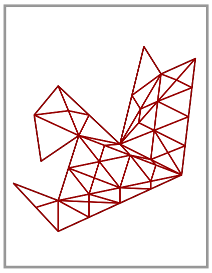


CHAPTER 15

Triangulation: basic graphics algorithms

We now know how to draw parametrized surfaces in a reasonably realistic manner. We do not yet, however, know exactly how to draw even simple regions on surfaces, such as spherical triangles, if we want to take into account the usual phenomena of shading and visibility.

The techniques explained for managing the whole surface suggest that the natural way to proceed is to chop up the region involved into small triangles, and then apply the surface parametrization to those triangles. The problem is thus reduced to that of decomposing an arbitrary 2D polygonal region into small triangles, and that will be the principal topic in this chapter. This is, as one might suspect, a very basic problem in computer graphics, and there are several valid approaches. The one taken here can be found in Chapter 3 of the book by de Berg et al. that I have already mentioned as the source of the algorithm for finding convex hulls in 2D, as well as for dealing with binary space partitions.

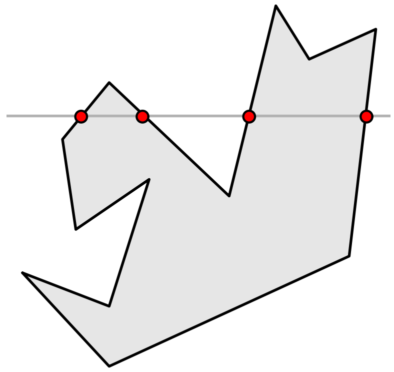
What is to be explained is the most complicated single topic in this whole book. It involves among other things some very basic computational structures we have so far managed to avoid, such as stacks and binary search trees. Implementing these in PostScript is . . . hmmmm . . . *educational*. Intriguing. Bracing. And still, overall, valuable. Using these and other basic data structures is common in sophisticated graphics programs because in order not to have to go backwards in a computation a lot of information has to be stored accessibly as we go along, and this can be tricky.

Practical triangulation involves three steps—decomposition into vertically monotone regions, then into triangles, and then into small triangles. None of these is uninteresting.

15.1. The monotone decomposition

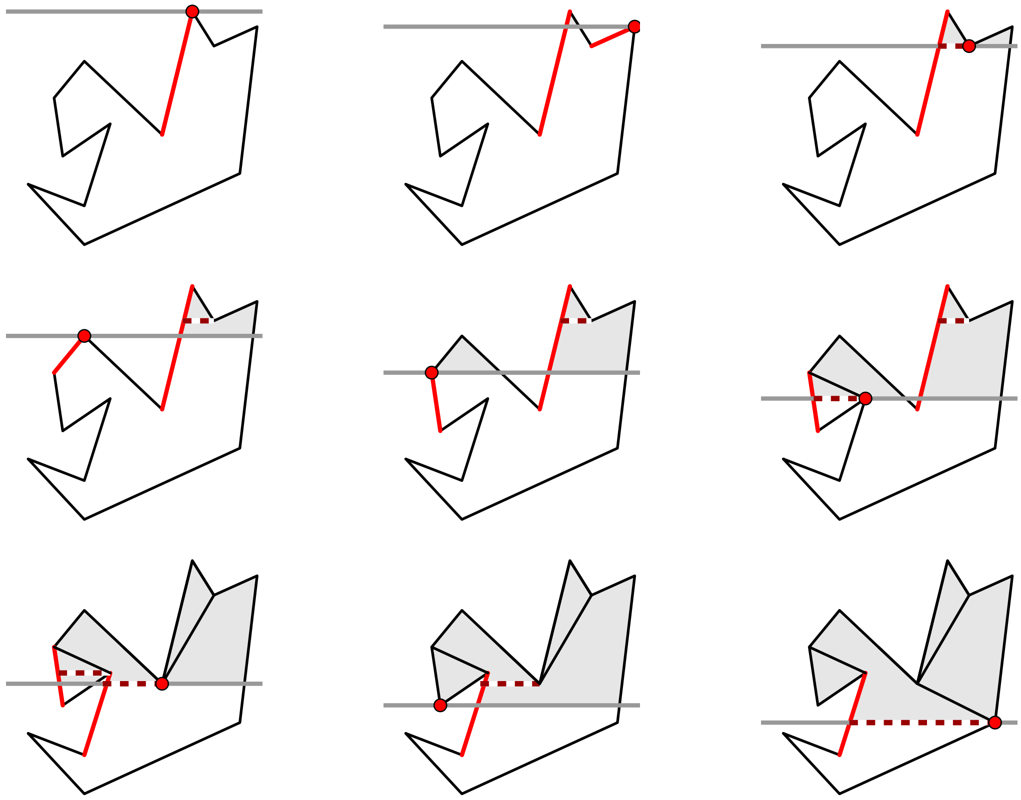
I'll assume that we are working with a simple planar region, one whose boundary is a simple closed polygon. I'll also assume for purely technical convenience that all edges are non-degenerate, which means that no two successive vertices are the same. Orient the boundary according to the right hand rule, so that the interior of the region lies on its left bank (using here the geographical terminology applied to rivers).

