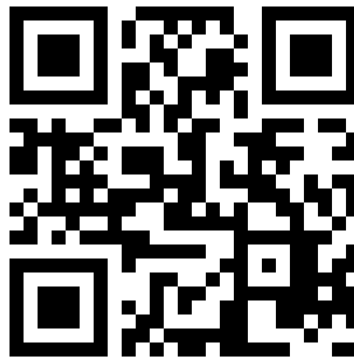
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

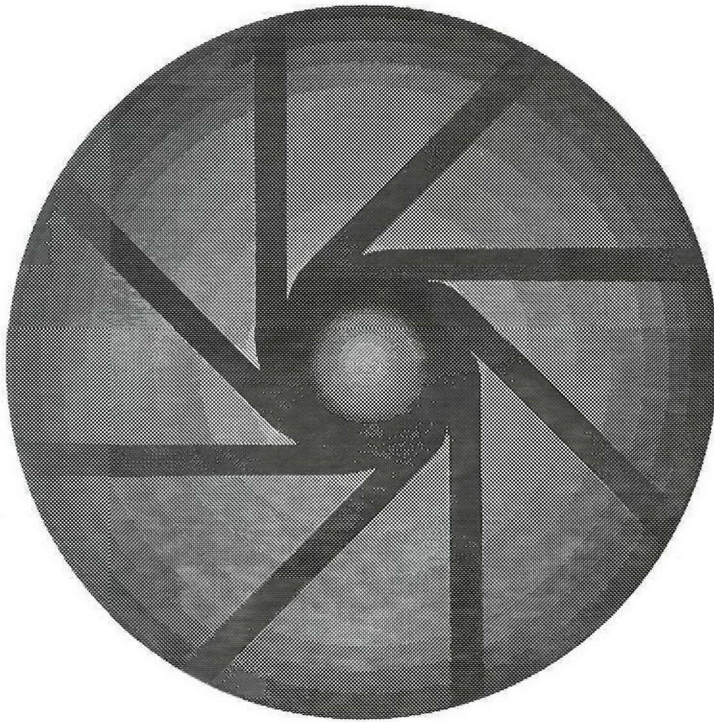
Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

COMPUTATIONAL GEOMETRY AND COMPUTER GRAPHICS IN C++



MICHAEL J. LASZLO

https://hemanthrajhemu.github.io

Part II **APPLICATIONS, 105**

5 INCREMENTAL INSERTION, 106

5.1 Insertion Sort, 107
5.1.1 Analysis, 108

5.2 Finding Star-Shaped Polygons, 109
5.2.1 What are Star-Shaped Polygons? 109
5.2.2 Finding Star-Shaped Polygonizations, 110

5.3 Finding Convex Hulls: Insertion Hull, 112
5.3.1 What are Convex Hulls? 112
5.3.2 Insertion Hull, 113
5.3.3 Analysis, 115

5.4 Point Enclosure: The Ray-Shooting Method, 116

5.5 Point Enclosure: The Signed Angle Method, 120

5.6 Line Clipping: The Cyrus-Beck Algorithm, 122

5.7 Polygon Clipping: The Sutherland-Hodgman Algorithm, 125

5.8 Triangulating Monotone Polygons, 128
5.8.1 What are Monotone Polygons? 129
5.8.2 The Triangulation Algorithm, 129
5.8.3 Correctness, 134
5.8.4 Analysis, 135

5.9 Chapter Notes, 135

5.10 Exercises, 136

6 INCREMENTAL SELECTION, 137

6.1 Selection Sort, 137
6.1.1 Off-Line and On-Line Programs, 138

6.2 Finding Convex Hulls: Gift-Wrapping, 139
6.2.1 Analysis, 141

6.3 Finding Convex Hulls: Graham Scan, 141

6.4 Removing Hidden Surfaces: The Depth-Sort Algorithm, 145
6.4.1 Preliminaries, 145
6.4.2 The Depth-Sort Algorithm, 146
6.4.3 Comparing Two Triangles, 150
6.4.4 Refining a List of Triangles, 152

https://hemanthrajhemu.github.io

6.5 Intersection of Convex Polygons, 154
6.5.1 Analysis and Correctness, 160
6.5.2 Robustness, 161

6.6 Finding Delaunay Triangulations, 162
6.6.1 Finding the Mate of an Edge, 168

6.7 Chapter Notes, 170

6.8 Exercises, 172

7 **PLANE-SWEEP ALGORITHMS, 173**

7.1 Finding the Intersections of Line Segments, 174
7.1.1 Representing Event-Points, 174
7.1.2 The Top-Level Program , 176
7.1.3 The Sweepline Structure, 177
7.1.4 Transitions, 178
7.1.5 Analysis, 181

7.2 Finding Convex Hulls: Insertion Hull Revisited, 182
7.2.1 Analysis, 183

7.3 Contour of the Union of Rectangles, 183
7.3.1 Representing Rectangles, 184
7.3.2 The Top-Level Program, 186
7.3.3 Transitions, 188
7.3.4 Analysis, 191

7.4 Decomposing Polygons into Monotone Pieces, 191
7.4.1 The Top-Level Program, 192
7.4.2 The Sweepline Structure, 195
7.4.3 Transitions, 198
7.4.4 Analysis, 201

7.5 Chapter Notes, 201

7.6 Exercises, 201

8 **DIVIDE-AND CONQUER ALGORITHMS, 203**

8.1 Merge Sort, 204

8.2 Computing the Intersection of Half-Planes, 206
8.2.1 Analysis, 208

8.3 Finding the Kernel of a Polygon, 208
8.3.1 Analysis, 209

https://hemanthrajhemu.github.io

(convex) polygons p_i, we have $A_i = 0$ for each $i = 1, 2, \ldots, k$. It follows that $A = A_0 + A_1 + \cdots + A_k = -360 + 0 + \cdots + 0 = -360$.

Function `pointInPolygon2` solves the point enclosure problem for point `a` and polygon `p`. The signed angle at `a` relative to the polygon is accumulated in `total` as each edge is visited in turn. If `a` is found to lie on some edge (the signed angle relative to the edge equals 180), the function immediately returns the enumeration value `BOUNDARY`. Otherwise, when all polygon edges have been processed, it returns `INSIDE` or `OUTSIDE` depending on the final value of `total`.

```
int pointInPolygon2(Point &a, Polygon &p)
{
   double total = 0.0;
   for (int i = 0; i < p.size(); i++, p.advance(CLOCKWISE)) {
      Edge e = p.edge();
      double x = signedAngle(a, e);
      if (x == 180.0)
         return BOUNDARY;
      total += x;
   }
   return ((total < -180.0) ? INSIDE: OUTSIDE);
}
```

Program `pointInPolygon2` runs in time linear in the size of the polygon in the worst case.

5.6 Line Clipping: The Cyrus-Beck Algorithm

The process of discarding that portion of a geometric object that lies outside a given region is called *clipping*. Clipping is used for many purposes in computer graphics. In a windowing system, a window may serve as a small aperture into a panorama that extends far beyond. When drawing into the panorama, it is necessary to clip away those portions of objects that do not fall under the purview of the window. In some text editors, it is necessary to clip characters that do not fit on a line. In three-dimensional graphics, objects in space are clipped to a volume before being projected into the image plane, to avoid wasting time projecting things that will not be seen anyway.

In this section we present the Cyrus-Beck algorithm for clipping a line segment to a convex polygon. In the subsequent section we will cover the Sutherland-Hodgman algorithm for clipping an arbitrary polygon to a convex polygon.

Let s be a line segment and let p be a convex n-gon to which s is to be clipped. Here p is called a *clip polygon* and s the *subject*. We seek $s \cap p$, that portion of s which lies in p.

Let $\vec{s}$ denote one of the two directed infinite lines determined by s. Suppose we extend each of the n edges of polygon p to infinity, in both directions. Line $\vec{s}$ crosses all these extended edges of p in no more than n distinct intersection points. (If $\vec{s}$ is parallel to or collinear with some edge of p, the edge does not contribute an intersection point; moreover, if $\vec{s}$ passes through a vertex of polygon p, two intersection points coincide.)

https://hemanthrajhemu.github.io

The Cyrus-Beck clipping algorithm finds these intersection points and classifies each as either *potentially entering* (PE) or *potentially leaving* (PL). Suppose $\vec{s}$ crosses an extended edge e at intersection point i. Then point i is PE if $\vec{s}$ passes from the left of e to the right of e. Given our convention that the polygon interior lies to the right of each of its edges, $\vec{s}$ "potentially enters" the polygon at intersection point i. That is, if $s \cap p$ is nonempty, $s \cap p$ must lie *beyond* point i (see Figure 5.14). An intersection point i is PL if $\vec{s}$ passes from the right of e to the left of e. In this case, line $\vec{s}$ "potentially leaves" polygon p—if $s \cap p$ is nonempty, $s \cap p$ must lie *behind* point i.

Intersection points are easily classified as PE or PL. Let vector n be perpendicular to some edge e of the clip polygon, pointing to the right of e. Let vector $v = b - a$, where $\vec{s} = \vec{ab}$. Then intersection point i (at which $\vec{s}$ crosses e) is PE if the angle between n and v measures less than 90 degrees, and PL if this angle measures greater than 90 degrees. (If the angle measures 90 degrees, the intersection point does not exist since $\vec{s}$ must then be parallel to or collinear with edge e.) In terms of the dot product, point i is PE if $n \cdot v > 0$ and PL if $n \cdot v < 0$.

How is classification of the intersection points used? Suppose we order the intersection points along $\vec{s}$. Then line $\vec{s}$ intersects clip polygon p only if the intersection points comprise a sequence of PE intersection points followed by a sequence of PL intersection points. Moreover, the clipped line segment $s \cap p$ we seek extends from the last PE intersection point to the first PL intersection point (segments s_1 and s_2 of Figure 5.14).

On the other hand, if the PE and PL intersection points are interspersed along $\vec{s}$, clipped line segment $s \cap p$ is empty (s_3 of Figure 5.14). For in this case there must exist some point a along $\vec{s}$ such that some PL point lies behind a and some PE point lies beyond a. But since $s \cap p$ lies behind the PL point, $s \cap p$ must lie behind a; and since $s \cap p$ lies beyond the PE point, $s \cap p$ must lie beyond a. This is possible only if $s \cap p$ is empty.

Rather than work with intersection points directly, the Cyrus-Beck algorithm works with the parametric values (along $\vec{s}$) of these intersection points. The algorithm maintains

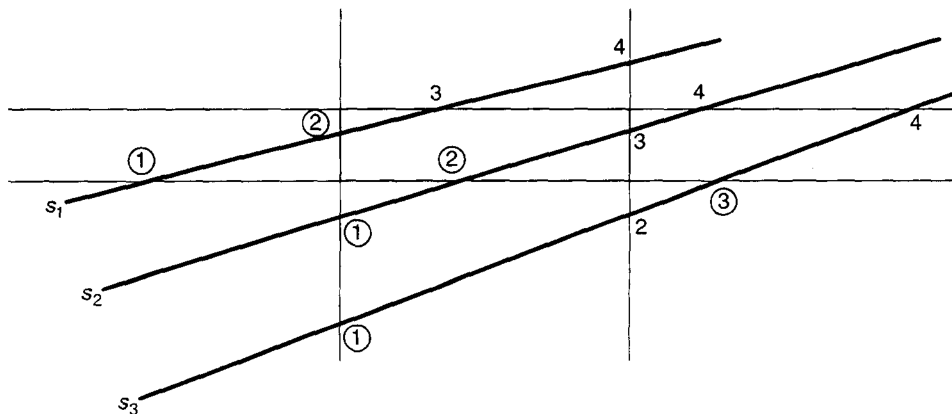


Figure 5.14: Clipping line segments s_1, s_2, and s_3 to a square clip polygon. Intersection points are labeled according to their order along each s_i, and the labels of potentially entering (PE) intersection points are encircled; the remaining intersection points are potentially leaving (PL).

https://hemanthrajhemu.github.io

a range $[t_0, t_1]$ of parametric values corresponding to the *current line segment*, which converges to the clipped segment $s \cap p$ we seek as the algorithm proceeds. Initially, the current line segment is line segment s, represented by the range [0, 1]. As the algorithm processes each edge e of the clip polygon, the current range either remains unchanged or shrinks (its lower limit increases or its upper limit decreases). Specifically, whenever a PE intersection point with parametric value t is discovered, the lower limit t_0 of the current range is updated: $t_0 = \max(t_0, t)$. Similarly, finding a PL intersection point with parametric value t requires that we update the upper limit of the current range: $t_1 = \min(t_1, t)$. When all edges have been processed (and consequently all intersection points found), the current range $[t_0, t_1]$ represents the clipped segment $s \cap p$ we seek. If $t_0 \leq t_1$, this clipped segment is nonempty; otherwise ($t_0 > t_1$) it is empty.

The following function clips subject line segment `s` to clip polygon `p` and returns `TRUE` if the result is nonempty and `FALSE` otherwise. If nonempty, the clipped line segment is passed back through reference parameter `result`:

```
bool clipLineSegment(Edge &s, Polygon &p, Edge &result)
{
   double t0 = 0.0;
   double t1 = 1.0;
   double t;
   Point v = s.dest - s.org;
   for (int i = 0; i < p.size(); i++, p.advance(CLOCKWISE)) {
      Edge e = p.edge();
      if (s.intersect(e, t)==SKEW) { // s and e cross at a point
         Edge f = e;
         f.rot();
         Point n = f.dest - f.org;
         if (dotProduct(n, v) > 0.0) {
            if (t > t0)
               t0 = t;
         } else {
            if (t < t1)
               t1 = t;
         }
      } else {   // s and e are parallel or collinear
         if (s.org.classify(e) == LEFT)
            return FALSE;
      }
   }
   if (t0 <= t1) {
      result = Edge(s.point(t0), s.point(t1));
      return TRUE;
   }
   return FALSE;
}
```

Observe how function `clipLineSegment` handles the case where s is parallel to some edge e of the clip polygon. If s lies to the left of e, the function immediately returns

https://hemanthrajhemu.github.io

FALSE and exits. Otherwise the function ignores e and goes on to the next edge. The algorithm clearly runs in time proportional to the size of the clip polygon.

5.7 Polygon Clipping: The Sutherland-Hodgman Algorithm

Polygon clipping, the process of clipping a subject polygon to a clip polygon, is more interesting than line clipping, for what results from the process is not just a collection of line segments, but a collection of *polygons*. Moreover, the problem of polygon clipping challenges us to exploit the structure inherent in the subject polygon, to treat it as more than a mere collection of line segments. In this section we cover the Sutherland-Hodgman polygon clipping algorithm. Given a *convex* clip polygon p and an arbitrary subject polygon s, the algorithm constructs the region $s \cap p$, a collection of zero or more polygons.

The Sutherland-Hodgman algorithm clips the subject polygon to each edge of the convex clip polygon in turn. The subject polygon is first clipped to one edge of the clip polygon, then the polygon that results is clipped to the next edge, and so on: The polygon that results from each clip operation is "piped" into the next clip operation. We are done when the subject polygon has been clipped to every edge of the clip polygon. The algorithm is illustrated in Figure 5.15, where the clip polygon is a square.

The algorithm is implemented by function `clipPolygon`, which is passed a subject polygon `s` and a convex clip polygon `p`. The result is passed back through reference parameter `result`. The function returns `TRUE` only if the result is nonempty:

```
bool clipPolygon(Polygon &s, Polygon &p, Polygon* &result)
{
   Polygon *q = new Polygon(s);
   Polygon *r;
   int flag = TRUE;
```

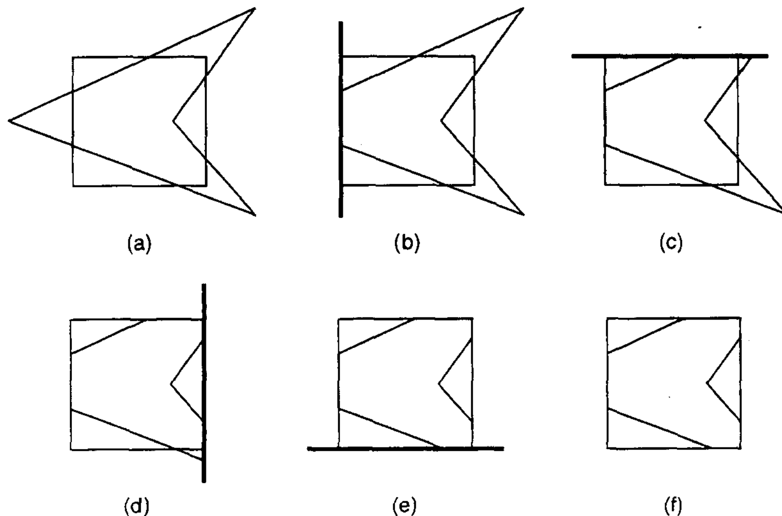


Figure 5.15: The Sutherland-Hodgman clipping algorithm.

https://hemanthrajhemu.github.io

```
    for (int i = 0; i < p.size(); i++, p.advance(CLOCKWISE)) {
        Edge e = p.edge();
        if (clipPolygonToEdge(*q, e, r)) {
            delete q;
            q = r;
        } else {
            delete q;
            flag = FALSE;
            break;
        }
    }
    if (flag) {
        result = q;
        return TRUE;
    }
    return FALSE;
}
```

In each iteration of `clipPolygon`, variable `q` points to the current subject polygon and variable `r` to the polygon that results from clipping `q` to edge `e` of the clip polygon. Initially, `q` is made to point to a copy of subject polygon `s` (it points to a *copy* of `s` so `s` is not destroyed). Clipping `q` to edge `e` is accomplished by the call to function `clipPolygonToEdge`, which returns `TRUE` if the polygon `r` which results is nonempty. If `r` turns out to be nonempty, it is piped into the next clip operation by the assignment instruction `q=r`; otherwise function `clipPolygon` exits, returning `FALSE`.

