to $v_i$.
```

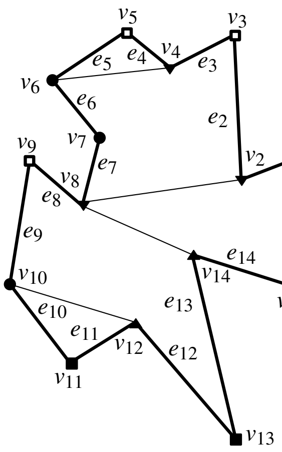
At the start vertex v_5 in the example figure, for instance, we insert e_5 into the tree $\mathcal{T}$.

HANDLEENDVERTEX(v_i)

```
1.  if $helper(e_{i-1})$ is a merge vertex
2.     then Insert the diagonal connecting $v_i$ to $helper(e_{i-1})$ in $\mathcal{D}$.
3.  Delete $e_{i-1}$ from $\mathcal{T}$.
```

In the running example, when we reach end vertex v_{15}, the helper of the edge e_{14} is v_{14}. v_{14} is not a merge vertex, so we don't need to insert a diagonal.

```
HANDLESPLITVERTEX($v_i$)
1.  Search in $\mathcal{T}$ to find the edge $e_j$ directly left of $v_i$.
2.  Insert the diagonal connecting $v_i$ to $helper(e_j)$ in $\mathcal{D}$.
3.  $helper(e_j) \leftarrow v_i$
4.  Insert $e_i$ in $\mathcal{T}$ and set $helper(e_i)$ to $v_i$.
```



For split vertex v_{14} in our example, e_9 is the edge to the left. Its helper is v_8, so we add a diagonal from v_{14} to v_8.

```
HANDLEMERGEVERTEX($v_i$)
1.  if $helper(e_{i-1})$ is a merge vertex
2.     then Insert the diagonal connecting $v_i$ to $helper(e_{i-1})$ in $\mathcal{D}$.
3.  Delete $e_{i-1}$ from $\mathcal{T}$.
4.  Search in $\mathcal{T}$ to find the edge $e_j$ directly left of $v_i$.
5.  if $helper(e_j)$ is a merge vertex
6.     then Insert the diagonal connecting $v_i$ to $helper(e_j)$ in $\mathcal{D}$.
7.  $helper(e_j) \leftarrow v_i$
```

For the merge vertex v_8 in our example, the helper v_2 of edge e_7 is a merge vertex, so we add a diagonal from v_8 to v_2.

The only routine that remains to be described is the one to handle a regular vertex. The actions we must take at a regular vertex depend on whether $\mathcal{P}$ lies locally to its left or to its right.

```
HANDLEREGULARVERTEX($v_i$)
1.  if the interior of $\mathcal{P}$ lies to the right of $v_i$
2.     then if $helper(e_{i-1})$ is a merge vertex
3.              then Insert the diagonal connecting $v_i$ to $helper(e_{i-1})$ in $\mathcal{D}$.
4.           Delete $e_{i-1}$ from $\mathcal{T}$.
5.           Insert $e_i$ in $\mathcal{T}$ and set $helper(e_i)$ to $v_i$.
6.     else  Search in $\mathcal{T}$ to find the edge $e_j$ directly left of $v_i$.
7.           if $helper(e_j)$ is a merge vertex
8.              then Insert the diagonal connecting $v_i$ to $helper(e_j)$ in $\mathcal{D}$.
9.           $helper(e_j) \leftarrow v_i$
```

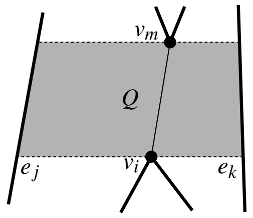
For instance, at the regular vertex v_6 in our example, we add a diagonal from v_6 to v_4.

It remains to prove that MAKEMONOTONE correctly partitions $\mathcal{P}$ into monotone pieces.