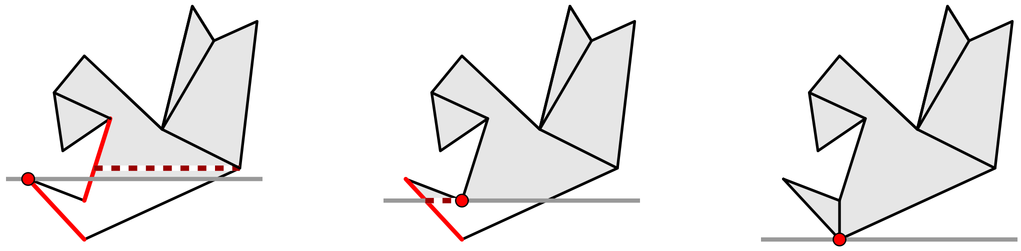
A region is called **monotone** (or more precisely **vertically monotone**) if its boundary descends on its left side and ascends on its right. In the exceptional case that its boundary possesses horizontal fragments, a horizontal segment moving left to right is considered to be **descending**, one moving right to left to be **ascending**. A region fails to be monotone if traversing the boundary on its left or right side involves some reversals in vertical direction. Roughly speaking, a vertically monotone region is one with the property that any horizontal slice, maybe rotated counter-clockwise slightly, always meets its boundary in two or fewer points. The figure below, for example, is clearly not monotone since some horizontal slices intersect it in more than two points.



As we shall see later, a monotone region is relatively simple to triangulate. The principal step in the whole process, and the one that takes the most time in a technical sense, turns out to be chopping it into monotone pieces. How to do this will turn out to be a rather intricate business.

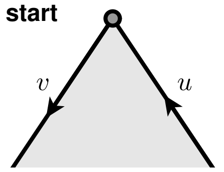
The basic idea in the monotone decomposition is to perform a top-to-bottom sweep of the region by a horizontal line. As the **sweep line** is moved down, something happens whenever it meets a vertex of the boundary, adding a diagonal occasionally from the vertex encountered to some other vertex encountered before, and updating the structures necessary to doing this. The updating involves keeping track of the descending polygon segments intersecting the sweep line, as well as of the vertices last encountered just to the right of them. Adding diagonals means splitting regions into halves. Here is how the process goes for the region above:



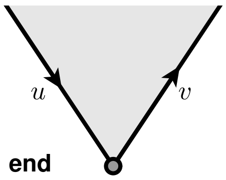


The figures exhibit some extra data that I'll explain later.

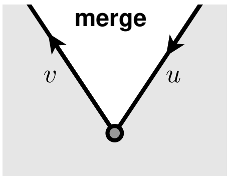
As the sweep proceeds, it jumps from one vertex to the next lower one. What it does at that point depends on the type of vertex it encounters. There are six possibilities, according to the relative position of the entering and leaving vectors u and v. Call a vector $[x, y]$ **ascending** if $y > 0$ or $y = 0$ and $x < 0$, and call it **descending** otherwise. Let $u^\perp$ be u rotated counter-clockwise by $90°$.



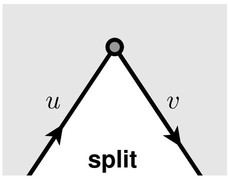
A **start** vertex. Here u is ascending, v descending, and $v \bullet u^\perp > 0$.



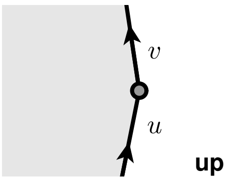
An **end** vertex. Here u is descending, v ascending, and $v \bullet u^\perp > 0$.



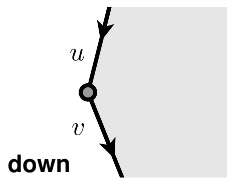
A **merge** vertex. Here u is descending, v ascending, and $v \bullet u^\perp < 0$.



A **split** vertex. Here u is ascending, v descending, and $v \bullet u^\perp < 0$.