Function `clipPolygonToEdge` clips subject polygon s to the right side of an edge e of the clip polygon. An output polygon is grown incrementally into the clipped polygon we seek. The idea is to compare each edge of s to edge e in turn. Depending on the result of each comparison, zero, one, or two vertices are inserted into the output polygon under construction.

The four possible relationships between e and an edge of s are shown in Figure 5.16. Where $\overrightarrow{ab}$ is the current edge of s, the contribution to the output polygon resulting in each case is as follows:

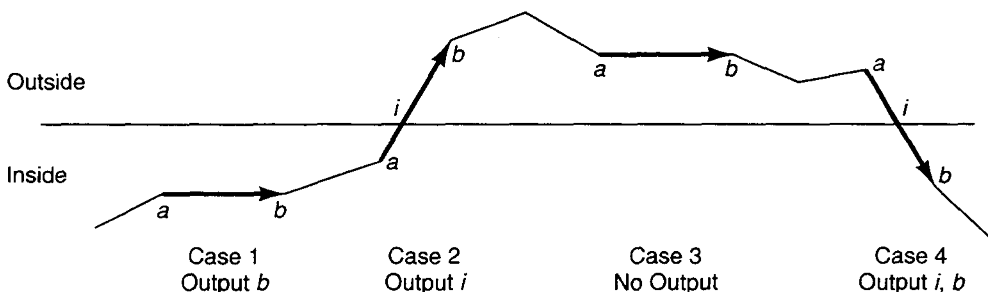


Figure 5.16: Possible relationships between an edge and the clip half-plane: (Case 1) output b; (Case 2) output i; (Case 3) no output; (Case 4) output i, then b.

https://hemanthrajhemu.github.io

1. *Edge* $\overrightarrow{ab}$ *lies to the right of* e. Output vertex b.
2. *Edge* $\overrightarrow{ab}$ *crosses from the right of* e *to the left of* e. Output the point i where $\overrightarrow{ab}$ crosses e.
3. *Edge* $\overrightarrow{ab}$ *lies to the left of* e. No output.
4. *Edge* $\overrightarrow{ab}$ *crosses from the left of* e *to the right of* e. Output i, then b, where $\overrightarrow{ab}$ crosses edge e at point i.

Function `clipPolygonToEdge` clips subject polygon `s` to edge `e`. It returns the resulting polygon through reference parameter `result`, and returns `TRUE` only if polygon `result` is nonempty:

```
bool clipPolygonToEdge(Polygon &s, Edge &e, Polygon* &result)
{
   Polygon *p = new Polygon;
   Point crossingPt;
   for (int i = 0; i < s.size(); s.advance(CLOCKWISE), i++)
      Point org = s.point();
      Point dest = s.cw()->point();
      int orgIsInside = (org.classify(e) != LEFT);
      int destIsInside = (dest.classify(e) != LEFT);
      if (orgIsInside != destIsInside) {
         double t;
         e.intersect(s.edge(), t);
         crossingPt = e.point(t);
      }
      if (orgIsInside && destIsInside)          // case 1
         p->insert(dest);
      else if (orgIsInside && !destIsInside) {   // case 2
         if (org != crossingPt)
            p->insert(crossingPt);
         }
      else if (!orgIsInside && !destIsInside)    // case 3
         ;
      else {                                    // case 4
         p->insert(crossingPt);
         if (dest != crossingPt)
         p->insert(dest);
      }
   }
   result = p;
   return (p->size() > 0);
}
```

What happens if `clipPolygonToEdge` is handed a problem whose solution consists of *multiple* polygons? In this case, `clipPolygonToEdge` produces a single polygon that contains degenerate boundary edges. The situation is depicted in Figure 5.17.

https://hemanthrajhemu.github.io

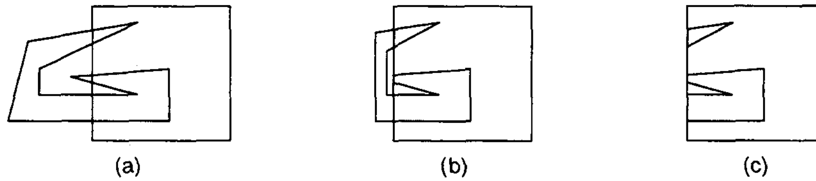


Figure 5.17: (a) A clipping problem; (b) the resulting polygon with three degenerate edges (displaced horizontally in the figure); and (c) the collection of two polygons it represents.

To partition the polygon into non-degenerate pieces, we first sort the endpoints of the degenerate edges along the common line with which they are all collinear. We then apply `Vertex::splice` repeatedly to excise the degenerate edges. Since this refinement will not be needed for the applications in Chapter 8 that makes use of function `clipPolygonToEdge`, we will not pursue it further.

Let us analyze program `clipPolygon` in terms of the size $|s|$ of subject polygon `s` and the size $|p|$ of clip polygon `p`. Function `clipPolygonToEdge` runs in $O(|s|)$ time and is called at most once per edge of the clip polygon, or at most $|p|$ times. Hence program `clipPolygon` runs in $O(|s|\ |p|)$ time in the worst case.

5.8 Triangulating Monotone Polygons

A *triangulation* of a polygon is a decomposition of the polygon into a set of triangles. Triangulations are often used to reduce problems involving complicated regions to problems involving triangles, which, because triangles are among the simplest of regions, are generally easier to solve. For instance, to determine whether a given point lies in a nonconvex polygon, we can triangulate the polygon and then answer yes only if the point belongs to at least one of the triangles. Or to render a higher-order surface embedded in space, we can approximate the surface by a mesh of triangles, which can be rendered more easily.

In this section we present a linear-time algorithm to triangulate polygons of a special type, known as *monotone polygons*. With this algorithm's appearance in 1978, researchers achieved the first method for triangulating arbitrary n-gons in $O(n \log n)$ time:

1. Decompose the polygon into monotone pieces in $O(n \log n)$ time.
2. Triangulate the monotone pieces in total $O(n)$ time.

In Chapter 7 we will present an $O(n \log n)$-time algorithm for decomposing a polygon into monotone pieces.

An important question, which has been settled only recently, is whether a general triangulation algorithm faster than $O(n \log n)$ is possible. Faster triangulation algorithms have been developed, but some solve only special cases in which the input polygon is constrained, and the improved performance of others depends on additional properties of the polygon (such as the number of reflex vertices it possesses). Yet in recent years several general triangulation algorithms which run in $o(n \log n)$ time have been developed. In 1991 Bernard Chazelle devised an optimal $O(n)$-time algorithm.

https://hemanthrajhemu.github.io

5.8.1 What Are Monotone Polygons?

A vertex chain is said to be *monotone* if every vertical line crosses it in at most one point. When a monotone chain is traversed beginning from its leftmost vertex, its vertices are visited by increasing x-coordinates.

A polygon is *monotone* if its boundary is composed of two monotone chains: the polygon's *upper* chain and *lower* chain. Each chain terminates at the polygon's leftmost vertex and rightmost vertex and contains zero or more vertices in between. Figure 5.18 gives some examples. Observe that the (nonempty) intersection of a vertical line and a monotone polygon consists of either a vertical line segment or a point.

5.8.2 The Triangulation Algorithm

Let p be a monotone polygon, and let us relabel its vertices as $v_1, v_2, \ldots, v_n$ by increasing x-coordinates since our algorithm will examine the vertices in this order. The algorithm produces a succession of monotone polygons $p = p_0, p_1, \ldots, p_n = \emptyset$. Polygon p_i, the result of examining vertex v_i, is obtained by splitting zero or more triangles from the previous polygon p_{i-1}. The algorithm is finished when we are left with p_n, the empty polygon—the collection of triangles accumulated along the way represents the triangulation of the original polygon p.

The algorithm maintains a stack s of vertices that have been examined but not yet fully processed (some as yet undiscovered triangles may meet these vertices). As vertex v_i is about to be examined, the stack contains some of the vertices of polygon p_{i-1}. Certain stack invariants are maintained as the algorithm proceeds.[1] Specifically, where the vertices on the stack are labeled $s_1, s_2, \ldots, s_t$ from the bottom of the stack to the top, the following conditions are maintained:

1. $s_1, s_2, \ldots, s_t$ are ordered by increasing x-coordinates and includes every vertex of p_{i-1} that lies both to the right of s_1 and to the left of s_t,
2. $s_1, s_2, \ldots, s_t$ are consecutive vertices in either p_{i-1}'s upper chain or its lower chain,
3. vertices $s_2, s_3, \ldots, s_{t-1}$ are reflex vertices in p_{i-1} (the measure of each of their interior angles exceeds 180 degrees), and

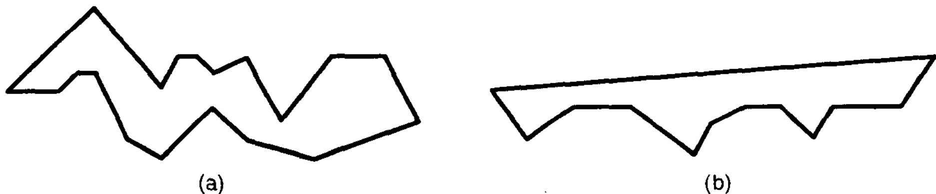


Figure 5.18: Two monotone polygons. The upper chain of polygon (b) consists of a single edge only.

[1] An *invariant* is a condition that holds true at specific points of the algorithm, such as at the start of every iteration of a given loop.

https://hemanthrajhemu.github.io

4. in polygon p_{i-1}, the next vertex v_i to be examined stands in one of these relations to vertices s_t and s_1:
   (a) v_i is adjacent to s_t but not to s_1, or
   (b) v_i is adjacent to s_1 but not to s_t, or
   (c) v_i is adjacent to both s_1 and s_t.

The three cases of condition 4 are shown in Figure 5.19.

The action taken when vertex v_i is examined depends on which one of stack conditions 4a, 4b, or 4c currently holds. The actions, illustrated in Figure 5.19, are as follows:

Case 4a *Vertex v_i is adjacent to s_t but not to s_1*: While $t > 1$ and internal angle $\angle v_i s_t s_{t-1}$ measures less than 180 degrees, split off triangle $\Delta v_i s_t s_{t-1}$, then pop s_t from the stack. Finally, push v_i. The algorithm uses the fact that $\angle v_i s_t s_{t-1} < 180$ only if either (1) s_{t-1} lies to the left of $\overrightarrow{v_i s_t}$ if v_i belongs to polygon p_{i-1}'s upper chain or (2) s_{t-1} lies to the right of $\overrightarrow{v_i s_t}$ if v_i belongs to the lower chain.

Case 4b *Vertex v_i is adjacent to s_1 but not to s_t*: Split off the polygon determined by vertices $v_i, s_1, s_2, \ldots, s_t$, then empty the stack, then push s_t followed by v_i. The polygon defined by the vertices $v_i, s_1, s_2, \ldots, s_t$ is in fact fan shaped with apex v_i (i.e., v_i belongs to its kernel). The algorithm then triangulates this polygon.

Case 4c *Vertex v_i is adjacent to both s_1 and s_t*: In this case $v_i = v_n$ and polygon p_{i-1}, determined by vertices $v_i, s_1, s_2, \ldots, s_t$, is fan shaped with apex v_n. The algorithm triangulates this polygon directly, and exits.

Figure 5.20 runs the algorithm on a small problem (the stages are ordered top to bottom, left to right). In each stage, the vertex being examined is circled, and the vertices on the stack are labeled $s_1, \ldots, s_t$.

The following program, `triangulateMonotonePolygon`, is passed a monotone polygon `p` and returns a list of triangles representing a triangulation of the polygon. The program assumes that polygon `p`'s window is positioned over its leftmost vertex:

```
enum { UPPER, LOWER };

List<Polygon*> *triangulateMonotonePolygon(Polygon &p)
```

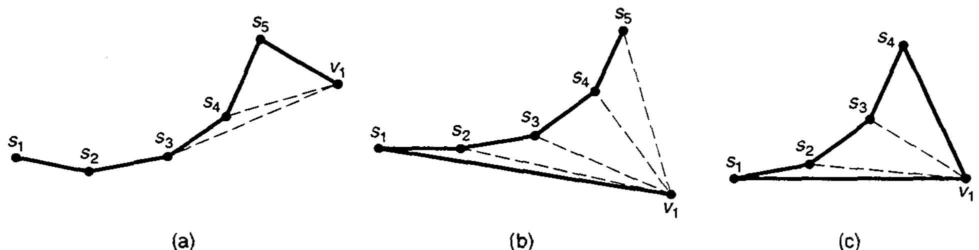


Figure 5.19: The three cases that occur while triangulating a monotone polygon: Vertex v is (a) adjacent to s_t but not to s_1, (b) adjacent to s_1 but not to s_t, or (c) adjacent to both s_1 and s_t.

https://hemanthrajhemu.github.io

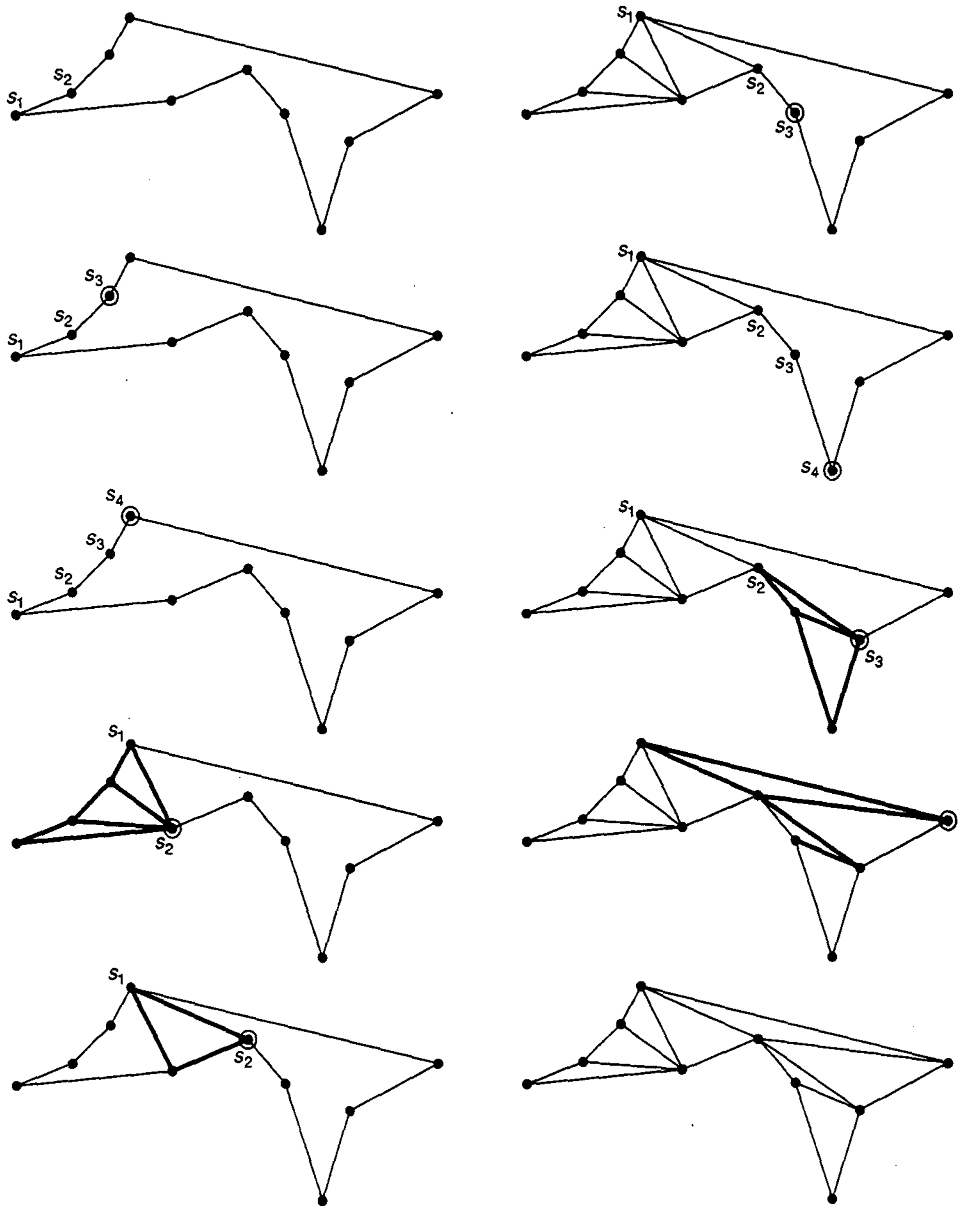


Figure 5.20: Triangulating a small monotone polygon. The triangles discovered at each stage are highlighted. The stages proceed from top to bottom, left to right.

```
{
   Stack<Vertex*> s;
   List<Polygon*> *triangles = new List<Polygon*>;
   Vertex *v, *vu, *vl;
   leastVertex(p, leftToRightCmp);
```