Lemma 3.5 *Algorithm* MAKEMONOTONE *adds a set of non-intersecting diagonals that partitions* $\mathcal{P}$ *into monotone subpolygons.*

Proof. It is easy to see that the pieces into which $\mathcal{P}$ is partitioned contain no split or merge vertices. Hence, they are monotone by Lemma 3.4. It remains to prove that the added segments are valid diagonals (that is, that they don't intersect the edges of $\mathcal{P}$) and that they don't intersect each other. To this end we will show that when a segment is added, it intersects neither an edge of $\mathcal{P}$ nor any of the previously added segments. We shall prove this for the segment added in HANDLESPLITVERTEX; the proof for the segments added inHANDLEENDVERTEX, HANDLEREGULARVERTEX, and HANDLEMERGE-VERTEX is similar. We assume that no two vertices have the same y-coordinate; the extension to the general case is fairly straightforward.

Consider a segment $\overline{v_m v_i}$ that is added by HANDLESPLITVERTEX when v_i is reached. Let e_j be the edge to the left of v_i, and let e_k be the edge to the right of v_i. Thus $helper(e_j) = v_m$ when we reach v_i.



We first argue that $\overline{v_m v_i}$ does not intersect an edge of $\mathcal{P}$. To see this, consider the quadrilateral Q bounded by the horizontal lines through v_m and v_i, and by e_j and e_k. There are no vertices of $\mathcal{P}$ inside Q, otherwise v_m would not be the helper of e_j. Now suppose there would be an edge of $\mathcal{P}$ intersecting $\overline{v_m v_i}$. Since the edge cannot have an endpoint inside Q and polygon edges do not intersect each other, it would have to intersect the horizontal segment connecting v_m to e_j or the horizontal segment connecting v_i to e_j. Both are impossible, since for both v_m and v_i, the edge e_j lies immediately to the left. Hence, no edge of $\mathcal{P}$ can intersect $\overline{v_m v_i}$.

Now consider a previously added diagonal. Since there are no vertices of $\mathcal{P}$ inside Q, and any previously added diagonal must have both of its endpoints above v_i, it cannot intersect $\overline{v_m v_i}$. $\square$

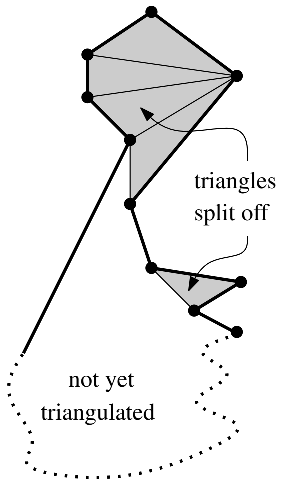
We now analyze the running time of the algorithm. Constructing the priority queue $\mathcal{Q}$ takes linear time and initializing $\mathcal{T}$ takes constant time. To handle an event during the sweep, we perform one operation on $\mathcal{Q}$, at most one query, one insertion, and one deletion on $\mathcal{T}$, and we insert at most two diagonals into $\mathcal{D}$. Priority queues and balanced search trees allow for queries and updates in $O(\log n)$ time, and the insertion of a diagonal into $\mathcal{D}$ takes $O(1)$ time. Hence, handling an event takes $O(\log n)$ time, and the total algorithm runs in $O(n \log n)$ time. The amount of storage used by the algorithm is clearly linear: every vertex is stored at most once in $\mathcal{Q}$, and every edge is stored at most once in $\mathcal{T}$. Together with Lemma 3.5 this implies the following theorem.

Theorem 3.6 *A simple polygon with n vertices can be partitioned into y-monotone polygons in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.*

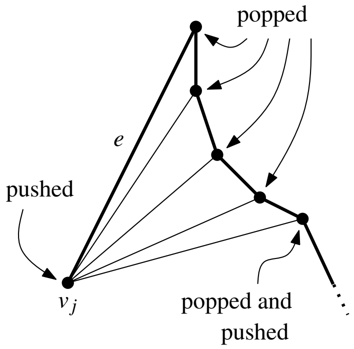
3.3 Triangulating a Monotone Polygon

We have just seen how to partition a simple polygon into y-monotone pieces in $O(n \log n)$ time. In itself this is not very interesting. But in this section we show that monotone polygons can be triangulated in linear time. Together these results imply that any simple polygon can be triangulated in $O(n \log n)$ time, a nice improvement over the quadratic time algorithm that we sketched at the beginning of the previous section.