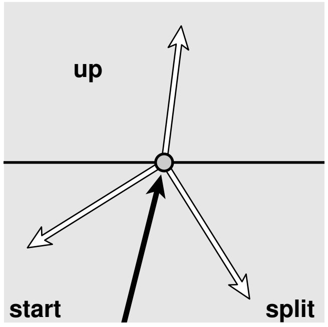


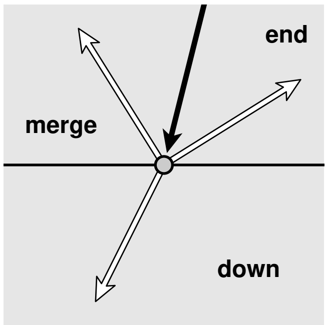
An **up** vertex. Here u and v are both ascending.



A **down** vertex. Here u and v are both descending.

The classification might be clearer from these pictures:

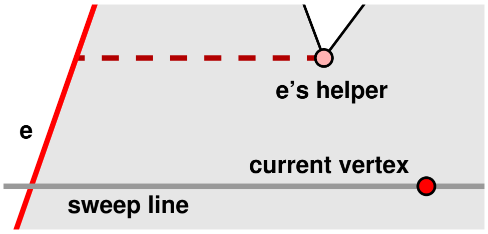




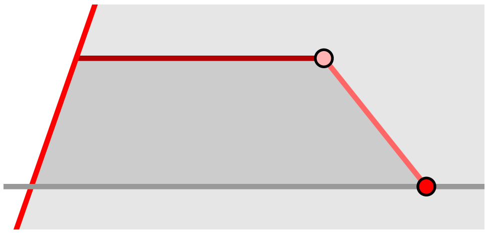
A region is monotone precisely when it has no merge or split vertices, so the point of the process is to draw diagonal lines in order to eliminate them, by separating the polygon into smaller pieces. The difficulty is to know when and how to to do this. There are two basic principles involved in carrying out the sweep: (1) Whenever we encounter a split vertex we must draw a diagonal line upwards from it; (2) whenever we encounter a merge vertex we must mark it and be sure to draw a diagonal line upwards to it eventually. Thus every diagonal drawn will have either a merge or a split vertex at one end, possibly both. Knowing when to add one will require keeping track of a number of different things as we go, the extra data indicated in the figures above.

First of all, we must keep a list $\mathcal{L}$ of all the descending boundary segments encountered by the current sweep line. The important thing is to be able to determine for any vertex what boundary segment lies immediately to its left, and this can change quite drastically as we move down. The list $\mathcal{L}$ will be have to be ordered and searchable as well as dynamically maintained. We'll see later how to deal with the technical issues that arise.

Second, diagonals will always be drawn back from a vertex we have just met to one we encountered earlier. Geometrically, it is easy to describe what the earlier one is. If e is any descending segment of the polygon, its **helper** at any given moment is the vertex u that can be connected to e across the interior of our region, lowest among those above the sweep line with this property.



When a helper is a merge vertex it is a candidate for the top of a diagonal to be drawn from below; when it is a split vertex a diagonal must be drawn upwards from it. Each descending boundary segment in the sweep intersection list is associated to its helper, and this assignment may change in the course of the sweep.



The crucial property of a helper is that the region bounded by the sweep line, the descending segment e (just to the left of the current vertex), the line to the helper, and the diagonal from the current vertex to the helper is always a region free of other vertices and edges.

What about the ordered list of intersections? It will be a **search list** with dynamic data. At any moment it is to store all the descending boundary segments intersecting the sweep line, ordered left to right according to the intersection. There are three operations we must be able to perform on it: • remove a given edge; • insert an edge in the correct place; • find the edge in the list immediately to the left of a given vertex. It is the last operation that is the crucial one, but maintaining this property correctly requires the previous two.

In addition to this dynamic list, we must make up a list—a static queue—of all vertices of the polygon right at the beginning of the sweep, ordered according to height, top to bottom. This requires an initial sort, and then keeping track of vertices already dealt with. In principle, it is this sort that sets one lower bound to the complexity of the whole process, since if there are n vertices it will generally require $O(n \log n)$ time.

15.2. The algorithm

Start with the array p of 2D points representing the vertices p_i of our simple closed polygon. We'll assume there to be at least 3. This array will be the argument to the triangulation algorithm. The output will be an array of polygons of the same type making up the monotone decomposition.

We build the vertex list, each (enhanced) vertex now an array of

- a point p_i
- a predecessor edge
- a successor edge
- a type
- an index

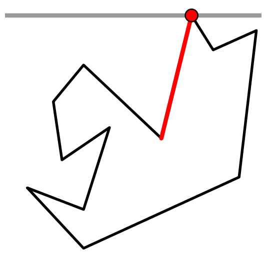
The index is the index of the vertex in the array of vertices making up the original polygon. It is needed for technical reasons when making up the polygons that go into the output. Deciding the type of a vertex will require looking at predecessor and successor edges. At the same time we build the edges, where an edge is an array of

- starting vertex
- end vertex
- the function $Ax + By + C$, which vanishes on the edge and is positive on the inner side
- the edge's helper
- its node it corresponds to in the list $\mathcal{L}$, if any

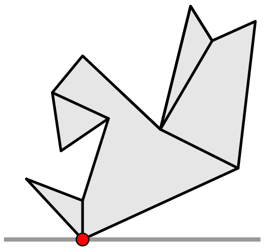
The function $Ax + By + C$ is used to compute the intersection of the sweep line with the edge. When an edge is first created from a pair of vertices its helper is `null`, as is its node pointer.