https://hemanthrajhemu.github.io

```
    v = vu = vl = p.v();
    s.push(v);
    int chain = advancePtr(vl, vu, v);
    s.push(v);
    while (1) {   /* outer while loop */
       chain = advancePtr(vl, vu, v);
       if (adjacent(v, s.top()) &&
            !adjacent(v, s.bottom())) { // case 4a
          int side = (chain == UPPER) ? LEFT : RIGHT;
          Vertex *a = s.top();
          Vertex *b = s.nextToTop();
          while ((s.size() > 1) &&
                  (b->classify(v->point(),a->point()) == side)) {
             if (chain == UPPER) {
                p.setV(b);
                triangles->append(p.split(v));
             } else {
                p.setV(v);
                triangles->append(p.split(b));
             }
             s.pop();
             a = b;
             b = s.nextToTop();
          }
          s.push(v);
       } else if (!adjacent(v, s.top())) {   // case 4b
          Polygon *q;
          Vertex *t = s.pop();
          if (chain == UPPER) {
             p.setV(t);
             q = p.split(v);
          } else {
             p.setV(v);
             q = p.split(t);
             q->advance(CLOCKWISE);
          }
          triangulateFanPolygon(*q, triangles);
          while (!s.empty())
             s.pop();
          s.push(t);
          s.push(v);
       } else {   // case 4c
          p.setV(v);
          triangulateFanPolygon(p, triangles);
          return triangles;
       }
    }  /* end of outer while */
}
```

https://hemanthrajhemu.github.io

Function adjacent returns TRUE just if the two vertices it is passed are adjacent:

```
bool adjacent(Vertex *v, Vertex *w)
{
   return ((w == v->cw()) || (w == v->ccw()));
}
```

Program triangulateMonotonePolygon merges the upper and lower chains of polygon p as it proceeds, thereby taking advantage of the fact that the vertices in each chain are already ordered by increasing x-coordinates [otherwise an $O(n \log n)$ time sort would be necessary]. In each iteration, variable v points to the vertex to be examined. The program maintains two variables, vu and vl, which point to the last vertex examined in p's upper and lower chains, respectively. As the program proceeds, these pointers are marched left to right by function advancePtr. Each time advancePtr is called, it advances either vu or vl and updates v to point to whichever is advanced:

```
int advancePtr(Vertex* &vu, Vertex* &vl, Vertex* &v)
{
   Vertex *vun = vu->cw();
   Vertex *vln = vl->ccw();
   if (vun->point() < vln->point()) {
      v = vu = vun;
      return UPPER;
   } else {
      v = vl = vln;
      return LOWER;
   }
}
```

Function advancePtr returns UPPER or LOWER, indicating which of the two chains v belongs to. Program triangulateMonotonePolygon uses this value to ensure that its subsequent call to split returns the piece detached from the main polygon, rather than the main polygon from which the piece was detached.

To triangulate a fan shaped polygon, we iteratively find the triangles which fan out from some common apex v. To do this, we traverse the polygon starting from v, and at each vertex w that is not adjacent to v, we split along the chord connecting v to w. This is performed by function triangulateFanPolygon, which destructively decomposes the n-gon p into $n - 2$ triangles and appends these to list triangles. The function assumes that polygon p is fan shaped, and its window is positioned over some apex:

```
void triangulateFanPolygon(Polygon &p, List<Polygon*> *triangles)
{
   Vertex *w = p.v()->cw()->cw();
   int size = p.size();
   for (int i = 3; i < size; i++) {
```

https://hemanthrajhemu.github.io

```
        triangles->append(p.split(w));
        w = w->cw();
    }
    triangles->append(&p);
}
```

Figure 5.21 depicts a triangulation produced by the algorithm. The monotone polygon contains 35 vertices.

5.8.3 Correctness

We must show two things: (1) that every diagonal the algorithm finds in iteration i is (a chord) internal to polygon p_{i-1}; (2) that the algorithm restores the four stack conditions from one iteration to the next. (That the chords which are found decompose the original polygon into *triangles* is apparent from Figure 5.19.)

First consider diagonal $\overline{v_i s_{t-1}}$ of Figure 5.19a (here $t = 5$). Letting triangle $T = \triangle s_{t-1} s_t v_i$, observe that no vertex of polygon p_{i-1} can lie in T: The vertices $s_0, \ldots, s_{t-2}$ lie to the left of T's leftmost vertex s_{t-1}, and vertices v_j for $j > i$ lie to the right of T's rightmost vertex v_i. Hence any edge that crosses diagonal $\overline{v_i s_{t-1}}$ must leave triangle T by one of the edges $\overrightarrow{s_{t-1} s_t}$ or $\overrightarrow{s_t v_i}$, which is impossible since these are boundary edges of p_{i-1}. Thus diagonal $\overrightarrow{v_i s_{t-1}}$ is a chord. Now split triangle T from p_{i-1}. The same argument shows that the remaining diagonals introduced in Case 4a are also chords.

Next consider Case 4b, depicted in Figure 5.19b. By stack conditions 2 and 3, polygon T, determined by vertices $v_i, s_1, \ldots, s_t$, is fan shaped with apex v_i. Observe that no vertex of p_{i-1} can lie in the interior of T—were one or more vertices to do so, the rightmost such vertex would currently be on the stack, and hence on the boundary of p_{i-1}. Since the interior of T is free of vertices, any edge which crosses $\overrightarrow{v_i s_t}$ must also cross one of the edges $\overrightarrow{v_i s_1}, \overrightarrow{s_1 s_2}, \ldots, \overrightarrow{s_{t-1} s_t}$, which is impossible since these are boundary edges of p_{i-1}. It is shown similarly that, in Case 4c (Figure 5.19c), p_{i-1} is fan shaped with apex v_i and interior free of vertices.

Next we argue that the stack conditions are maintained from one iteration to the next. At most v_i and s_t are pushed on the stack, and when they are both pushed, they are pushed in the correct horizontal order. Thus stack condition 1 is maintained. The vertices comprise a vertex chain in p_{i-1} by induction (in Case 4a) or by the fact that the stack is reset to two adjacent vertices (in Case 4b), so condition 2 is satisfied. The stack's vertices are reflex (except for top and bottom vertices) because v_i is pushed only when the vertex on the top

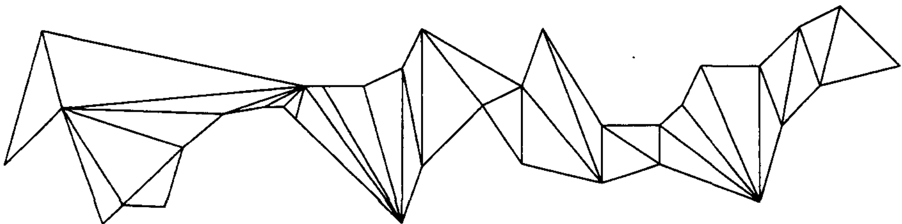


Figure 5.21: Triangulation of a monotone 35-gon.

https://hemanthrajhemu.github.io

of the stack (onto which it gets pushed) would become a reflex angle. Hence condition 3 is met (it is vacuously satisfied in Case 4b since the stack contains only two items.) Finally, v_i must be adjacent to at least one of s_1 or s_t because the monotonicity of p_{i-1} guarantees that v_i has a neighbor to its left, and all the stack's vertices except s_1 and s_t already have both neighbors accounted for. Hence stack condition 4 is satisfied.

5.8.4 Analysis

Program `triangulateMonotonePolygon` runs in $\Theta(n)$ time, where the input monotone polygon contains n vertices. To see the upper bound $O(n)$, observe that each iteration of the two inner `while` loops (in Cases 4a and 4b) pops a vertex from the stack. Yet every vertex is pushed onto the stack at most once (when it is first examined) and hence can be popped from the stack at most once. Since the algorithm (1) performs $O(n)$ constant-time stack operations and (2) spends constant time between successive stack operations, it runs in $O(n)$ time. The lower bound follows from the fact that each of the n vertices must be examined.

5.9 Chapter Notes

The need to sort arises in many settings and is ancient, predating computers and possibly even written language. Not surprisingly, many of the algorithms we cover in this book rely on sorting. During the course of this book, we present three sorting methods that belong to every computer scientist's repertoire—insertion sort, selection sort, and merge sort. These methods are so fundamental that they are all but unattributable. Yet despite their seeming simplicity, research continues to this day into the complexity of these and other basic sorting algorithms.

The number of polygonizations of a set of n points is exponential in n; however, the number of *star-shaped* polygonizations is bounded above by $O(n^4)$. If no three points in the point set are collinear, the kernels of the different star-shaped polygonizations partition the point set's convex hull. An algorithm for constructing this partition in $O(n^4)$ time is presented in [22], which leads to an $O(n^5)$-time algorithm for finding all star-shaped polygons. An algorithm for finding the kernel of any n-sided polygon in $O(n)$ time is presented in [53]; we present a simpler but less efficient $O(n \log n)$ algorithm in Chapter 8. The superb book *Art Gallery Theorems and Algorithms* by Joseph O'Rourke explores problems related to visibility inside a polygon [60].

The notion of convex hull makes sense for sets of points in d-dimensional space for any $d \geq 1$: The convex hull is the intersection of all convex polytopes that contain the points. Some methods for finding the convex hull of planar point sets are framed in terms of d-dimensional space. Insertion hull, for example, is a special case of the *beneath-beyond method* for finding the convex hull of points in d-dimensional space [45].

The line clipping method of section 5.6 is known as the Cyrus-Beck clipping algorithm, and the polygon clipping method of section 5.7 as the Sutherland-Hodgman algorithm [75]. Other well-known clipping methods include the Cohen-Sutherland and the midpoint subdivision [77] line clipping algorithms, and the Weiler-Atherton [88] polygon clipping algorithm. This last algorithm is general, allowing both subject and clip polygons to be non-

https://hemanthrajhemu.github.io

convex and to possess holes. This generality permits its use in an algorithm for performing hidden surface removal, also presented in [88].

The algorithm for triangulating monotone polygons is given in [31], which gives a more general definition of *monotone* than we have assumed: A vertex chain c is *monotone* relative to some line ℓ if every line perpendicular to ℓ intersects c in at most one point. A polygon is *monotone* if its boundary is composed of two vertex chains that are monotone relative to the same line. Our monotone polygons are actually polygons that are monotone relative to the horizontal axis. The article [67] presents a linear-time algorithm for deciding whether there exists a line relative to which a polygon is monotone.

Chazelle's [19] triangulation algorithm runs in optimal $O(n)$ time. A survey of polygon partitioning techniques is presented in [60, 61].

5.10 Exercises

1. Program `starPolygon` finds only fan-shaped polygons. Generalize the program so, when passed any point s lying in the convex hull of the set of points, it finds a star-shaped polygon containing s in its kernel.
2. Devise an $O(n \log n)$-time algorithm for finding a star-shaped polygon in a point set.
3. Prove that the kernel of a polygon is convex.
4. Write a version of insertion hull that runs in $O(n \log n)$ time. (Hint: Presort the points so the calls to `pointInConvexPolygon` and `closestVertex` are unnecessary.)
5. Show that $\Omega(n \log n)$ time is necessary to solve the convex hull finding problem. [Hint: Devise an efficient reduction from sorting to convex hull finding. Use the fact that $\Omega(n \log n)$ is a lower bound for sorting.]
6. Show that point p belongs to convex hull $\mathcal{CH}(S)$ if and only if there exist three points of S such that the triangle they determine contains p.
7. This question involves the Cyrus-Beck line clipping algorithm. Show that a line $\overline{s}$ intersects a convex polygon p only if the intersection points, when ordered along $\overline{s}$, consist of a subsequence of PE intersection points followed by a subsequence of PL intersection points.
8. Modify program `clipLineSegment` so it clips an infinite line, rather than a line segment, to a convex polygon.
9. Devise an algorithm to remove the degenerate edges which are sometimes produced by `clipPolygonToEdge`.
10. Given a polygon p and some point a inside p, devise a method for finding some ray that originates at a and crosses the minimum number of p's edges. (Hint: Consider sorting the vertices of p radially around a.)
11. Show that any triangulation of an n-gon uses $n-3$ chords to decompose the polygon into $n-2$ triangles.

https://hemanthrajhemu.github.io

6

Incremental Selection

Incremental selection methods solve problems incrementally, a little at a time. These methods, however, process the input in an order of their own making, rather than in the order in which the input is presented. Incremental selection involves scanning the input to "select" the best item to process next.

In some applications of incremental selection, the order in which items are to be processed can be determined in advance. In such cases the input can be *presorted*. In other applications the order cannot be anticipated, and each decision concerning which item to process next depends on what has been achieved so far.

In this chapter we will look at both kinds of applications. We will start with selection sort, an exemplar for incremental selection. We will then consider two more methods for constructing the convex hull of a finite point set: the gift-wrapping method and the Graham scan. The third geometric algorithm we will cover is the depth-sort method for performing hidden surface removal given a collection of triangles in space. Our next algorithm computes the intersection of two convex polygons in the plane. Our last algorithm constructs a special triangulation of a finite point set in the plane, known as the Delaunay triangulation.

6.1 Selection Sort

Selection sort, another sorting algorithm, repeatedly extracts the smallest item from a set until the set is empty. For sorting an array of items, it works as follows: Find the smallest item and exchange it with the item in the array's first position. Then find the smallest of

https://hemanthrajhemu.github.io

the remaining items (to the right of the first position) and exchange it with the item in the second position. Continue in this manner until the array is sorted.

The function template `selectionSort` sorts the items in array `a[0..n-1]`. For each `i` from 0 through $n-1$, iteration `i` selects the smallest item from among `a[i..n-1]` and then exchanges this item with `a[i]`:

```
template<class T>
void selectionSort(T a[], int n, int (*cmp)(T,T))
{
   for (int i = 0; i < n-1; i++) {
      int min = i;
      for (int j = i+1; j < n; j++)
         if ((*cmp)(a[j], a[min]) < 0)
            min = j;
      swap(a[i], a[min]);
   }
}
```

The body of the inner `for` loop performs the selection for each increment of `i`. Variable `min` holds the index of the smallest item examined in the current scan. We maintain the *index* of the item, rather than the item itself, so the subsequent exchange can be performed.

Function `swap` exchanges its two arguments:

```
template <class T> void swap(T &a, T &b)
{
   T t = a;
   a = b;
   b = t;
}
```

The running time of selection sort is $T(n) = \sum_{i=1}^{n} I(i)$, where $I(i)$ time is needed to select the ith smallest item. Since selecting this item takes $n - i$ comparisons, the program performs $T(n) = (n-1) + (n-2) + \cdots + 1 = n(n-1)/2$, or about $n^2/2$, comparisons in total. Although this running time is comparable to that of insertion sort, selection sort is generally preferable: Selection sort performs only n exchanges, whereas insertion sort performs about $n^2/2$ half-exchanges in the worst case (where shifting an item one position to the right counts as a half-exchange).

6.1.1 Off-Line and On-Line Programs

Selection sort is an example of an *off-line* method. This means that all its input data must be available from the start. It is easy to see why: If the smallest item were to arrive only after some other (larger) item were already selected, the opportunity to put the smallest item in the first position of the array would be lost. All incremental selection methods are in fact off-line since selection of the correct item at each stage cannot be guaranteed if all items are not accessible.

https://hemanthrajhemu.github.io

Insertion sort, like all incremental insertion methods, is an example of an *on-line* method. An on-line program does not look ahead at its input. This means that its input can arrive as a stream over time and does not have to be available in its entirety from the start. Although `insertionSort` happens to have access to the entire input array, it does not scan the array prior to inserting items; rather, it peels off one item at a time without looking ahead.

On-line programs are most useful in real-time settings, where the input data are generated on the fly. Text editors and flight simulators are on-line, since input to these programs is generated in real time by a user whose decisions cannot be anticipated. On the down side, on-line programs may do work which, on the basis of input data that arrives only later (too late), turns out to have been wasted effort. An example is the convex hull program `insertionHull` of the previous chapter, which sometimes assembles large current hulls only to disassemble them later. We now turn to a method for constructing convex hulls which avoids this sort of wasted work because it is based on the incremental selection approach.