Let $\mathcal{P}$ be a y-monotone polygon with n vertices. For the moment we assume that $\mathcal{P}$ is *strictly y-monotone*, that is, we assume that $\mathcal{P}$ is y-monotone and does not contain horizontal edges. Thus we always go down when we walk on the left or right boundary chain of $\mathcal{P}$ from the highest vertex of $\mathcal{P}$ to the lowest one. This is the property that makes triangulating a monotone polygon easy: we can work our way through $\mathcal{P}$ from top to bottom on both chains, adding diagonals whenever this is possible. Next we describe the details of this greedy triangulation algorithm.



The algorithm handles the vertices in order of decreasing y-coordinate. If two vertices have the same y-coordinate, then the leftmost one is handled first. The algorithm requires a stack $\mathcal{S}$ as auxiliary data structure. Initially the stack is empty; later it contains the vertices of $\mathcal{P}$ that have been encountered but may still need more diagonals. When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible. These diagonals split off triangles from $\mathcal{P}$. The vertices that have been handled but not split off—the vertices on the stack—are on the boundary of the part of $\mathcal{P}$ that still needs to be triangulated. The lowest of these vertices, which is the one encountered last, is on top of the stack, the second lowest is second on the stack, and so on. The part of $\mathcal{P}$ that still needs to be triangulated, and lies above the last vertex that has been encountered so far, has a particular shape: it looks like a funnel turned upside down. One boundary of the funnel consists of a part of a single edge of $\mathcal{P}$, and the other boundary is a chain consisting of reflex vertices, that is, the interior angle at these vertices is at least 180°. Only the highest vertex, which is at the bottom of the stack, is convex. This property remains true after we have handled the next vertex. Hence, it is an invariant of the algorithm.



Now, let's see which diagonals we can add when we handle the next vertex. We distinguish two cases: v_j, the next vertex to be handled, lies on the same chain as the reflex vertices on the stack, or it lies on the opposite chain. If v_j lies on the opposite chain, it must be the lower endpoint of the single edge e bounding the funnel. Due to the shape of the funnel, we can add diagonals from v_j to all vertices currently on the stack, except for the last one (that is, the one at the bottom of the stack); the last vertex on the stack is the upper vertex of e, so it is already connected to v_j. All these vertices are popped from the stack. The untriangulated part of the polygon above v_j is bounded by the diagonal that connects v_j to the vertex previously on top of the stack and the edge of $\mathcal{P}$ extending downward from this vertex, so it looks like a funnel and the invariant is preserved. This vertex and v_j remain part of the not yet triangulated polygon, so they are pushed onto the stack.

The other case is when v_j is on the same chain as the reflex vertices on the stack. This time we may not be able to draw diagonals from v_j to all vertices on the stack. Nevertheless, the ones to which we can connect v_j are all consecutive and they are on top of the stack, so we can proceed as follows. First, pop one vertex from the stack; this vertex is already connected to v_j by an edge of $\mathcal{P}$. Next, pop vertices from the stack and connect them to v_j until we encounter one where this is not possible. Checking whether a diagonal can be drawn from v_j to a vertex v_k on the stack can be done by looking at v_j, v_k, and the previous vertex that was popped. When we find a vertex to which we cannot connect v_j, we push the last vertex that has been popped back onto the stack. This is either the last vertex to which a diagonal was added or, if no diagonals have been added, it is the neighbor of v_j on the boundary of $\mathcal{P}$—see Figure 3.5. After this

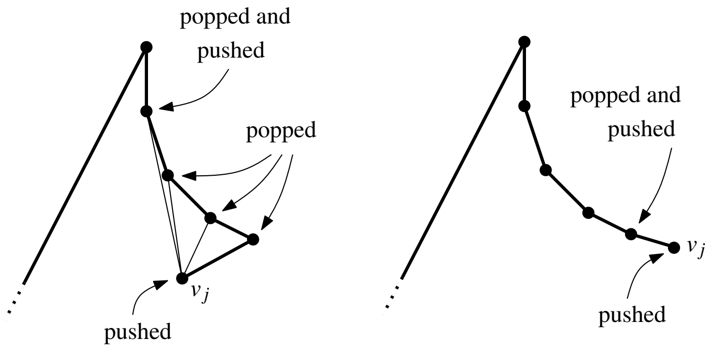


Figure 3.5
Two cases when the next vertex is on the same side as the reflex vertices on the stack

has been done we push v_j onto the stack. In both cases the invariant is restored: one side of the funnel is bounded by a part of a single edge, and the other side is bounded by a chain of reflex vertices. We get the following algorithm. (The algorithm is actually similar to the convex hull algorithm of Chapter 1.)