We sort the vertex list, top to bottom, left to right in case of matches. Put this sorted list into a queue. Initialize the dynamic search tree $\mathcal{L}$, with (as I have already said) methods for insertion of an edge according to its position, removal of an edge by designation of its node in the tree, and location of an edge just to the left of a given vertex.

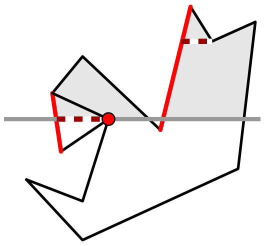
While this queue isn't empty, keep popping a vertex v off the queue. According to what type this vertex is:



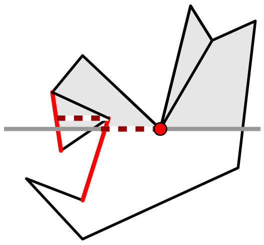
Start. Put the successor edge of v in $\mathcal{L}$. Set its helper equal to v.



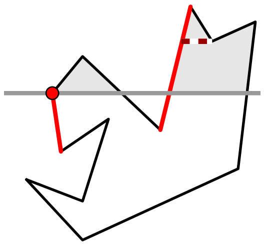
End. Let e be the predecessor edge. If its helper u is a merge, make a diagonal from v to u. Remove e from $\mathcal{L}$.



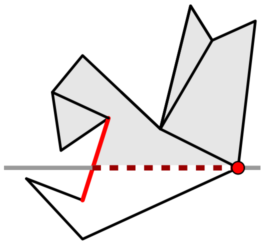
Split. Let e be the edge immediately left of v. Make a diagonal from v to its helper, and set the new helper of e to be v. Add the successor edge of v to $\mathcal{L}$.



Merge. Let e be the predecessor of v. If u is its helper and it is a merge, then make a diagonal from v to u. Remove e from $\mathcal{L}$. Let f be the edge in $\mathcal{L}$ directly left of v. If its helper w is a merge, make a diagonal from it to v. Set the new helper of f equal to v.

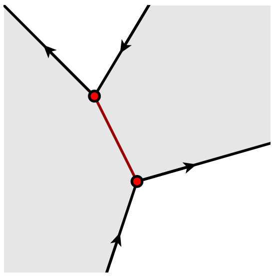


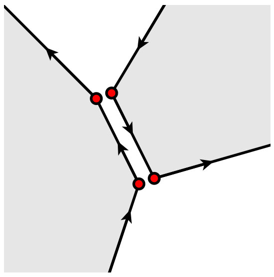
Down. The region lies to the right of v. Let e be the predecessor of v. If its helper u is a merge, make a diagonal from it to v. Remove e from $\mathcal{L}$. Insert the successor edge in $\mathcal{L}$ and set its helper equal to v.



Up. The region is to its left. Let e be directly left in $\mathcal{L}$. If its helper is a merge, make a diagonal from it to v. Set the helper of e equal to v.

Making a diagonal means splitting a polygonal region into two pieces. This is done as these figures indicate:





The partition therefore adds two new vertices to the collection and reassigns predecessor and successor edges. Some care has to be taken to be sure when calling this procedure that assignment of helpers is compatible with this split.

It is an interesting exercise to check that this all works fine—diagonals never intersect edges or other diagonals, and every split or merge vertex is fixed up sooner or later.

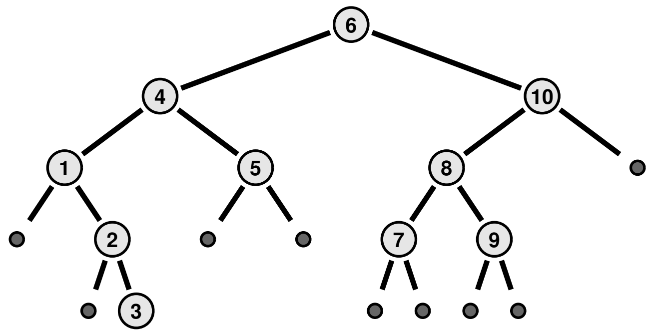
15.3. The intersection list

As part of the process of finding a monotone decomposition, we need a list $\mathcal{L}$ of descending edges intersected by the current sweep line, ordered by the x-value of the intersection. This list changes as the sweep line moves down. There are three procedures required of it—we must be able to