6.2 Finding Convex Hulls: Gift Wrapping

One way to construct the convex hull of a finite point set S in the plane mimics how one would go about drawing it with straightedge and pencil. First select some point $a \in S$ that clearly belongs to the convex hull boundary—the leftmost vertex suffices. Then pivot a vertical ray clockwise around a until it first hits some other point b in S; segment $\overline{ab}$ is an edge of the convex hull. To find the next edge, continue pivoting the ray clockwise, this time around b, until the ray encounters some other point c; segment $\overline{bc}$ is the next edge of the convex hull. Continue in this fashion until returning to point a. Figure 6.1 depicts the process, which is known as the *gift-wrapping* method.

The process of pivoting the ray around each point is the "selection" part of the algorithm. To select the point that follows point a on the convex hull boundary, we seek point b such that no point lies to the left of ray $\overrightarrow{ab}$. The points are examined in turn, while the algorithm keeps track of the leftmost candidate encountered so far. Only those points not yet known to lie on the convex hull boundary need be examined.

The following program `giftwrapHull` returns a polygon representing the convex hull of the `n` points in array `s`. The array `s` should have length `n + 1` since the program places a sentinel point in `s[n]`:

```
Polygon *giftwrapHull(Point s[], int n)
{
   int a, i;
   for (a = 0, i = 1; i < n; i++)
      if (s[i] < s[a])
         a = i;
   s[n] = s[a];
   Polygon *p = new Polygon;
   for (int m = 0; m < n; m++) {
      swap(s[a], s[m]);
```

https://hemanthrajhemu.github.io

```
        p->insert(s[m]);
        a = m + 1;
        for (int i = m + 2; i <= n; i++) {
            int c = s[i].classify(s[m], s[a]);
            if (c == LEFT || c == BEYOND)
                a = i;
        }
        if (a == n)
            return p;
    }
    return NULL;
}
```

Pivoting the ray around some point s[m] is simulated by the inner for loop. Point s[a] is the leftmost point the ray has encountered so far. If a new point s[i] lies to the left of the ray that originates at s[m] and passes through s[a], then the pivoting ray would hit s[i] before s[a], so a is updated. Variable a is also updated if s[i] lies *beyond* s[a]—point s[a] cannot be a vertex of the convex hull if it lies between points s[m] and s[i].

Observe that function giftwrapHull rearranges the points in array s. At the end of iteration m, subarray s[0..m] contains the known vertices of the convex hull in clockwise rotation, and s[m+1..n-1] contains the remaining points, which may or may not prove to be vertices of the convex hull. It is these latter points which must be examined in subsequent iterations.

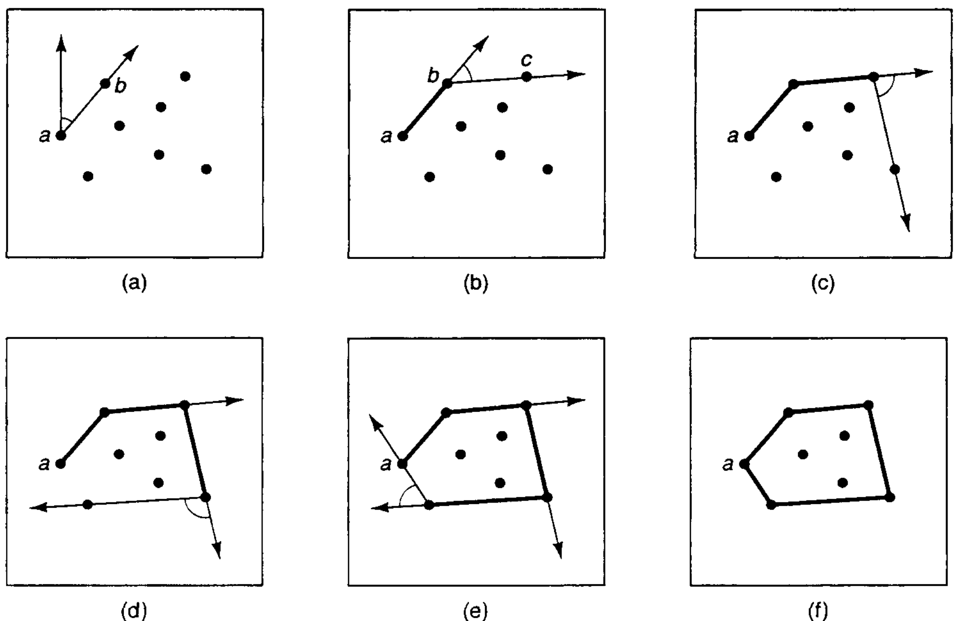


Figure 6.1: Gift wrapping a set of points in the plane.

https://hemanthrajhemu.github.io

6.2.1 Analysis

To analyze the gift-wrapping method, note that pivoting around the mth point requires $n - m - 2$ (constant-time) point-line classifications. Since only h pivots are performed [where h is the number of vertices in convex hull $\mathcal{CH}(S)$], the total running time is $O(hn)$. If every one of the n points is a vertex of $\mathcal{CH}(S)$ (i.e., if $h = n$), the running time is $O(n^2)$, comparable to that of `insertionHull`. On the other hand, whenever h is small compared to n, the gift-wrapping method is faster than the insertion hull method.

A running time like $O(hn)$ is said to be *output sensitive* since it includes a factor h that depends on the size of the output. For analyzing programs which run more quickly the less output they produce (not all programs behave like this), output-sensitive bounds provide a tighter estimate of behavior than do bounds not sensitive to output size. In the case of gift wrapping, the $O(hn)$ running time indicates that the program is efficient when the convex hull is small; this fact is not captured by the $O(n^2)$ estimate of running time, which is sensitive only to input size but not to output size.

6.3 Finding Convex Hulls: Graham Scan

In this section we cover the Graham scan, a convex-hull finding method named for its inventor, R. L. Graham. The Graham scan finds the convex hull of a finite point set S in two phases. In the *presorting phase*, the algorithm selects an extreme point $p_0 \in S$ and sorts the remaining points of S radially around p_0. In the *hull finding phase*, the algorithm iteratively processes the sorted points, thereby producing a sequence of current hulls which converges to convex hull $\mathcal{CH}(S)$. Presorting simplifies the hull finding phase: Each point processed during the hull finding phase gets inserted into the current hull, no questions asked; moreover, the vertices to be removed from the current hull are easy to find. This compares favorably with the way that the insertion hull method of the previous chapter processes each point: It must decide whether to insert the point into the current hull and, if so, traverse the current hull boundary full circle to determine which vertices are to be removed.

Given point set S, Graham scan first finds some extreme point $p_0 \in S$. We will take p_0 to be the point of S with minimum y-coordinate, or the rightmost such point in the case of a tie. The remaining points are then sorted by polar angle around p_0. If two points have the same polar angle, the point closer to p_0 is considered less than the more distant point. This is the dictionary order relation for points based on their polar coordinates relative to p_0, realized by comparison function `polarCmp` of section 5.2. Let us relabel the remaining points $p_1, p_2, \ldots, p_{n-1}$ according to this ordering, as in Figure 6.2.

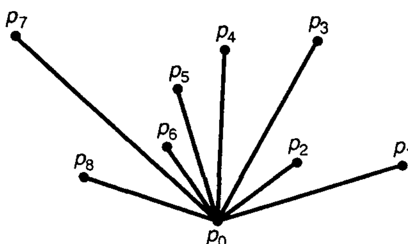


Figure 6.2: Labeling of points based on their polar coordinates relative to p_0.

https://hemanthrajhemu.github.io

During the hull finding phase, Graham scan maintains a current hull over those points that have already been inserted. Figure 6.3 illustrates the algorithm in action. Consider the insertion of point p_7 (Figure 6.3f). Because the points are ordered radially around p_0, it is

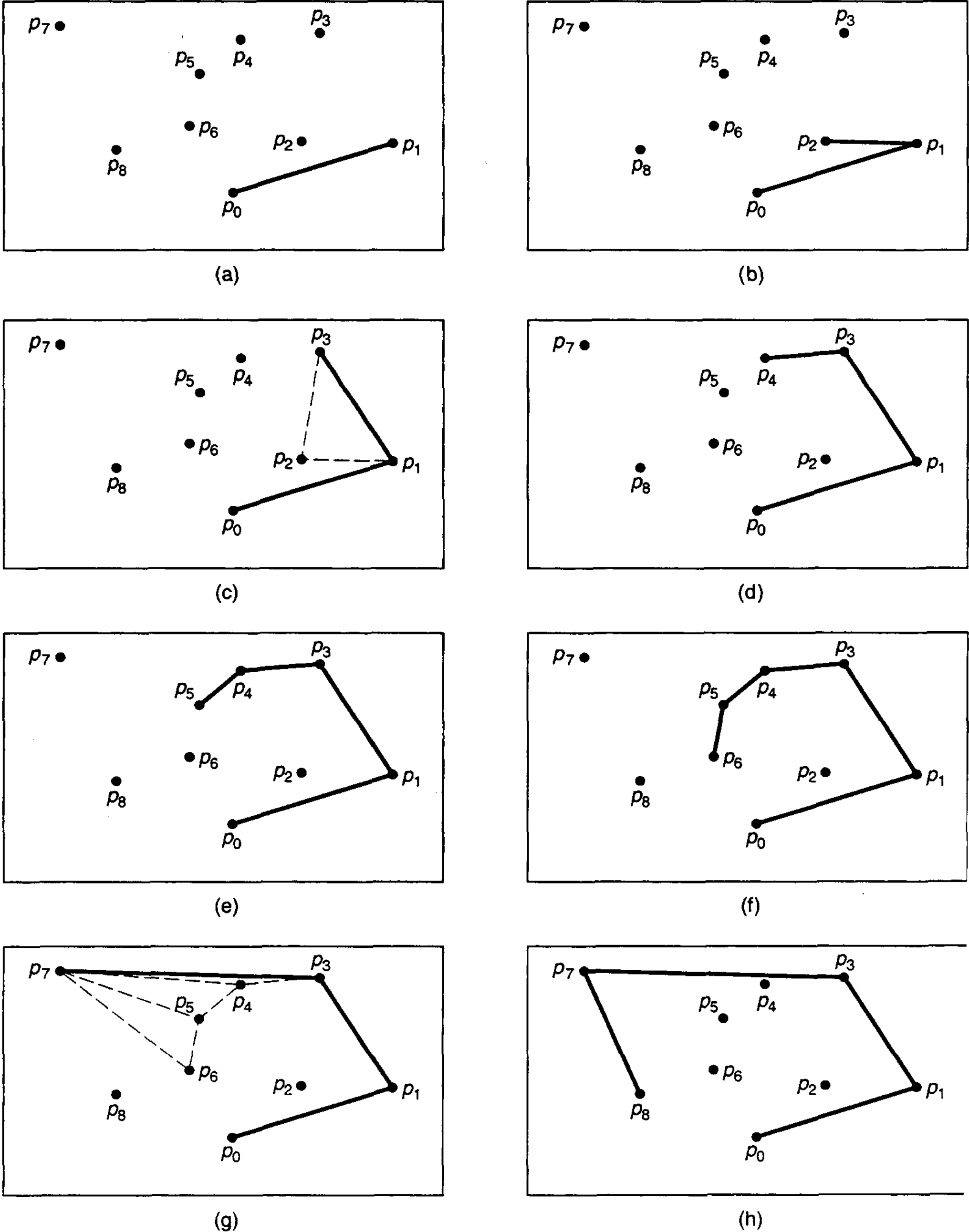


Figure 6.3: The Graham scan in action.

https://hemanthrajhemu.github.io

clear that p_7 is to be inserted and that p_0 is to become its predecessor. But which point is to become p_7's successor? To answer this, we make use of the fact that every vertex must represent a left turn in a counterclockwise traversal of the convex hull boundary. Consider point p_6, our first candidate. Since the angle $\angle p_5 p_6 p_7$ represents a non-left turn (p_7 lies to the right of edge $\overrightarrow{p_5 p_6}$), we remove p_6 from the current hull. Next we consider p_5. Since angle $\angle p_4 p_5 p_7$ also represents a non-left turn, we similarly remove p_5 from the current hull. Similarly, we remove p_4 as well since angle $\angle p_3 p_4 p_7$ is also not a left turn. When we consider point p_3, things are different: Angle $\angle p_1 p_3 p_7$ does in fact represent a left turn, so we have found p_7's successor in the updated current hull (p_3).

Program `grahamScan` is passed an array `pts` of `n` points and returns a polygon representing `pts`'s convex hull. The program works in five stages—the first two comprise the presorting phase, and the remaining three the hull finding phase:

1. Find extreme point p_0.
2. Sort the remaining sites by their polar coordinates relative to p_0.
3. Initialize the current hull.
4. Grow the current hull until it equals the convex hull of all `n` sites.
5. Convert the current hull to a `Polygon` object and return it.

The program is defined as follows:

```
Point originPt;

Polygon *grahamScan(Point pts[], int n)
{
 // stage 1
   int m = 0;
   for (int i = 1; i < n; i++)
      if ((pts[i].y < pts[m].y) ||
          ((pts[i].y == pts[m].y) && (pts[i].x < pts[m].x)))
         m = i;
   swap(pts[0], pts[m]);
   originPt = pts[0];
 // stage 2
   Point **p = new (Point*)[n];
   for (i = 0; i < n; i++)
      p[i] = &pts[i];
   selectionSort(&p[1], n-1, polarCmp); // or any sorting method
 // stage 3
   for (i = 1; p[i+1]->classify(*p[0], *p[i]) == BEYOND; i++)
      ;
   Stack<Point*> s;
   s.push(p[0]);
   s.push(p[i]);
 // stage 4
   for (i = i+1; i < n; i++) {
      while (p[i]->classify(*s.nextToTop(), *s.top()) != LEFT)
         s.pop();
```

https://hemanthrajhemu.github.io

```
      s.push(p[i]);
   }
 // stage 5
   Polygon *q = new Polygon;
   while (!s.empty())
      q->insert(*s.pop());
   delete p;
   return q;
}
```

Stage 1 is straightforward. In stage 2 we allocate array `p` and initialize its elements to point to the `Points` in array `pts`. We require an array of *pointers* so we can employ one of our generalized sorting routines. Then we sort array `p` based on comparison function `polarCmp`, which was defined in section 5.2 in the context of finding star polygons in point sets. Recall that the global variable `originPt` is used to communicate the origin point—in this case point p_0—to function `polarCmp`.

Stages 3 and 4 maintain the current hull in a stack `s`. Letting set $S_i = \{p_0, p_1, \ldots, p_i\}$, the stack represents the current hull $\mathcal{CH}(S_i)$ as follows. Where the points in the stack are labeled $s_1, s_2, \ldots, s_t$ from the bottom of the stack to the top, the stack satisfies these two *stack conditions*:

1. $p_0 = s_1, s_2, \ldots, s_t = p_i$ are the vertices of current hull $\mathcal{CH}(S_i)$ in counterclockwise rotation, and
2. edge $\overrightarrow{s_1 s_2}$ is an edge of the final convex hull $\mathcal{CH}(S)$.

Stage 3 establishes these conditions initially. The `for` loop steps along ray $\overrightarrow{p_0 p_1}$ until arriving at the last (most distant) p_i along the ray; then it pushes p_0 and p_i onto the stack. Stack condition 1 is satisfied because line segment $\overline{p_0 p_i}$ *is* the convex hull of S_i since points $p_1, \ldots, p_{i-1}$ lie between p_0 and p_i. Stack condition 2 is satisfied because $\overline{p_0 p_i}$ is an edge of $\mathcal{CH}(S)$.

In stage 4, illustrated in Figure 6.4, point p_i is processed to produce current hull $\mathcal{CH}(S_i)$. Program `grahamScan` pops $s_t, s_{t-1}, \ldots, s_{k+1}$ from the stack until reaching s_k, the topmost point of the stack such that angle $\angle s_{k-1} s_k p_i$ represents a left turn. Since these points that are popped lie in the interior of triangle $\triangle p_0 p_i s_k$ or along one of the edges $\overline{p_0 p_i}$ or $\overline{p_i s_k}$, none of them can be a vertex of $\mathcal{CH}(S_i)$. Since only these points and none others are removed from the stack, the points that remain, together with p_i, are the vertices of $\mathcal{CH}(S_i)$. Because stack condition 1 ensures that $s_1, \ldots, s_k$ are ordered correctly within the stack and p_i follows these in the polar angle ordering, the new stack contents $(s_1, \ldots, s_k, p_i)$ are correctly ordered in counterclockwise rotation. It follows that stack condition 1 is maintained.