```
Algorithm TRIANGULATEMONOTONEPOLYGON($\mathcal{P}$)
Input. A strictly $y$-monotone polygon $\mathcal{P}$ stored in a doubly-connected edge
   list $\mathcal{D}$.
Output. A triangulation of $\mathcal{P}$ stored in the doubly-connected edge list $\mathcal{D}$.
1.  Merge the vertices on the left chain and the vertices on the right chain of $\mathcal{P}$
    into one sequence, sorted on decreasing $y$-coordinate. If two vertices have
    the same $y$-coordinate, then the leftmost one comes first. Let $u_1, \dots, u_n$
    denote the sorted sequence.
2.  Initialize an empty stack $\mathcal{S}$, and push $u_1$ and $u_2$ onto it.
3.  for $j \leftarrow 3$ to $n-1$
4.      do if $u_j$ and the vertex on top of $\mathcal{S}$ are on different chains
5.            then Pop all vertices from $\mathcal{S}$.
6.                 Insert into $\mathcal{D}$ a diagonal from $u_j$ to each popped vertex,
                   except the last one.
7.                 Push $u_{j-1}$ and $u_j$ onto $\mathcal{S}$.
8.            else Pop one vertex from $\mathcal{S}$.
9.                 Pop the other vertices from $\mathcal{S}$ as long as the diagonals from
                   $u_j$ to them are inside $\mathcal{P}$. Insert these diagonals into $\mathcal{D}$. Push
                   the last vertex that has been popped back onto $\mathcal{S}$.
10.                Push $u_j$ onto $\mathcal{S}$.
11. Add diagonals from $u_n$ to all stack vertices except the first and the last one.
```

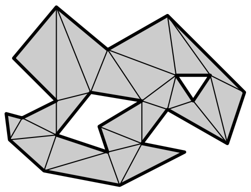
How much time does the algorithm take? Step 1 takes linear time and Step 2 takes constant time. The **for**-loop is executed $n-3$ times, and one execution may take linear time. But at every execution of the **for**-loop at most two vertices are pushed. Hence, the total number of pushes, including the two in Step 2, is bounded by $2n-4$. Because the number of pops cannot exceed the number of pushes, the total time for all executions of the **for**-loop is $O(n)$. The last step of the algorithm also takes at most linear time, so the total algorithm runs in $O(n)$ time.

Theorem 3.7 *A strictly y-monotone polygon with n vertices can be triangulated in linear time.*

We wanted a triangulation algorithm for monotone polygons as a subroutine for triangulating arbitrary simple polygons. The idea was to first decompose a polygon into monotone pieces and then to triangulate these pieces. It seems that we have all the ingredients we need. There is one problem, however: in this section we have assumed that the input is a *strictly* y-monotone polygon, whereas the algorithm of the previous section may produce monotone pieces with horizontal edges. Recall that in the previous section we treated vertices with the same y-coordinates from left to right. This had the same effect as a slight rotation of the plane in clockwise direction such that no two vertices are on a horizontal line. It follows that the monotone subpolygons produced by the algorithm of the previous section are strictly monotone in this slightly rotated plane. Hence, the triangulation algorithm of the current section operates correctly if we treat vertices with the same y-coordinate from left to right (which corresponds to working in the rotated plane). So we can combine the two algorithms to obtain a triangulation algorithm that works for any simple polygon.

How much time does the triangulation algorithm take? Decomposing the polygon into monotone pieces takes $O(n \log n)$ time by Theorem 3.6. In the second stage we triangulate each of the monotone pieces with the linear-time algorithm of this section. Since the sum of the number of vertices of the pieces is $O(n)$, the second stage takes $O(n)$ time in total. We get the following result.

Theorem 3.8 *A simple polygon with n vertices can be triangulated in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.*



We have seen how to triangulate simple polygons. But what about polygons with holes, can they also be triangulated easily? The answer is yes. In fact, the algorithm we have seen also works for polygons with holes: nowhere in the algorithm for splitting a polygon into monotone pieces did we use the fact that the polygon was simple. It even works in a more general setting: Suppose we have a planar subdivision $\mathcal{S}$ and we want to triangulate that subdivision. More precisely, if B is a bounding box containing all edges of $\mathcal{S}$ in its interior, we want to find a maximal set of non-intersecting diagonals—line segments connecting vertices of $\mathcal{S}$ or B that do not intersect the edges of $\mathcal{S}$—that partitions B into triangles. Figure 3.6 shows a triangulated subdivision. The edges of the subdivisions and of the bounding box are shown bold. To compute such a triangulation we can use the algorithm of this chapter: first split the subdivision into monotone pieces, and then triangulate the pieces. This leads to the following theorem.