- insert an edge into the list;
- remove an edge;
- find the edge just to the left of any given value of x

How can we implement such a list in PostScript? The simplest thing to do is maintain an array of all the current edges, ordered left to right. But then all three operations would be quite inefficient—if n is the length of the current list, then locating an edge would on the average require $n/2$ comparisons, and inserting an edge would mean on the average shifting $n/2$ edges one place forward. Maintaining a **linked list** with nodes holding both the data to be stored as well as a link to the next node would improve insertion and removal somewhat, but location would still be poor. The solution I use is a **binary search tree**. This is a tree whose nodes contain data as well as links to two descendants and a parent node. A good reference for this, or at least for a search tree closely related to the one we'll use, is Chapter 14 of the book by Sedgewick listed at the end of this chapter.

Often, the nodes of a binary search tree store data associated to a numerical key. Each node of the tree has left and right descendants, left and right, which may be `null`. All of a node's left descendants have a key value less than its own, and all of its right descendants have a key value greater than or equal to its own.



The point of a binary search tree is that we don't have to search through the whole list to deal with its entries, but instead just have to go down through the branches of the tree. The first is an $O(n)$ process, the second on the average $O(\log n)$. In our case, each node of the tree will store a descending edge, and the nodes are sorted in the tree according to the x-value of an edge's intersection with the current sweep line. Explicitly, if the equation of the edge is $Ax + By + C = 0$ with $[A, B]$ pointing inwards, and y is the height of the sweep line, then

$$x = -\frac{By + C}{A}$$

if $A \neq 0$, otherwise (the edge is horizontal) the x-coordinate of the top vertex of the edge. Insertion of data in such a tree is simple, as is location of a predecessor, but removal is tricky. Refer to Sedgewick's book for details of how it works.

There is something a little unusual about this search tree—the sweep line changes continually, and the key value of a node changes with it. This turns out not to matter, because although the key value of a node changes, the ordering of the current nodes does not change except with respect to a new insertion or removal. In the implementation we use, the search tree is assigned a procedure `key` that assigns to each node its x-value based on the current value of y.

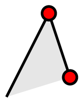
15.4. Triangulation

Once we have decomposed the original region into monotone subregions, the next step is to triangulate them. This is reasonably straightforward. Again I follow the book by de Berg et al.

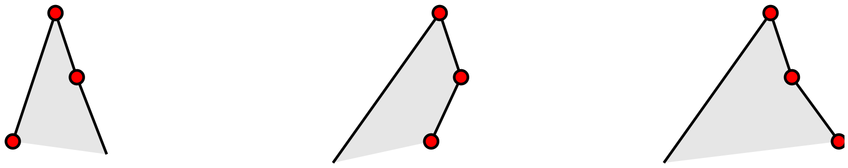
The basic principles of the algorithm can be formulated succinctly:

- *Go from top to bottom.*
- *Make a triangle whenever possible.*
- *Cut away dead wood.*

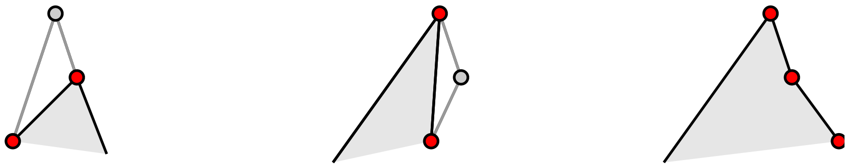
Let's see how these work out in practice at the beginning of a triangulation. We start with a single point, the top of the polygon. Since the region is monotone, the next point has to lie along an edge from the first. So the starting point always looks essentially like this (up to left-right interchange):



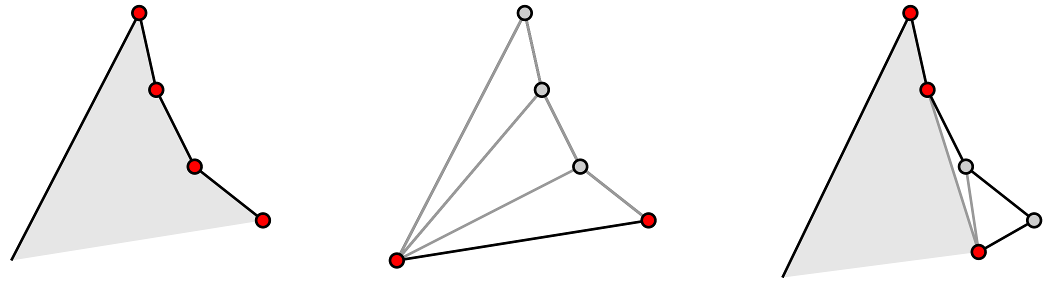
When we place a third point, however, there are three rather distinct possibilities:



Following the principles laid out above, this is the result:



The interesting thing is that in the first two cases we are essentially back at the starting point. In each of those, a triangle has been cut off, one point has become dead, and we are again looking at two active points, which form vertices of a shrunken region. Whereas in the third case we are in a new situation with 3 points still active, their fate as yet undetermined. It is easy to see by an inductive argument that this is essentially what always happens in moving down one level—either we find ourselves in a simpler situation, or we find ourselves facing a fan configuration like that here on the left:



This is basically the same problem we saw above. There will be one of two basic possibilities—the next point to be scanned lies either on the left, and we connect it to all the active points on the right (except the top, which is already connected), or on the right, and we connect it to all the active points that are visible to it (leaving active the top point connected as well as all above it). In any event, we update the list of active points and find ourselves facing a similar situation for the next step.

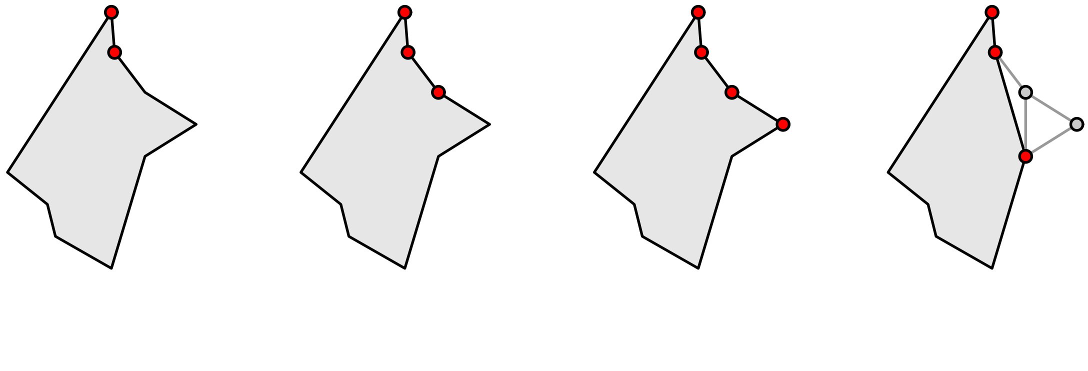
Here is a more detailed account of how things go. We start off by ordering the vertices, top to bottom, classifying them according to whether they lie on the left or right side of our polygon. This is possible because of monotonicity. Also because of monotonicity, this ordering can be done by starting at the top point and merging left and right sides by going backwards and forwards from the top. Then we set up a **stack** of active points. For the uninitiated, I recall that a stack is one of the most basic of all structures in computation, machine or human—items on a stack represent work postponed. Items on a stack are placed there and removed according to **LIFO** (**L**ast **I**n, **F**irst **O**ut) protocol. Putting something onto the top of the stack is called **pushing it**, and removing it from the top is called **popping** it. When you put a hammer down to go find a nail, you are pushing the use of the hammer onto your personal stack, and when you come back with the nail you are popping it. The procedures I use to implement a stack in PostScript are discussed at the end of this chapter.

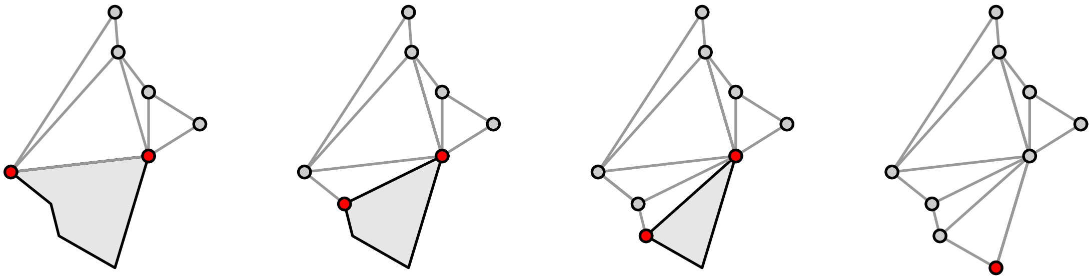
We now put the two top vertices on the stack, and start examining the remaining vertices, top to bottom. A vertex stored on the stack hold its 2D coordinates as well as the designation of what side it lies on. Suppose we are looking at the vertex v_i. Let u be at the top of the stack. What we do next depends on the exact situation:

(1) The vertex u is on the same side of the polygon as v_i. Pop u. Pop other vertices from the stack, as long as they are visible withing the region from v_i. Make triangles from these and their predecessor. Push the last one popped and v_i onto the stack.

(2) It is on the opposite side. Pop all vertices from the stack. Make up diagonals for all but the last one. Push u and v_i onto the stack.