The purpose of stack condition 2 is to guarantee that s_k exists. Since edge $\overrightarrow{s_1 s_2}$ is an edge of $\mathcal{CH}(S)$, every p_i that gets processed lies to the left of $\overrightarrow{s_1 s_2}$. Since angle $\angle s_1 s_2 p_i$ represents a left turn, it follows that s_k exists for some $k \geq 2$. Moreover, since the original s_1 and s_2 are never popped from the stack, stack condition 2 is maintained.

Stage 5 of `grahamScan` grows a `Polygon` object `q` by iteratively popping a point from the stack and inserting it into `q`. By stack condition 1, the points are popped in clockwise order.

https://hemanthrajhemu.github.io

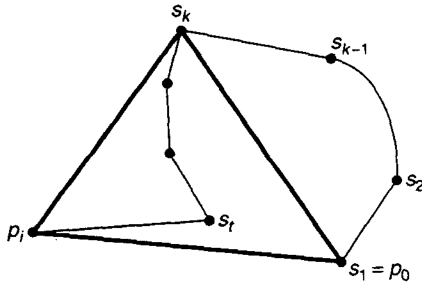


Figure 6.4: Inserting point p_i to produce current hull $\mathcal{CH}(S_i)$.

With regard to running time, it is easy to see that stages 1, 3, and 5 each take $O(n)$ time. In stage 4, the body of the `while` loop is performed at most once per point (once popped from the stack, a point never returns to the stack a second time). Hence stage 4 also takes $O(n)$ time. Therefore, total running time is dominated by the initial sort (stage 2), so Graham scan runs in $O(n \log n)$ time if an appropriate sorting method is used. It is noteworthy that Graham scan runs in linear time if the point set is known to be sorted initially.

6.4 Removing Hidden Surfaces: The Depth-Sort Algorithm

6.4.1 Preliminaries

Three-dimensional computer graphics typically involves modeling a scene in space and then forming an image of the scene in a process known as *rendering*. To render, we select a position in space from which to view the scene and, based on this viewing position and several additional viewing parameters, project the scene into a plane, where the image is formed.

What makes rendering challenging is that some of the objects in the scene, and portions of other objects, are hidden from view and so should not appear in the final image. Some of the objects may lie outside the field of view (identifying these objects is the problem of clipping). In addition, some objects (and portions of objects) may be hidden by other opaque objects that lie between them and the viewing position. The problem of identifying these hidden objects is known as the *hidden surface removal* problem.

In this section we solve the hidden surface removal problem through *depth sorting*. The scene will be represented by a collection of triangles in space. This model is in wide use, in large part because it accommodates a wide range of scenes. For instance, any surface can be approximated by a mesh of triangles which, by making the mesh sufficiently fine, can be made to resemble the surface as closely as desired. Even relatively coarse meshes are useful in practice since shading methods applied during rendering can greatly enhance the impression of the surface's curvature.

The projection we will employ maps points in space along lines parallel to the z-axis: Point (x,y,z) projects to point $(x,y,0)$. This projection, known as *orthographic parallel projection*, can be assumed without loss of generality: Given the set of viewing parameters describing some desired view, a sequence of transformations can be performed which reduces the original rendering problem to one involving orthographic parallel projection.

https://hemanthrajhemu.github.io

By convention, we will assume that the viewing position is in the $-z$-half-space (behind the xy-plane), that the scene lies in the $+z$-half-space (beyond the xy-plane), and that depth increases—objects are farther away—as z increases.

We will further assume that the triangles in the scene are oriented such that they are viewed from their negative half-spaces—their normal vectors point away from the viewing position (Figure 6.5). This assumption is less limiting than might first appear. When using a mesh of triangles to model the surface of a solid, the triangles are oriented consistently, relative to the solid's interior: for example, the normals all point toward the interior of the solid. In a prerendering step known as *backface culling*, we discard those triangles whose normals point toward the viewing position since they cannot be seen—the solid's interior lies between each such triangle and the viewing position. We are left with only those triangles whose normals point away from the viewing position. Even when a mesh of triangles is used to represent a "free-floating" surface that is not the boundary of a solid (so there exists no solid to occlude triangles), the triangles can be reoriented to ensure that their normals point away from the viewing position.

6.4.2 The Depth-Sort Algorithm

Hidden surface removal is most easily performed on a set of triangles which do not overlap in z-coordinate. First sort the triangles by decreasing z (from far to near), and then paint them in this order. If a triangle is visible, it will paint over whatever it hides, and nothing will be painted over it. This approach is sometimes called the *painter's algorithm* since it is how a painter might first paint the background, then the scene at intermediate depth, and finally the foreground. Each layer is painted on top of the previous, more distant layer.

The painter's algorithm exploits the fact that it is safe to paint something if it does not hide anything to be painted later. We will say that a list of triangles is *visibility ordered* if it is safe to paint each one in the given order—that is, no triangle hides any that follow. More formally, a list of triangles $P_1 \prec P_2 \prec \cdots \prec P_n$ is visibility ordered with respect to viewpoint p if and only if this holds: If $P_i \prec P_j$, then P_i does not obscure P_j when viewed from p. *Depth sorting* is the process of arranging a set of triangles into visibility order.

Some sets of triangles admit more than one visibility ordering. A simple example is that of two triangles, neither of which obscures the other. Other sets admit a unique visibility ordering, and others, as we shall see shortly, admit none at all.

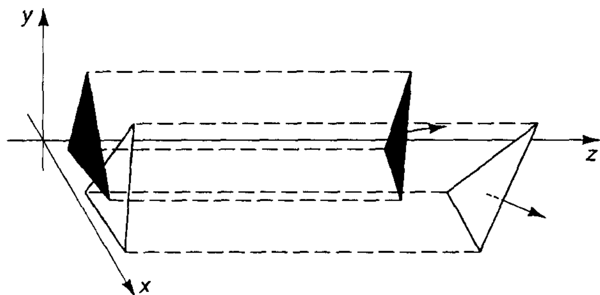


Figure 6.5: The setting for hidden surface removal.

https://hemanthrajhemu.github.io

Hidden surface removal is more difficult to perform on a set of triangles which overlap in z-coordinate. Based on the painter's algorithm, we would like to determine which of two given triangles obscures the other by comparing canonical values selected from each triangle's z-extent. (A triangle's *z-extent* is the range of z-coordinates it spans; equivalently, it is the perpendicular projection of the triangle's bounding box into the z-axis.) However, this does not work. If, for example, we use z^M, the maximum value in each triangle's z-extent, the triangles in Figure 6.6 would be ordered $A \prec B$ since $z_A^M > z_B^M$. But this is not a visibility ordering since A obscures B and so cannot be safely painted first (their actual visibility ordering is $B \prec A$). This example illustrates that depth sorting generally requires that a given list of triangles be rearranged, even if the original list is tentatively ordered from far to near. The algorithm we will present rearranges the order of a tentatively ordered list by performing a sequence of *shuffle* operations.

Some sets of triangles admit no visibility ordering at all. If two triangles in the set interpenetrate each other as in Figure 6.7a, no visibility ordering is possible—neither triangle can precede the other in any legal visibility ordering since each obscures the other. A visibility ordering may be impossible even if the triangles are assumed not to interpenetrate each other. None of the triangles of Figure 6.7b can precede the other two in any legal visibility ordering since each obscures one of the remaining two.

The way out of this impasse involves *refining* the original set of triangles: splitting certain triangles into triangular pieces so the set of triangles which results *can* be depth sorted. If triangle A of Figure 6.7a is split by the plane of triangle B into pieces A_1, A_2, and A_3 (as in Figure 6.8a), the set of triangles that results is visibility ordered by $A_1 \prec B \prec A_2 \prec A_3$. If triangle C of Figure 6.7b is split by the plane of D into C_1, C_2, and C_3 (Figure 6.8b), we have the visibility ordering $C_1 \prec D \prec E \prec C_2 \prec C_3$. The set of

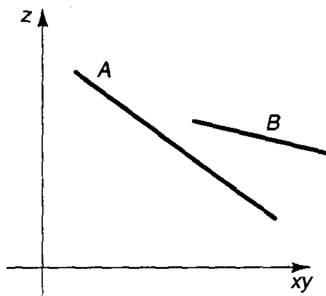


Figure 6.6: The visibility ordering of these triangles is $B \prec A$ even though $z_A^M > z_B^M$.

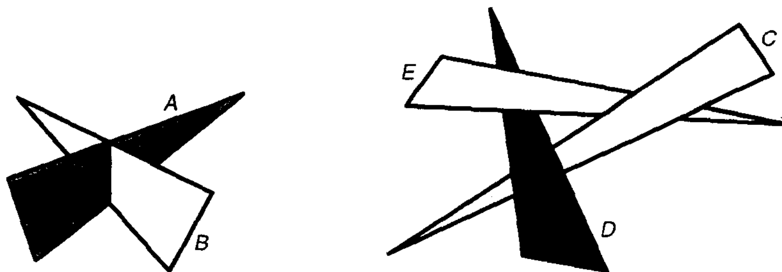


Figure 6.7: No collection of triangles containing either of these configurations can be visibility ordered.

https://hemanthrajhemu.github.io

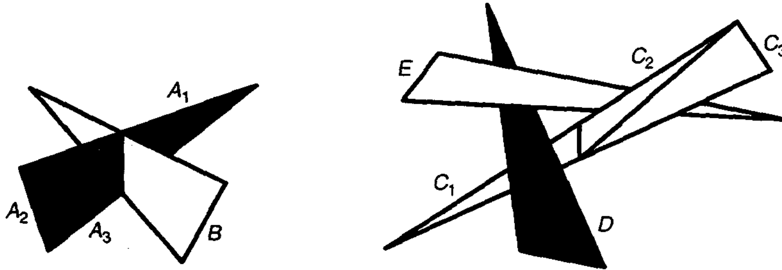


Figure 6.8: Refinements of Figure 6.7 are visibility ordered by (a) $A_1 \prec B \prec A_2 \prec A_3$ and (b) $C_1 \prec D \prec E \prec C_2 \prec C_3$.

triangles that results from splitting a set of triangles into pieces is known as a *refinement* of the original set.

The aforementioned ideas form the basis of our algorithm for depth sorting a set of triangles. First, we sort the triangles according to their maximum depths z^M from far to near. The resulting list represents a first approximation (or tentative) visibility ordering. Then we transform this into a true visibility ordering by incrementally shuffling the list and, whenever necessary, refining the list.

The algorithm works like this. Let S be the tentatively visibility ordered list, and let p be the first triangle in S. We wish to decide whether it is safe to paint p—that is, whether p does not obscure q for every $q \in S$. To do so, we compare p to each triangle q in list S whose z-extent overlaps that of p. For each triangle q, we ask whether it is possible for p to hide q. If it turns out that p obscures none of the triangles q, it is safe to paint p; hence we remove p from S and paint it, and then resume the algorithm using the first element in list S as the new p.

Alternatively, if it happens that p obscures some triangle q, we check whether q also obscures p or whether q has already been shuffled once. If either condition holds, we split q into pieces by the plane of p and then, within list S, replace q by its pieces (the refinement operation). (It is necessary to check whether q has already been shuffled in order to prevent an infinite loop to which configurations like those of Figure 6.7 would otherwise lead.) If neither condition holds, then the positions of p and q are interchanged in list S (the shuffle operation), and the algorithm resumes with q, now the first element of list S, serving as the new p.

Program `depthSort` depth sorts an array `tri` of `n` pointer-to-triangles and returns a visibility-ordered list of triangles. The tentatively ordered list of triangles is pointed to by local variable `s`, and the final depth-ordered list by variable `result`:

```
List<Triangle3D*> *depthSort(Triangle3D *tri[], int n)
{
   List<Triangle3D*> *result = new List<Triangle3D*>;
   Triangle3D **t = new (Triangle3D*)[n];
   for (int i = 0; i < n; i++)
      t[i] = new Triangle3D(*tri[i]);
   insertionSort(t, n, triangleCmp);
   List<Triangle3D*> *s = arrayToList(t, n);
   while (s->length() > 0) {   /* while */
```

https://hemanthrajhemu.github.io

```
      Triangle3D *p = s->first();
      Triangle3D *q = s->next();
      int hasShuffled = FALSE;
      for (; !s->isHead()&&overlappingExtent(p,q,2); q=s->next())
         if (mayObscure(p, q)) {
            if (q->mark || mayObscure(q, p))
               refineList(s, p);
            else {
               shuffleList(s, p);
               hasShuffled = TRUE;
               break;
            }
         }
      if (!hasShuffled) {
         s->first();
         s->remove();
         result->append(p);
      }
   }  /* while */
   return result;
}
```

Sorting is used to construct the initial tentatively ordered list. The comparison function `triangleCmp` compares two `Triangle3D`s according to their maximum depth:

```
int triangleCmp(Triangle3D *a, Triangle3D *b)
{
   if (a->boundingBox().dest.z > b->boundingBox().dest.z)
      return -1;
   else if (a->boundingBox().dest.z < b->boundingBox().dest.z)
      return 1;
   else
      return 0;
}
```

Function `arrayToList`, which was defined in Chapter 3, is then used to transform the sorted array of pointers into a list.

Function call `overlappingExtent(p,q,2)` returns `TRUE` if triangles `p` and `q` overlap in z-coordinate (the third argument specifies the coordinate via one of the indices 0, 1, or 2). The implementation of function `overlappingExtent` uses the fact that two intervals in the real number line intersect if and only if the left endpoint of one of the intervals is contained in the other interval:

```
bool overlappingExtent(Triangle3D *p, Triangle3D *q, int i)
{
   Edge3D pbox = p->boundingBox();
   Edge3D qbox = q->boundingBox();
   return (((pbox.org[i] <= qbox.org[i]) &&
```

https://hemanthrajhemu.github.io

```
            (qbox.org[i] <= pbox.dest[i])) ||
           ((qbox.org[i] <= pbox.org[i]) &&
            (pbox.org[i] <= qbox.dest[i])));
}
```

We shuffle list s with function shuffleList, which exchanges the first item p in the list with the item q occurring in the list's window:

```
void shuffleList(List<Triangle3D*> *s, Triangle3D *p)
{
   Triangle3D *q = s->val();
   q->mark = TRUE;
   s->val(p);
   s->first();
   s->val(q);
}
```

6.4.3 Comparing Two Triangles

The depthSort program uses the function call mayObscure(p,q) to determine whether triangle p potentially hides triangle q. Function mayObscure performs five tests in order of increasing complexity. As soon as one of these tests succeeds, p has been shown not to obscure q. Alternatively, if none of the five tests succeeds, then p potentially obscures q. The five tests are as follows:

1. Do the x-extents of p and q not overlap?
2. Do the y-extents of p and q not overlap?
3. Is p entirely behind or on the plane of q?
4. Is q entirely in front of or on the plane of p?
5. Do the projections of p and q not overlap?

Tests 3 and 4 are shown in Figure 6.9.

Most of the machinery for performing the tests is already in place. Tests 1 and 2 make use of the triangles' bounding boxes to compare x-extents and y-extents, respectively.

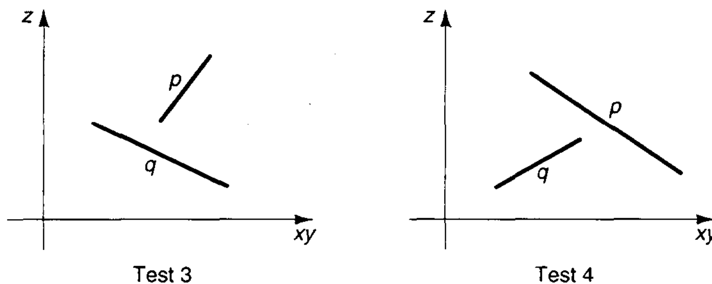


Figure 6.9: Two of the tests checked by mayObscure: (Test 3) p is behind the plane of q; (test 4) q is in front of the plane of p.

https://hemanthrajhemu.github.io

Test 3 classifies the three vertices of p with respect to the plane of q, and it succeeds if none of the vertices lies in front of the plane (in q's negative half-space). Similarly, test 4 succeeds if none of q's vertices lies behind the plane of p (in p's positive half-space). Test 5 is performed using function `projectionsOverlap`, which returns `TRUE` if the two triangles it is passed overlap in their projection.