Theorem 3.9 *A planar subdivision with n vertices in total can be triangulated in $O(n \log n)$ time with an algorithm that uses $O(n)$ storage.*

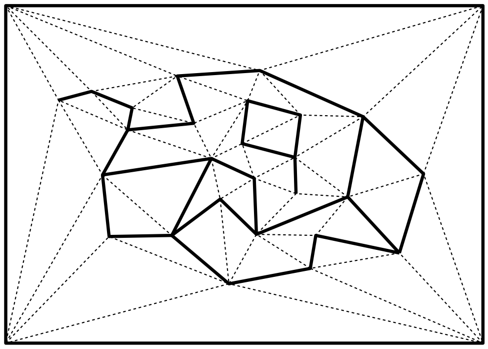


Figure 3.6
A triangulated subdivision

3.4 Notes and Comments

The Art Gallery Problem was posed in 1973 by Victor Klee in a conversation with Vasek Chvátal. In 1975 Chvátal [128] gave the first proof that $\lfloor n/3 \rfloor$ cameras are always sufficient and sometimes necessary; a result that became known as the *Art Gallery Theorem* or the *Watchman Theorem*. Chvátal's proof is quite complicated. The much simpler proof presented in this chapter was discovered by Fisk [178]. His proof is based on the *Two Ears Theorem* by Meisters [277], from which the 3-colorability of the graph that is a triangulation of a simple polygon follows easily. The algorithmic problem of finding the minimum number of guards for a given simple polygon was shown to be NP-hard by Aggarwal [10] and Lee and Lin [246]. The book by O'Rourke [298] and the overview by Shermer [355] contain an extensive treatment of the Art Gallery Problem and numerous variations.

A decomposition of a polygon, or any other region, into simple pieces is useful in many problems. Often the simple pieces are triangles, in which case we call the decomposition a triangulation, but sometimes other shapes such as quadrilaterals or trapezoids are used—see also Chapters 6, 9, and 14. We only discuss the results on triangulating polygons here. The linear time algorithm to triangulate a monotone polygon described in this chapter was given by Garey et al. [188], and the plane sweep algorithm to partition a polygon into monotone pieces is due to Lee and Preparata [250]. Avis and Toussaint [32] and Chazelle [85] described other algorithms for triangulating a simple polygon in $O(n \log n)$ time.

For a long time one of the main open problems in computational geometry was whether simple polygons can be triangulated in $o(n \log n)$ time. (For triangulating subdivisions with holes there is an $\Omega(n \log n)$ lower bound.) In this chapter we have seen that this is indeed the case for monotone polygons. Linear-time triangulation algorithms were also found for other special classes of polygons [108, 109, 170, 184, 214] but the problem for general simple polygons remained open for a number of years. In 1988 Tarjan and Van Wyk [368] broke the $O(n \log n)$ barrier by presenting an $O(n \log \log n)$ algorithm. Their algorithm was later simplified by Kirkpatrick et al. [237]. Randomization—an approach used in Chapters 4, 6, 9, and 11—proved to be a good tool in developing even faster algorithms: Clarkson et al. [134], Devillers [141], and Seidel [345] presented algorithms with $O(n\log^* n)$ running time, where $\log^* n$ is the iterated logarithm of n (being the number of times you can take the logarithm before the result is smaller than 1). These algorithms are not only slightly faster than the $O(n\log\log n)$ algorithm, but also simpler. Seidel's algorithm is closely related to the algorithm for constructing a trapezoidal decomposition of a planar subdivision described in Chapter 6. However, the question whether a simple polygon can be triangulated in linear time was still open. In 1990 this problem was finally settled by Chazelle [92, 94], who gave a (quite complicated) deterministic linear time algorithm. A randomized linear time algorithm was developed later by Amato et al. [15].