This works because there are always at least two vertices on the stack. Here is how the whole process goes for one region:



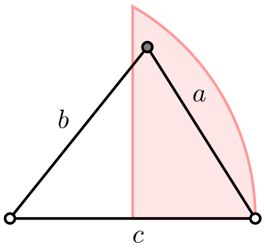


15.5. Small triangles

The triangles produced so far will vary quite a lot in shape and size. The last step is to produce from them a collection of uniform small size. How to do so elegantly depends on the shape of the triangle.

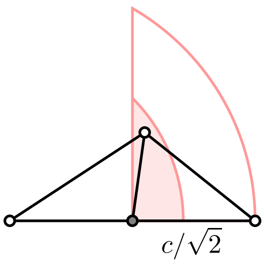
The mathematical problem can be formulated quite precisely: *How to partition a given triangle into a minimum number of triangles of no more than some specified diameter δ?* This looks like a very difficult problem. If the diameter concerned is very small and hence a huge number of small triangles would be necessary, the likely solution likely involves paving a large part of the triangle with equilateral triangles. In practice, however, the diameter will not be all that small and it doesn't seem to me worthwhile to try for the best solution. So we look instead at the problem of partitioning a triangle into a reasonable number of smaller triangles of diameter no more than δ.

We can get some idea of what we are facing by first sorting the lengths of the sides, so we may assume $a \le b \le c$. If we orient the triangle so its longest side lies along the interval $0 < x \le c$, and reflect if necessary, we may assume the third vertex to lie in the region shown below:

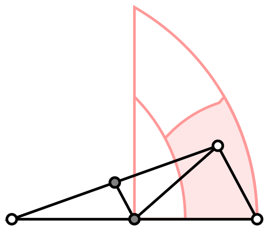


There are several extreme possibilities. Of course if $c \le \delta$ we do nothing. Otherwise the triangle is going to partitioned into successively smaller ones in a series of steps until the diameter is at most δ, and the principal problem is to decide what one step amounts to. Somewhat arbitrarily, I choose the following procedure, which guarantees that at each step I reduce the diameter of the triangles being considered by a factor of $1/\sqrt{2}$:

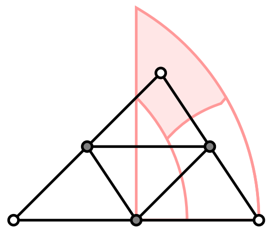
- *If $b/c \le 1/\sqrt{2}$, we make a single cut:*



- *Otherwise, if $a/c \le 1/2$, we make two cuts:*



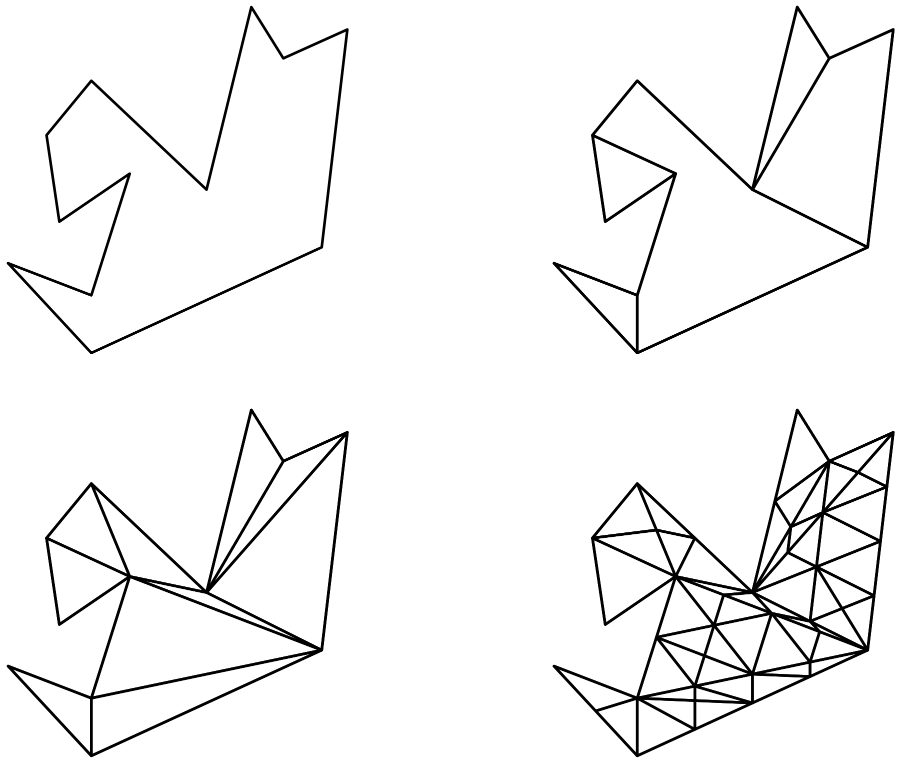
- *In all other cases, we make three:*



In the final algorithm, we use a stack to hold triangles that remain to be subdivided. While the stack isn't empty, we pop triangles off the stack and subdivide them. Any triangles that are small enough are output, while those that are not are put back on the stack. It is most efficient if a triangle on the stack holds the lengths of its sides as well as its vertices. If $n = \lceil \log(c/\sqrt{2}\delta) \rceil$ then the stack may be set at size $4n$.