Function `mayObscure` applies the tests to triangles `p` and `q` until one of the tests succeeds. If none succeeds, the function returns `TRUE`, indicating that it is possible for `p` to obscure `q`.

```
bool mayObscure(Triangle3D *p, Triangle3D *q)
{
   int i;
 // case 1
   if (!overlappingExtent(p, q, 0))
      return FALSE;
 // case 2
   if (!overlappingExtent(p, q, 1))
      return FALSE;
 // case 3
   for (i = 0; i < 3; i++)
      if ((*p)[i].classify(*q) == NEGATIVE)
         break;
   if (i == 3) return FALSE;
 // case 4
   for (i = 0; i < 3; i++)
      if ((*q)[i].classify(*p) == POSITIVE)
         break;
   if (i == 3) return FALSE;
 // case 5
   if (!projectionsOverlap(p, q))
      return FALSE;
   return TRUE;
}
```

Let us focus on test 5. To decide whether the projections of triangles p and q overlap, we first project the triangles into the xy-plane, producing the plane triangles P and Q. We then apply three tests to P and Q to see if they overlap. If any of the tests succeed, the projections of p and q overlap; otherwise they do not. The three tests are as follows:

1. Does some vertex of P lie in Q?
2. Does some vertex of Q lie in P?
3. Does some edge of P intersect some edge of Q?

The first test detects the case that P is contained in the interior of Q, and the second test detects the case that Q is contained in the interior of P. Overlap due to any remaining configuration is detected by the third test (although some of these configurations will first be picked up by the first or second test).

https://hemanthrajhemu.github.io

Function `projectionsOverlap` is passed triangles `p` and `q` and returns `TRUE` if and only if their projections overlap. It uses function `project` (defined in section 4.6) to obtain the projections of *p* and *q*:

```
bool projectionsOverlap(Triangle3D *p, Triangle3D *q)
{
   int answer = TRUE;
   Polygon *P = project(*p, 0, 1);
   Polygon *Q = project(*q, 0, 1);
   for (int i = 0; i < 3; i++, P->advance(CLOCKWISE))
      if (pointInConvexPolygon(P->point(), *Q))
         goto finish;
   for (i = 0; i < 3; i++, Q->advance(CLOCKWISE))
      if (pointInConvexPolygon(Q->point(), *P))
         goto finish;
   for (i = 0; i < 3; i++, P->advance(CLOCKWISE)) {
      double t;
      Edge ep = P->edge();
      for (int j = 0; j < 3; j++, Q->advance(CLOCKWISE)) {
         Edge eq = Q->edge();
         if (ep.cross(eq, t) == SKEW_CROSS)
            goto finish;
      }
   }
   answer = FALSE;
 finish:
   delete P;
   delete Q;
   return answer;
}
```

6.4.4 Refining a List of Triangles

In the depth-sort algorithm, refining list `s` involves splitting triangle `q` by the plane of candidate polygon `p` and then replacing `q` (within list `s`) by the two or three pieces into which it has been split. This is accomplished by function `refineList`, which is passed the current list `s` and the candidate triangle `p`. Triangle `q` is assumed to be the current item of list `s`:

```
void refineList(List<Triangle3D*> *s, Triangle3D *p)
{
   Triangle3D q = s->val();
   Triangle3D *q1, *q2, *q3;
   int nbrPieces = splitTriangleByPlane(*q, *p, q1, q2, q3);
   if (nbrPieces > 1) {
      delete s->remove();
      s->insert(q1);
      s->insert(q2);
```

https://hemanthrajhemu.github.io

```
      if (nbrPieces == 3)
         s->insert(q3);
   }
}
```

Triangle splitting is performed by function `splitTriangleByPlane`, defined next. Input parameters consist of triangle `q` to be split and splitter triangle `p`. The pieces of `q` produced by the function are passed back through the reference parameters `q1`, `q2`, and `q3`. (Parameter `q3` is not used if `q` is split into only two pieces.) The function returns the number of pieces it yields:

```
int splitTriangleByPlane(Triangle3D &q, Triangle3D &p,
      Triangle3D* &q1, Triangle3D* &q2, Triangle3D* &q3)
{
   Point3D crossingPts[2];
   int edgeIds[2], cl[3];
   double t;
   int nbrPts = 0;
   for (int i = 0; i < 3; i++)
      cl[i] = q[i].classify(p);
   for (i = 0; i < 3; i++)
      if (((cl[i]==POSITIVE) && (cl[(i+1)%3]==NEGATIVE)) ||
          ((cl[i]==NEGATIVE) && (cl[(i+1)%3]==POSITIVE))) {
         Edge3D e(q[i], q[(i+1)%3]);
         e.intersect(p, t);
         crossingPts[nbrPts] = e.point(t);
         edgeIds[nbrPts++] = i;
      }
   if (nbrPts == 0)
      return 1;
   Point3D a = q[edgeIds[0]];
   Point3D b = q[(edgeIds[0]+1) % 3];
   Point3D c = q[(edgeIds[0]+2) % 3];
   if (nbrPts == 1) {
      Point3D d = crossingPts[0];
      q1 = new Triangle3D(d, b, c, q.id);
      q2 = new Triangle3D(a, d, c, q.id);
   } else {
      Point3D d = crossingPts[0];
      Point3D e = crossingPts[1];
      if (edgeIds[1] == (edgeIds[0]+1)%3) {
         q1 = new Triangle3D(d, b, e, q.id);
         q2 = new Triangle3D(a, d, e, q.id);
         q3 = new Triangle3D(a, e, c, q.id);
      } else {
         q1 = new Triangle3D(a, d, e, q.id);
         q2 = new Triangle3D(b, e, d, q.id);
         q3 = new Triangle3D(c, e, b, q.id);
      }
```

https://hemanthrajhemu.github.io

```
    }
    return (nbrPts + 1);
}
```

In the first of two phases, function splitTriangleByPlane computes the points at which the plane of p crosses the edges of q. These crossing points are stored in array crossingPts, and the edges of q which contain the crossing points are stored in array edgeIds. Here an edge is identified by the identifier of its origin vertex (0, 1, or 2) within triangle q.

In its second phase, splitTriangleByPlane computes the pieces of q. The vertices of triangle q are labeled a, b, and c relative to the first crossing point d, as in Figure 6.10. Under this labeling scheme, triangle q is then split according to diagram (a) of Figure 6.10 if there is one crossing point d, and according to diagram (b) or (c) if there are two crossing points d and e.

With regard to performance of depth sorting, some configurations of n triangles in space require the algorithm to split the list into as many as $\Theta(n^2)$ pieces. Since the list s can become as long as $\Theta(n^2)$ and processing each candidate triangle p can take time proportional to the length of the list, depth sorting runs in $O(n^4)$ time at worst. However, such configurations are rare, and the algorithm performs well in practice. Furthermore, the list of polygons produced by depth sorting can be piped into any graphics system for display; the same cannot be said of all hidden surface removal methods, for some methods depend on the resolution of the display device.

6.5 Intersection of Convex Polygons

In this section we consider the problem of forming the intersection polygon $P \cap Q$ of two convex polygons P and Q. Except where noted, we will assume that the two polygons intersect non-degenerately: When two edges intersect, they do so at a single point which is not a vertex of either polygon. Given this assumption of non-degeneracy, intersection polygon $P \cap Q$ consists of alternating chains of P and Q. Each pair of consecutive chains is joined at an *intersection point*, at which the boundaries of P and Q cross (Figure 6.11).

There are several solutions to this problem that run in time linear in the total number of vertices. The algorithm we present here is especially clever and easy to implement. Given two convex polygons P and Q as input, the algorithm maintains a window over an edge of P and one over an edge of Q. The idea is to advance these windows around the polygon boundaries while growing the intersection polygon $P \cap Q$: The windows chase each other

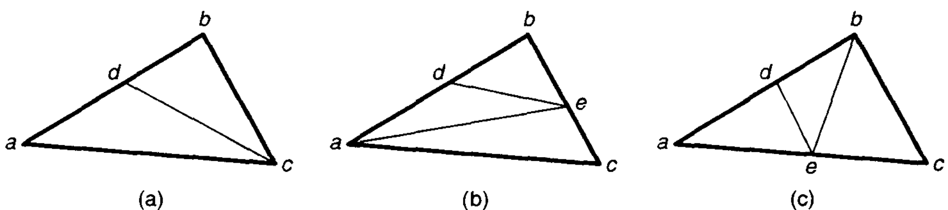


Figure 6.10: Splitting a triangle into (a) two pieces and (b and c) three pieces.

https://hemanthrajhemu.github.io

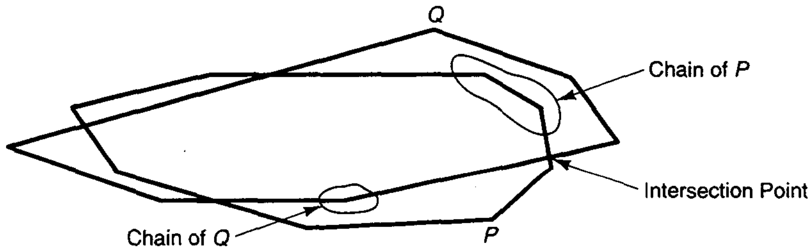


Figure 6.11: Structure of the intersection polygon $P \cap Q$.

clockwise around their respective polygons in search of intersection points. Since intersection points are discovered in the order they occur around $P \cap Q$, the intersection polygon is complete when some intersection point is discovered for the second time. Alternatively, if not a single intersection point is found after so many iterations, then the polygon boundaries do not intersect. In this case, simple tests are used to determine whether one of the polygons contains the other in its interior or if they do not intersect at all.

The notion of a *sickle* is handy for explaining the algorithm. In Figure 6.12, the sickles are the six shaded regions. Each is bounded by a chain from P and a chain from Q, and each terminates in two consecutive intersection points. The *inner chain* of a sickle is that chain which lies along the boundary of the intersection polygon. Observe that an intersection polygon is encircled by an even number of sickles whose inner chains alternate between P and Q.

In terms of sickles, the algorithm for finding the intersection polygon proceeds in two phases. In phase 1, P's window p and Q's window q are advanced clockwise until positioned over edges that belong to the same sickle. Each window starts off in arbitrary position. (For brevity, we will use p to denote both P's window as well as the edge in the window. Thus "the origin of p" refers to the origin of the edge in P's window, and the instruction "advance p" means we are to advance P's window to the next edge. Similarly, q denotes both Q's window as well as the edge in the window. We will also sometimes refer to edges p and q as *current edges*.)

In phase 2, p and q continue to be advanced clockwise, but this time moving in unison from sickle to adjacent sickle. Before either window leaves the current sickle for the next, edges p and q cross at the intersection point where the two sickles meet.

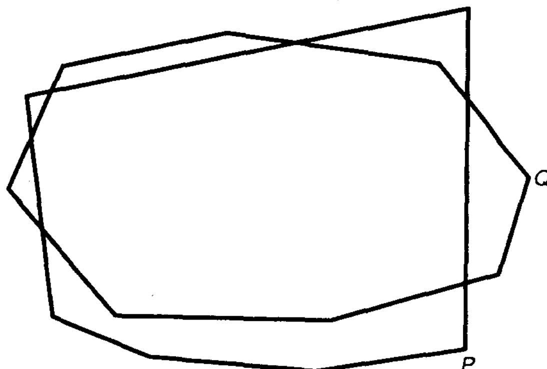


Figure 6.12: The sickles encircling the intersection polygon.

https://hemanthrajhemu.github.io

The intersection polygon is grown during this second phase. Whenever p is about to be advanced, edge p's destination endpoint is inserted into the intersection polygon *if edge p belongs to the current sickle's inner chain.* Similarly, when q is to be advanced, edge q's destination endpoint is inserted if q belongs to the current sickle's inner chain. Whenever p and q cross, the intersection point at which they cross is inserted into the intersection polygon.

The algorithm employs *advance rules* to decide which window to advance in each iteration. The advance rules depend on the following notion: An edge a is said to *aim* at edge b if the infinite line determined by b lies in front of a (Figure 6.13). Edge a aims at b if either of these conditions hold:

- $\vec{a} \times \vec{b} \geq 0$ and point `a.dest` does not lie to the right of $\vec{b}$, or
- $\vec{a} \times \vec{b} < 0$ and point `a.dest` does not lie to the left of $\vec{b}$.

Note that $\vec{a} \times \vec{b} \geq 0$ corresponds to the case in which the counterclockwise angle from vector $\vec{a}$ to $\vec{b}$ measures less than 180 degrees.

Function `aimsAt` returns `TRUE` if and only if edge `a` aims at edge `b`. The parameter `aclass` indicates the classification of endpoint `a.dest` relative to edge `b`. The parameter `crossType` equals `COLLINEAR` if and only if edges `a` and `b` are collinear:

```
bool aimsAt(Edge &a, Edge &b, int aclass, int crossType)
{
      Point2 va = a.dest - a.org;
      Point2 vb = b.dest - b.org;
      if (crossType != COLLINEAR) {
         if ((va.x * vb.y) >= (vb.x * va.y))
            return (aclass != RIGHT);
         else
            return (aclass != LEFT);
      } else {
         return (aclass != BEYOND);
      }
}
```

If edges a and b are collinear, a aims at b if endpoint `a.dest` does not lie beyond b. This is used to ensure that a is advanced, rather than b, when the two edges intersect degenerately

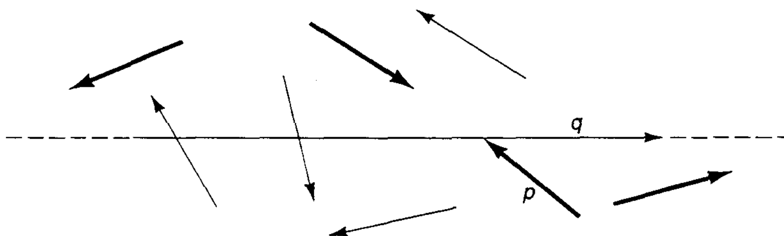


Figure 6.13: Only the thickened edges aim at edge q; the other edges do not.

https://hemanthrajhemu.github.io

in more than one point. By allowing a to "catch up" with b, we ensure that no intersection points are skipped over.

Let us return to our discussion of the advance rules. The advance rules are designed so that the intersection point which should be found next is not skipped over. They distinguish between the current edge which *may* contain the next intersection point and the current edge which *cannot possibly* contain the next intersection point; the window over the latter edge is then (safely) advanced. The advance rules distinguish between the following four cases, illustrated in Figure 6.14. In this account, edge a is considered *outside* edge b if endpoint `a.dest` lies to the left of b.

1. *p and q aim at each other*: Advance the window over whichever edge, p or q, is outside the other. In Figure 6.14a, we advance the window over p. The next intersection point cannot lie on edge p since p is outside the intersection polygon.

2. *p aims at q but q does not aim at p*: Insert p's destination endpoint into the intersection polygon if p is not outside q, and then advance window p. In Figure 6.14b, p cannot contain the *next* intersection point (although it may contain *some* intersection point if p is not outside q). The figure shows the situation in which edge p, whose window is to be advanced, *is not* outside edge q.

3. *q aims at p but p does not aim at q*: Insert q's destination endpoint into the intersection polygon if q is not outside p, and then advance window q (Figure 6.14c). This case is symmetric to the previous case. The figure shows the situation in which edge q, whose window is to be advanced, *is* outside edge p.

4. *p and q do not aim at each other*: Advance the window over whichever current edge is outside the other. In Figure 6.14d we advance window p since edge p is outside edge q.

Figure 6.15 illustrates the algorithm at work. Each edge bears label i if reached in iteration i (some edges bear two labels since they are reached twice). The two initial edges are labeled 0. In this figure, phase 2—when the two current edges belong to the same sickle—begins after three iterations.