The 3-dimensional equivalent to the polygon triangulation problem is this: decompose a given polytope into non-overlapping tetrahedra, where the vertices of the tetrahedra must be vertices of the original polytope. Such a decomposition is called a *tetrahedralization* of the polytope. This problem is much more difficult than the two-dimensional version. In fact, it is not always possible to decompose a polytope into tetrahedra without using additional vertices. Chazelle [86] has shown that for a simple polytope with n vertices, $\Theta(n^2)$ additional vertices may be needed and are always sufficient to obtain a decomposition into tetrahedra. This bound was refined by Chazelle and Palios [110] to $\Theta(n+r^2)$, where r is the number of reflex edges of the polytope. The algorithm to compute the decomposition runs in $O(nr+r^2\log r)$ time. Deciding whether a given simple polytope can be tetrahedralized without additional vertices is NP-complete [330].

3.5 Exercises

3.1 Prove that any polygon admits a triangulation, even if it has holes. Can you say anything about the number of triangles in the triangulation?

3.2 A *rectilinear polygon* is a simple polygon of which all edges are horizontal or vertical. Let $\mathcal{P}$ be a rectilinear polygon with n vertices. Give an example to show that $\lfloor n/4 \rfloor$ cameras are sometimes necessary to guard it.

3.3 Prove or disprove: The dual graph of the triangulation of a monotone polygon is always a chain, that is, any node in this graph has degree at most two.

3.4 Suppose that a simple polygon $\mathcal{P}$ with n vertices is given, together with a set of diagonals that partitions $\mathcal{P}$ into convex quadrilaterals. How many cameras are sufficient to guard $\mathcal{P}$? Why doesn't this contradict the Art Gallery Theorem?

3.5 Give the pseudo-code of the algorithm to compute a 3-coloring of a triangulated simple polygon. The algorithm should run in linear time.

3.6 Give an algorithm that computes in $O(n \log n)$ time a diagonal that splits a simple polygon with n vertices into two simple polygons each with at most $\lfloor 2n/3 \rfloor + 2$ vertices. *Hint:* Use the dual graph of a triangulation.

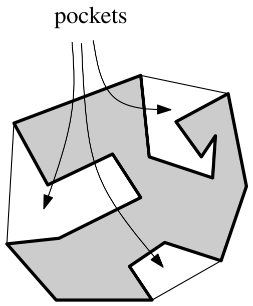
3.7 Let $\mathcal{P}$ be a simple polygon with n vertices, which has been partitioned into monotone pieces. Prove that the sum of the number of vertices of the pieces is $O(n)$.

3.8 The algorithm given in this chapter to partition a simple polygon into monotone pieces constructs a doubly-connected edge list for the partitioned polygon. During the algorithm, new edges are added to the DCEL (namely, diagonals to get rid of split and merge vertices). In general, adding an edge to a DCEL cannot be done in constant time. Discuss why adding an edge may take more than constant time, and argue that in the polygon-partitioning algorithm we can add a diagonal in $O(1)$ time nevertheless.

3.9 Show that if a polygon has $O(1)$ turn vertices, then the algorithm given in this chapter can be made to run in $O(n)$ time.

3.10 Can the algorithm of this chapter also be used to triangulate a set of n points? If so, explain how to do this efficiently.

3.11 Give an efficient algorithm to determine whether a polygon $\mathcal{P}$ with n vertices is monotone with respect to some line, not necessarily a horizontal or vertical one.



3.12 The *pockets* of a simple polygon are the areas outside the polygon, but inside its convex hull. Let $\mathcal{P}_1$ be a simple polygon with m vertices, and assume that a triangulation of $\mathcal{P}_1$ as well as its pockets is given. Let $\mathcal{P}_2$ be a convex polygon with n vertices. Show that the intersection $\mathcal{P}_1 \cap \mathcal{P}_2$ can be computed in $O(m+n)$ time.

3.13 The *stabbing number* of a triangulated simple polygon $\mathcal{P}$ is the maximum number of diagonals intersected by any line segment interior to $\mathcal{P}$. Give an algorithm that computes a triangulation of a convex polygon that has stabbing number $O(\log n)$.

3.14 Given a simple polygon $\mathcal{P}$ with n vertices and a point p inside it, show how to compute the region inside $\mathcal{P}$ that is visible from p.