15.6. Code

The triangulation code, which includes the monotone partition, triangulation, and subdivision of triangles, is in `triangulation.inc`. Here in summary is the whole sequence of steps:



The procedure `monotone` has a single argument, an array of 2D points making up a closed polygon, and returns an array of polygons making up a monotone decomposition. The procedure `monotone-triangulation` has a single argument, a monotone polygon, and returns an array of triangles making up a triangular partition of it. The procedure `subdivide` has two arguments, a triangle and a real number δ, and returns an array of triangles partitioning it into triangles of diameter no more than δ. The procedure `triangulation` is the composite of these—it has two arguments, a polygon and δ and returns an array of triangles of diameter at most δ.

The code for stacks is straightforward, implementing a stack as an array of length specified in advance, and can be found in `stack.inc`. This is a very simple package, and can be used surprisingly often, if only as an expandable array. The procedures in this file are

Arguments	Command	Left on stack; side effects
n	`new-stack`	a stack of potential size n
x stack	`stack-push`	pushes x onto the stack
stack	`stack-peek`	returns the current stack top, leaving it in place
stack	`stack-pop`	removes and turns the current stack top
stack	`stack-length`	returns current size
stack	`stack-empty`	empty or not?
stack	`stack-full`	more room?
stack	`stack-extend`	doubles the size
stack	`stack-array`	returns an array of exactly the right size
n	`stack-element-at`	returns n-th item from bottom on the stack

No checks are made on arrays when pushing or popping, so underflow and overflow can occur, producing one of those dreaded PostScript error messages. But you can check for these possibilities with `stack-empty` and `stack-full`. Of course stacks should be very familiar to you by now, since PostScript uses them all the time.

The code for the binary search tree I use in triangulation is in `dynamic-tree.inc`. Code for a simpler binary search tree, the one discussed explicitly in Sedgewick's book, can be found in `simple-tree.inc`. Here the keys are integers, indexing data to be stored in and retrieved from the tree. Sedgewick's book includes explicit code that turns out not be too hard to render in PostScript. After all, algorithms are based on ideas, more or less independently of language details. One major difficulty, and it indeed leads often to pain, is that in PostScript the only structures are arrays, and the only way to access items in an array is by index. Thus whereas in C you can write `x->key = 3` where `x` is a structure with a field named `key`, in PostScript you have to let `x` be an array, and you allocate one of its items, say that with index 2, to a field you can think of as its key. Thus `x->key = 3` in C becomes `x 2 3 put` in PostScript. Alas. Things like this can be done easily in principle, but in practice it's hard on the human memory to program like this. One small thing you can do, and I recommend it, is to make up a short dictionary of fields, here for example one defining `KEY` to be 2, so you can write `x KEY 3 put`. If you do this, you want to `begin` this dictionary only when you want to use it and `end` it when you don't need it any more, or lots of different local conventions like this will get confounded.

Another thing to keep in mind when translating C to PostScript is that an array in either language, or a structure in C, is actually a **pointer**, that is to say a single machine address where the items in the array or structure are stored. Changing data in the structure does not change this address itself. Copying it does not give a new copy of the data in the structure, but just a new copy of the address.

References

1. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, **Computational Geometry—algorithms and applications**, Springer-Verlag, 1997. The history of polygon triangulation is discussed at the end of Chapter 3. Planar sweeps are a commonly used technique in advanced graphics, and other examples of its use can also be found here. Particularly interesting mathematically is the treatment of Voronoi cell construction in Chapter 7. The problem I look at in this chapter is typical of a huge family of similar graphics problems requiring relatively subtle techniques for their solution. This book has a good selection of further references.

2. Robert Sedgewick, **Algorithms in C**, Addison-Wesley, 1990. This is a deservedly popular text covering many basic computer algorithms. There is explicit code in C, but it can be read fruitfully without knowing much about that programming language. The illustrations accompanying the explanations of algorithms are invaluable. The most recent edition is **Algorithms in Java**, and in addition to a change of programming language there is also a much expanded exposition.

3. There is much information available on the Internet about computational graphics. Much of it can be used profitably in practical illustration problems. A good place to begin is Godfried Toussaint's web site at

`http://www-cgrl.cs.mcgill.ca/~godfried/teaching/cg-web.html`