Program `convexPolygonIntersect` implements the algorithm. The program is passed polygons `P` and `Q` and returns a pointer to the resulting intersection polygon `R`. The call to function `advance` is used to advance one of the two current edges and to insert

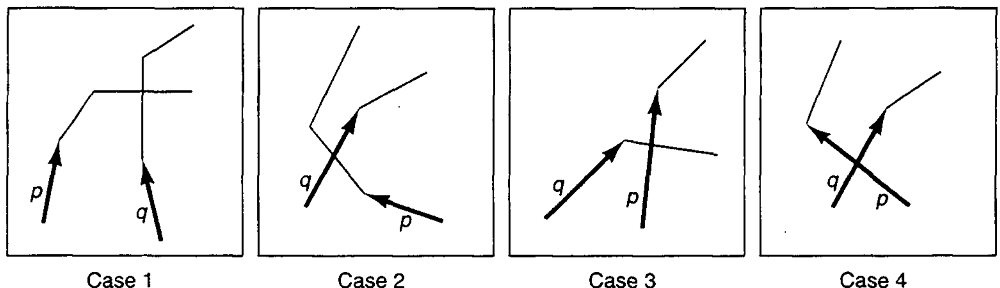


Figure 6.14: The four advance rules: (Case 1) Advance p, (Case 2) advance p, (Case 3) advance q, and (Case 4) advance p.

https://hemanthrajhemu.github.io

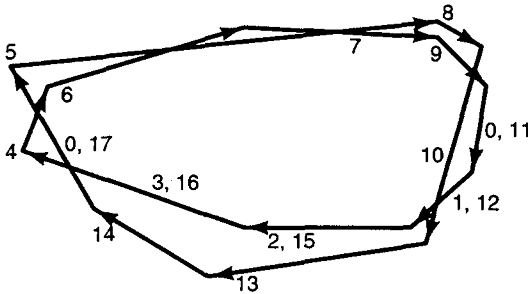


Figure 6.15: Finding the intersection polygon. An edge bears label i if it is reached in iteration i. The two initial edges are labeled 0.

conditionally the edge's destination endpoint into polygon R. The windows built into class `Polygon` are used.

```
enum { UNKNOWN, P_IS_INSIDE, Q_IS_INSIDE };

Polygon *convexPolygonIntersect(Polygon &P, Polygon &Q)
{
   Polygon *R;
   Point iPnt, startPnt;
   int inflag = UNKNOWN;
   int phase = 1;
   int maxItns = 2 * (P.size() + Q.size());
   for (int i = 1; (i<=maxItns) || (phase==2); i++) { // for
      Edge p = P.edge();
      Edge q = Q.edge();
      int pclass = p.dest.classify(q);
      int qclass = q.dest.classify(p);
      int crossType = crossingPoint(p, q, iPnt);
      if (crossType == SKEW_CROSS) {
         if (phase == 1) {
            phase = 2;
            R = new Polygon;
            R->insert(iPnt);
            startPnt = iPnt;
         } else if (iPnt != R->point()) {
            if (iPnt != startPnt)
               R->insert(iPnt);
            else
               return R;
         }
          if (pclass==RIGHT) inflag = P_IS_INSIDE;
          else if (qclass==RIGHT) inflag = Q_IS_INSIDE;
          else inflag = UNKNOWN;
      } else if ((crossType==COLLINEAR) &&
                 (pclass!=BEHIND) &&
                 (qclass!=BEHIND))
```

https://hemanthrajhemu.github.io

```
            inflag = UNKNOWN;
        bool pAIMSq = aimsAt(p, q, pclass, crossType);
        bool qAIMSp = aimsAt(q, p, qclass, crossType);
        if (pAIMSq && qAIMSp) {
            if ((inflag==Q_IS_INSIDE) ||
                ((inflag==UNKNOWN) && (pclass==LEFT)))
                advance(P, *R, FALSE);
            else
                advance(Q, *R, FALSE);
        } else if (pAIMSq) {
            advance(P, *R, inflag==P_IS_INSIDE);
        } else if (qAIMSp) {
            advance(Q, *R, inflag==Q_IS_INSIDE);
        } else {
            if ((inflag==Q_IS_INSIDE) ||
                ((inflag==UNKNOWN) && (pclass==LEFT)))
                advance(P, *R, FALSE);
            else
                advance(Q, *R, FALSE);
        }
    }   // for
    if (pointInConvexPolygon(P.point(), Q))
        return new Polygon(P);
    else if (pointInConvexPolygon(Q.point(), P))
        return new Polygon(Q);
    return new Polygon;
}
```

If $2(|P| + |Q|)$ iterations are performed without some intersection point being found, the main loop is exited since the polygon boundaries are then known not to cross. The subsequent calls to `pointInConvexPolygon` are used to determine whether $P \subset Q$, $Q \subset P$, or $P \cap Q = \emptyset$. Alternatively, if some intersection point `iPnt` is found, then the algorithm proceeds to grow the intersection polygon `R`, stopping only when `iPnt` is reached for the second time.

Variable `inflag` indicates which of the two input polygons is currently inside the other—that is, the polygon whose current edge lies in the inner chain of the current sickle. Moreover, `inflag` is set to `UNKNOWN` during phase 1, and whenever the two current edges are collinear and overlap. It is updated whenever a new intersection point is discovered.

Procedure `advance` advances the current edge of polygon `A`, representing either `P` or `Q`. The procedure also inserts the edge's destination endpoint x into intersection polygon `R`, if `A` is inside the other polygon and x was not the last point inserted into `R`:

```
void advance(Polygon2 &A, Polygon2 &R, int inside)
{
    A.advance(CLOCKWISE);
    if (inside && (R.point() != A.point()))
        R.insert(A.point());
}
```

https://hemanthrajhemu.github.io

6.5.1 Analysis and Correctness

The correctness proof bears out what is most remarkable about this algorithm: that the same set of advance rules works for *both* phases. The advance rules get p and q into the same sickle, and then they advance p and q in unison from sickle to sickle.

Correctness of the algorithm follows from two assertions:

1. If current edges p and q belong to the same sickle, then the next intersection point—at which the sickle terminates—will be found, and it will be found next.
2. If the boundaries of P and Q intersect, current edges p and q will cross at some intersection point after no more than $2(|P| + |Q|)$ iterations.

Assertion 2 ensures that the algorithm will find some intersection point, if one exists. Since edges p and q belong to the same sickle if they cross, assertion 1 then ensures that the remaining intersection points will be found in order.

Let us show assertion 1 first. Suppose that p and q belong to the same sickle and that q reaches the next intersection point first, before p. We will show that q then remains stationary while p catches up to the intersection point via a sequence of advances. Two cases can occur. First, assume that p is outside q (Figure 6.16a). In this case, q remains fixed while p is advanced by zero or more applications of rule 4, then by zero or more applications of rule 1, and then by zero or more applications of rule 2. In the second case, assume that p is not outside q (Figure 6.16b). In this case, q remains fixed while p is advanced by zero or more applications of rule 2. In the symmetric situation, where p reaches the next intersection point before q, edge q remains stationary while p catches up. This is shown as before, where the roles of p and q are swapped, and rule 3 replaces rule 2. Assertion 1 follows.

To show assertion 2, let us assume that the boundaries of P and Q intersect. After $|P| + |Q|$ iterations, either p or q must have traversed full circle around its polygon. Let us assume that p has. At some time, p must have been positioned such that it contains an intersection point at which polygon Q passes from the outside of P to its inside. This is the case because there are at least two intersection points and they alternate in direction of crossing. Let q be the edge in Q's window when p was so positioned.

In Figure 6.17, the boundary of Q is partitioned into two chains C_r and C_s. The first chain, C_r, terminates in edge q_r, the edge of Q that enters P through edge p. The other chain, C_s, terminates in edge q_s, whose destination vertex both lies to the right of, and

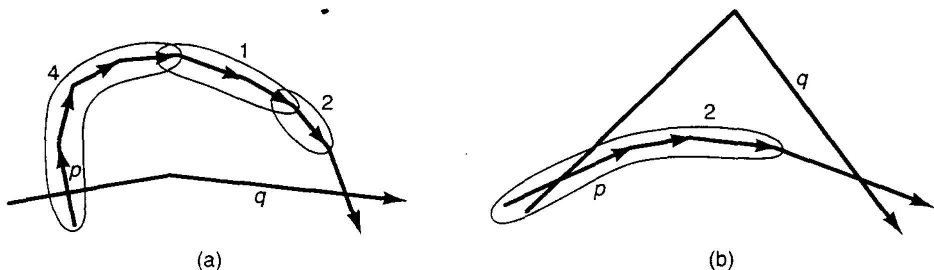


Figure 6.16: Advancing to the next intersection point.

https://hemanthrajhemu.github.io

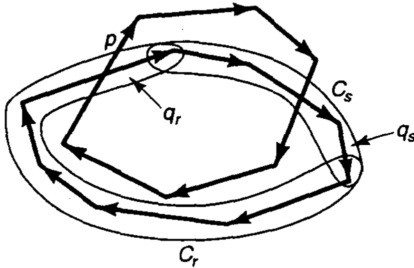


Figure 6.17: Illustrations for the proof that an intersection point is found if the boundaries of P and Q intersect.

is farthest from, the infinite line determined by edge p. There are two cases to consider, depending on which of the two chains edge q belongs to:

Case 1 $[q \in C_r]$ In this case, p remains fixed while q advances by zero or more applications of rule 3, then rule 4, then rule 1, and finally rule 3, at which time the intersection point is found.

Case 2 $[q_j \in C_s]$ In this case, q remains fixed while p advances by zero or more applications of rule 2, then rule 4, then rule 1, and finally rule 2, at which time p will be inside q. At this point, p and q may both be advanced a number of times—however, q cannot advance beyond its next intersection point until p first reaches q's previous intersection point (if p has not done so already). Since p and q end up in the same sickle, assertion 1 guarantees that after some number of additional advances, they will cross at the intersection point at which this sickle terminates.

To see why $2(|P|+|Q|)$ iterations are enough to find some intersection point, observe that the initial positions of p and q used to show assertion 2—the boundary of Q entering P through p, and q situated anywhere—were arrived at after no more than $|P|+|Q|$ iterations. (Actually, either this situation or the symmetric situation, in which the roles of p and q are swapped, is achieved after this many iterations.) Since neither p nor q then advances full circle around its polygon before reaching the first intersection point, no more than $|P|+|Q|$ additional iterations are needed.

6.5.2 Robustness

Our algorithm for finding the intersection of two convex polygons is most susceptible to round-off error when the two polygons intersect at a point that is a vertex of one or both polygons. One problem is that intersection points may be missed. In Figure 6.13, edges p and q intersect at point x, the destination endpoint of p. Using exact arithmetic, the parametric value of x along edge p equals one. However, using floating-point arithmetic, the parametric value actually calculated might exceed one by a slight amount, locating x *beyond* edge p. The intersection point would go undetected.

Function `crossingPoint`, used by program `convexPolygonIntersect` to compute the intersection point of two edges, attempts to resolve these sort of difficulties. Given two edges e and f, the function first computes the point at which *infinite lines* e and f intersect. If this point lies in the vicinity of one of the edges' four endpoints, the endpoint is

https://hemanthrajhemu.github.io

taken to be the point of intersection. As implemented, the function works with parametric values rather than points. By extending the range of parametric values along edge f, the edge is lengthened by distance `EPSILON2` in both directions. If the intersection point which would be computed lies within `EPSILON2` of one of f's endpoints, the intersection point is "snapped back" to the endpoint. Otherwise the same is done for edge e.

Function `crossingPoint` returns one of the values `COLLINEAR`, `PARALLEL`, `SKEW_NO_CROSS`, or `SKEW_CROSS` to indicate the relationship between edges `e` and `f`. If `SKEW_CROSS` is returned, indicating that the edges intersect at a point, their point of intersection is passed back through reference paramter `p`:

```
#define EPSILON2   1E-10

int crossingPoint(Edge &e, Edge &f, Point &p)
{
   double s,t;
   int classe= e.intersect(f, s);
   if ((classe==COLLINEAR) || (classe==PARALLEL))
      return classe;
   double lene = (e.dest-e.org).length();
   if ((s < -EPSILON2*lene) || (s > 1.0+EPSILON2*lene))
      return SKEW_NO_CROSS;
   f.intersect(e, t);
   double lenf = (f.org-f.dest).length();
   if ((-EPSILON2*lenf <= t) && (t <= 1.0+EPSILON2*lenf)) {
      if (t <= EPSILON2*lenf) p = f.org;
      else if (t >= 1.0-EPSILON2*lenf) p = f.dest;
      else if (s <= EPSILON2*lene) p = e.org;
      else if (s >= 1.0-EPSILON2*lene) p = e.dest;
      else p = f.point(t);
      return SKEW_CROSS;
   } else
      return SKEW_NO_CROSS;
}
```

If it relies on function `crossingPoint` to calculate points of intersection, our program for finding the intersection of convex polygons works even when the polygons intersect at vertices. This is important to us, for in Chapter 8 we will use the program in applications which unavoidably give rise to this special case. However, note that our program can fail if a vertex of one polygon lies very close—within `EPSILON2`—to the boundary of the other polygon, without actually touching the boundary.

6.6 Finding Delaunay Triangulations

A *triangulation* of a finite point set S is a triangulation of the convex hull $\mathcal{CH}(S)$ that uses all the points of S. The line segments of the triangulation may not cross—they may meet only at shared endpoints, points of S. Since the line segments enclose triangles, we usually

https://hemanthrajhemu.github.io

refer to them as edges. Figure 6.18 depicts two triangulations of the same set of points (ignore the circles in the figure for the moment).

Given a point set S, we have seen that the points of S can be partitioned into *boundary points*—those points of S which lie on the boundary of the convex hull $\mathcal{CH}(S)$—and *interior points*—those points which lie in the interior of $\mathcal{CH}(S)$. The edges of a triangulation of S can be classified similarly, as *hull edges* and *interior edges*. The hull edges are those edges that lie along the boundary of the convex hull $\mathcal{CH}(S)$, and the interior edges are the remaining edges, those that pierce the convex hull interior. Note that every hull edge connects two boundary points, whereas an interior edge can connect two points of either type; in particular, if an interior edge connects two boundary points, it is a chord of $\mathcal{CH}(S)$. Observe also that every edge of the triangulation is met by two *faces*: each interior edge by two triangles, and each hull edge by one triangle and the unbounded plane.

All point sets except the most trivial ones admit more than one triangulation. Remarkably, every triangulation of a given point set contains the same number of triangles, as the following theorem indicates:

Theorem 3 (Point-Set Triangulation Theorem) *Suppose point set S contains $n \geq 3$ points, not all collinear. Suppose further that i of the points are interior [lying in the interior of $\mathcal{CH}(S)$]. Then every triangulation of S contains exactly $n + i - 2$ triangles.*

To see why this theorem is true, first consider triangulating the $n - i$ boundary points. Since they are the vertices of a convex polygon, any such triangulation contains $(n - i) - 2$ triangles. (This is not hard to see; in Chapter 8 we will show that every triangulation of *any* m-sided polygon—convex or nonconvex—consists of $m - 2$ triangles.) Now consider incorporating the remaining i interior points into the triangulation, one at a time. We claim that adding each such point increases the number of triangles by two. The two cases illustrated in Figure 6.19 can occur. First, if the point falls in the interior of some triangle, the triangle is replaced by three new triangles. Second, if the point falls on some edge of the triangulation, each of the two triangles that meet the edge is replaced by two new triangles. It follows that after all i points are inserted, the total number of triangles is $(n - i - 2) + (2i)$, or simply $n + i - 2$.

In this section we present an algorithm to construct a special kind of triangulation known as a *Delaunay triangulation*. Such triangulations are well balanced in the sense that the triangles tend toward equiangularity. In Figure 6.18, for example, triangulation (a) is

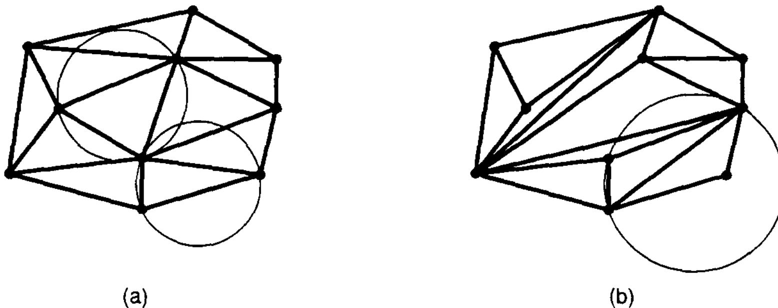


Figure 6.18: Two triangulations of the same set of points.

https://hemanthrajhemu.github.io

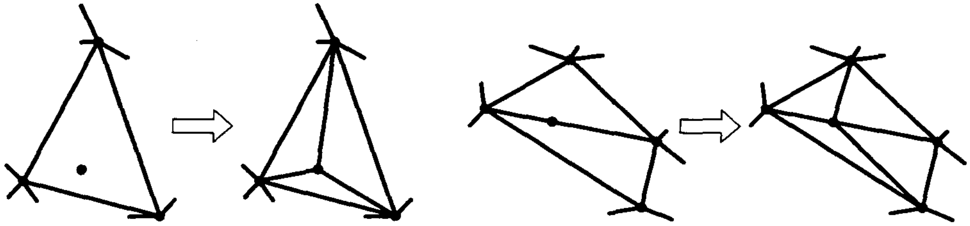


Figure 6.19: The two ways in which an interior site can be incorporated into a triangulation.

Delaunay whereas triangulation (b), which contains some long "slivers," is not Delaunay. Figure 6.20 shows the Delaunay triangulation of a large point set.

To define *Delaunay triangulation*, we need some new definitions. A set of points is *cocircular* if there exists some circle on whose boundary all the points lie. If the circle is unique, it is called the *circumcircle* of the points. The circumcircle of a triangle is simply the circumcircle of its three (non-collinear) vertices. A circle is said to be *point free* with respect to a given point set S if none of the points of S lies in the circle's interior. Points of S may, however, lie along the boundary of a point-free circle.

A triangulation of point set S is a *Delaunay triangulation* if the circumcircle of every triangle is point free. In triangulation (a) of Figure 6.18, the two circumcircles which have been drawn are clearly point free (you might want to draw the remaining circumcircles to verify that they are also point free). Since the circumcircle shown in triangulation (b) is not point free, the triangulation is not Delaunay.

We will make two assumptions about point set S to simplify the triangulation algorithm. First, to ensure that some triangulation exists, we will assume that S contains at least three points, not all collinear. Second, to ensure that the Delaunay triangulation is unique, we will assume that no four points of S are cocircular. It is easy to see that without this latter assumption, the Delaunay triangulation need not be unique: Four cocircular points admit two different Delaunay triangulations.

Our algorithm works by growing a current triangulation, triangle by triangle. Initially the current triangulation consists of a single hull edge, and at completion the current triangulation equals the Delaunay triangulation. In each iteration, the algorithm seeks a new triangle which attaches to the *frontier* of the current triangulation.

The definition of *frontier* depends on the following scheme, which classifies the edges of the Delaunay triangulation relative to the current triangulation. Every edge is either *dormant*, *live*, or *dead*:

- *Dormant edges*: An edge of the Delaunay triangulation is dormant if it has not yet been discovered by the algorithm.
- *Live edges*: An edge is live if it has been discovered but only one of its faces is known.
- *Dead edges*: An edge is dead if it has been discovered and both of its faces are known.

Initially only a single hull edge is live—the unbounded plane is known to meet it—and all the remaining edges are dormant. As the algorithm proceeds, edges transition from dormant to live, then from live to dead. The *frontier* at each stage consists of the set of live edges.

In each iteration, we select any edge e of the frontier and process it, which consists of seeking edge e's unknown face. If this face turns out to be some triangle t determined

https://hemanthrajhemu.github.io

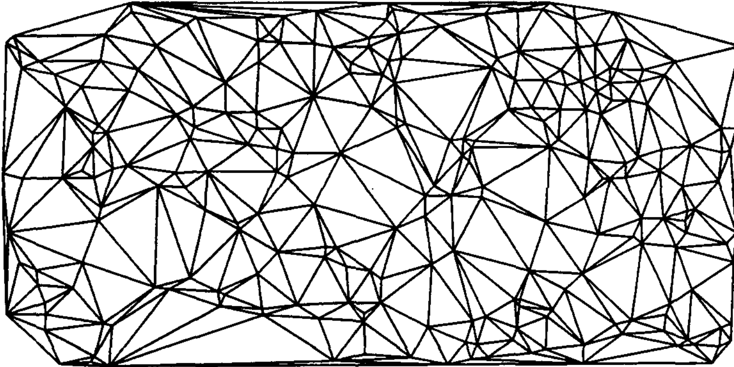


Figure 6.20: A Delaunay triangulation of 250 points chosen at random within a rectangle. The triangulation contains 484 triangles.

by the endpoints of e and some third vertex v, edge e dies since both of its faces are now known. Moreover, each of the other two edges of triangle t transition to the next state: from dormant to live, or from live to dead. Here vertex v is called the *mate* of edge e. Alternatively, if the unknown face turns out to be the unbounded plane, edge e simply dies. In this case e has no mate.

Figure 6.21 illustrates the algorithm. In the figure, the action proceeds top to bottom, then left to right. The frontier in each stage is darkened.

The following program, `delaunayTriangulate`, implements the algorithm. The program is handed an array `s` of `n` points and returns a list of triangles representing its Delaunay triangulation:

```
List<Polygon*> *delaunayTriangulate(Point s[], int n)
{
   Point p;
   List<Polygon*> *triangles = new List<Polygon*>;
   Dictionary<Edge*> frontier(edgeCmp);
   Edge *e = hullEdge(s, n);
   frontier.insert(e);
   while (!frontier.isEmpty()) {
      e = frontier.removeMin();
      if (mate(*e, s, n, p)) {
         updateFrontier(frontier, p, e->org);
         updateFrontier(frontier, e->dest, p);
         triangles->insert(triangle(e->org, e->dest, p));
      }
      delete e;
   }
   return triangles;
}
```

The triangles which make up the triangulation are maintained in the list `triangles`. The frontier is represented by the dictionary `frontier` of live edges. Each edge is directed such that its unknown face (yet to be sought) lies to the right of the edge. The comparison

https://hemanthrajhemu.github.io

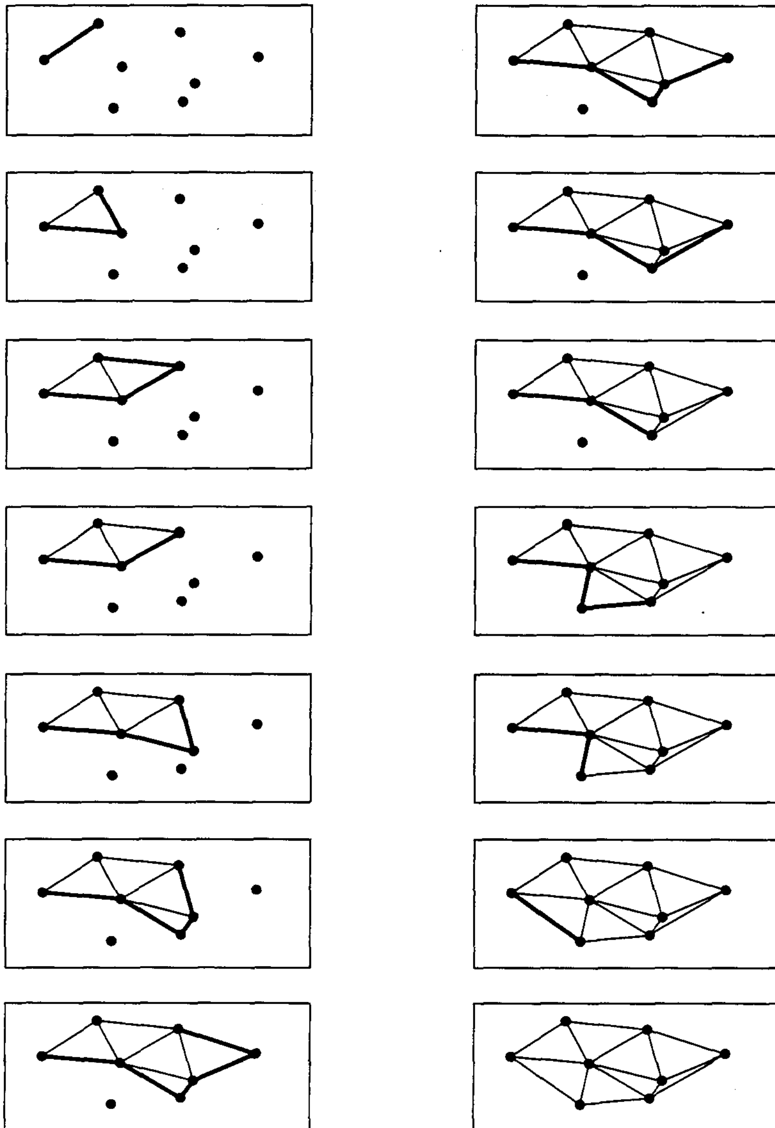


Figure 6.21: Growing a Delaunay triangulation. The edges of the frontier are highlighted.

function edgeCmp is used to perform look-up in the dictionary. It compares two edges' origins and, if these are the same, then compares their destinations:

```
int edgeCmp(Edge *a, Edge *b)
{
   if (a->org < b->org) return -1;
   if (a->org > b->org) return 1;
   if (a->dest < b->dest) return -1;
```

https://hemanthrajhemu.github.io

```
   if (a->dest > b->dest) return 1;
   return 0;
}
```

How does the frontier change from one iteration to the next, and how does function `updateFrontier` update the dictionary to reflect these changes? When a new triangle t attaches to the frontier, the state of the triangle's three edges changes. The edge of t which attaches to the frontier changes from live to dead. Function `updateFrontier` can ignore this edge since it will already have been removed from the dictionary by the call to `removeMin`. Each of the two remaining edges of t changes state from dormant to live if the edge is not already in the dictionary, or from live to dead if the edge is already in the dictionary. Figure 6.22 illustrates both cases. In the figure, we process the live edge $\overrightarrow{af}$ and, upon discovering that point b is its mate, add triangle $\triangle afb$ to the current triangulation. Then we look up edge $\overrightarrow{fb}$ in the dictionary—since it is not present, it has just been discovered for the first time, so its state changes from dormant to live. To update the dictionary, we flip $\overrightarrow{fb}$ so its unknown face lies to its right and then insert the edge into the dictionary. Next we look up edge $\overrightarrow{ba}$ in the dictionary—since it is present, it is already live (its known face is triangle $\triangle abc$). Since its unknown face, triangle $\triangle afb$, has just been discovered, we then remove the edge from the dictionary.

Function `updateFrontier` updates dictionary `frontier`, where the edge from point `a` to point `b` changes state:

```
void updateFrontier(Dictionary<Edge*> &frontier,
                    Point &a, Point &b)
{
   Edge *e = new Edge(a, b);
   if (frontier.find(e))
      frontier.remove(e);
   else {
      e->flip();
      frontier.insert(e);
   }
}
```

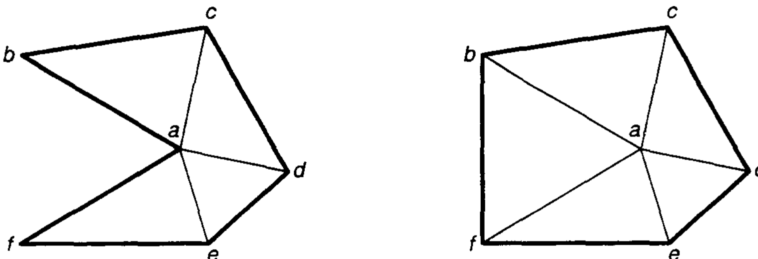


Figure 6.22: Attaching triangle $\triangle afb$ to live edge $\overrightarrow{af}$.

https://hemanthrajhemu.github.io

Function `hullEdge` returns a hull edge from among the `n` points of array `s`. The function essentially implements the initialization and first iteration of the gift-wrapping method:

```
Edge *hullEdge(Point s[], int n)
{
   int m = 0;
   for (int i = 1; i < n; i++)
      if (s[i] < s[m])
         m = i;
   swap(s[0], s[m]);
   for (m = 1, i = 2; i < n; i++) {
      int c = s[i].classify(s[0], s[m]);
      if ((c == LEFT) || (c == BETWEEN))
         m = i;
   }
   return new Edge(s[0], s[m]);
}
```

Function `triangle` simply constructs and returns a polygon over the three points it is passed:

```
Polygon *triangle(Point &a, Point &b, Point &c)
{
   Polygon *t = new Polygon;
   t->insert(a);
   t->insert(b);
   t->insert(c);
   return t;
}
```

6.6.1 Finding the Mate of an Edge

Let us turn our attention to the problem solved by function `mate`, that of determining whether a given live edge has a mate and, if so, finding it. Consider this: Any edge $\overrightarrow{ab}$ determines the infinite family of circles whose boundaries contain both endpoints a and b. Let $C(a,b)$ denote this family of circles (Figure 6.23).

The centers of the circles in $C(a,b)$ lie along edge $\overrightarrow{ab}$'s perpendicular bisector and can be put into one-to-one correspondence with the points of this bisector. To specify circles of the family, we parameterize the perpendicular bisector and identify each circle by the parametric value of the circle's center. The machinery of Chapter 4 provides a natural parameterization: Rotate edge $\overrightarrow{ab}$ 90 degrees into its perpendicular bisector and then use the parameterization along this edge. In Figure 6.23, we use C_r to denote the circle corresponding to parametric value r.

How do we find the mate of some live edge $\overrightarrow{ab}$ from among the points of S? Suppose that C_r is the circumcircle of $\overrightarrow{ab}$'s known face (in Figure 6.24, triangle $\triangle abc$ is the known

https://hemanthrajhemu.github.io

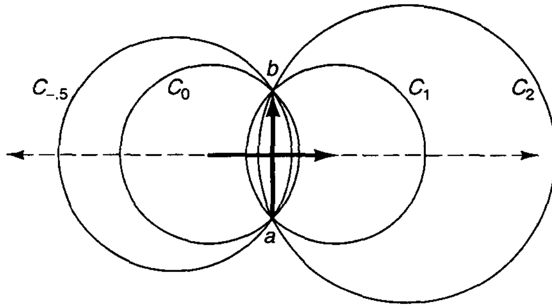


Figure 6.23: Four circles of the family $C(a,b)$ determined by edge $\overline{ab}$, and their parametric values.

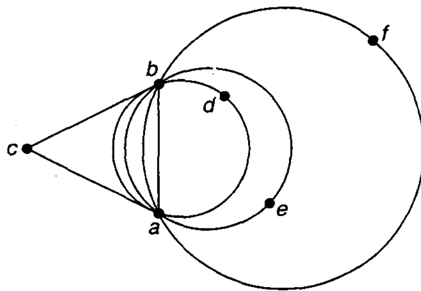


Figure 6.24: Finding the mate (d) of edge $\overline{ab}$.

face). If $\overrightarrow{ab}$'s known face is unbounded, then $r = -\infty$ and C_r is the half-plane to the left of $\overrightarrow{ab}$. We seek the smallest value $t > r$ such that some point of S (other than a or b) lies in the boundary of C_t. If no such value t exists, then edge $\overrightarrow{ab}$ has no mate. More picturesquely, this is like blowing a two-dimensional bubble through edge $\overrightarrow{ab}$. If the bubble eventually reaches some point of S, this point is the mate of edge $\overrightarrow{ab}$ (point d of Figure 6.24). Alternatively, if no point of S is reached and the bubble expands to fill the half-plane to the right of edge $\overrightarrow{ab}$, then $\overrightarrow{ab}$ has no mate.

Why does this work? Let C_r denote the circumcircle of edge $\overrightarrow{ab}$'s known face, and C_t the circumcircle of edge $\overrightarrow{ab}$'s unknown face. Here $t > r$, and $t = \infty$ if $\overrightarrow{ab}$ has no mate. Is circle C_t point free, as desired? To the left of $\overrightarrow{ab}$, C_t must be point free since C_r is point free and the portion of C_t which lies to the left of $\overrightarrow{ab}$ is contained in C_r. To the right of edge $\overrightarrow{ab}$, C_t must also be point free because, were some point q to lie in its interior, q would lie in the boundary of some circle $C_s \in C(a,b)$, where $r < s < t$, contradicting our choice of t. In our bubble analogy, the expanding bubble would reach point q before reaching the mate of edge $\overrightarrow{ab}$.

To find the mate of edge $\overrightarrow{ab}$, we consider only those points $p \in S$ that lie to the right of $\overrightarrow{ab}$. The center of the circle circumscribing any three points a, b, and p lies at the intersection of the perpendicular bisectors of $\overrightarrow{ab}$ and $\overrightarrow{bp}$. (Here we use the fact that the perpendicular bisectors of a triangle's edges intersect at the center of the triangle's circumcircle.) Rather than compute the center point of the circle, we compute its parametric

https://hemanthrajhemu.github.io

value along the perpendicular bisector of edge $\overrightarrow{ab}$. This way we can keep track of the smallest parametric value found so far.

This method is implemented by function `mate`, which returns `TRUE` if edge `e` has a mate and `FALSE` if it does not. If the mate exists, it is passed back through reference parameter `p`:

```
bool mate(Edge &e, Point s[], int n, Point &p)
{
   Point *bestp = NULL;
   double t, bestt = FLT_MAX;
   Edge f = e;
   f.rot();    // f is the perpendicular bisector of e
   for (int i = 0; i < n; i++)
      if (s[i].classify(e) == RIGHT) {
         Edge g(e.dest, s[i]);
         g.rot();
         f.intersect(g, t);
         if (t < bestt) {
            bestp = &s[i];
            bestt = t;
         }
      }
   if (bestp) {
      p = *bestp;
      return TRUE;
   }
   return FALSE;
}
```

In function `mate`, variable `bestp` points to the best point examined so far, and `bestt` holds the parametric value of the circle whose boundary contains the point. Note that only those points to the right of edge `e` are considered.

This algorithm for computing the Delaunay triangulation of a set of n points runs in $O(n^2)$ time because one edge leaves the frontier in each iteration. Since every edge leaves the frontier exactly once—every edge enters the frontier once and later leaves it, never to return—the number of iterations equals the number of edges in the Delaunay triangulation. Now the point-set triangulation theorem implies that any triangulation contains no more than $O(n)$ edges, so the algorithm performs $O(n)$ iterations. Because it spends $O(n)$ time per iteration, the algorithm runs in $O(n^2)$ time.

6.7 Chapter Notes

The gift-wrapping method, also known as Jarvis's march after the manner in which it marches around the convex hull boundary, is presented in [43]. The same basic idea can be

https://hemanthrajhemu.github.io

used to find the convex hull of points in higher dimensions [17]; in three dimensions, the method reminds us of how we would go about wrapping a gift. Graham scan is presented in [33].

There are numerous algorithms for finding convex hulls, and this book covers several: insertion hull, which runs in $O(n^2)$ time, where n is the number of points; plane sweep in $O(n \log n)$ time; Graham scan in $O(n \log n)$ time; gift wrapping in $O(nh)$ time, where the convex hull contains $h \leq n$ vertices; and merge hull in $O(n \log n)$ time. One interesting algorithm we will not cover is called *quick hull.* Like the quicksort algorithm after which it is modeled, quick hull takes $O(n^2)$ time in the worst case but $O(n \log n)$ time in the expected case [14, 25, 34]. An optimal convex hull finding algorithm was developed by Kirkpatrick and Seidel [46]. Where the convex hull it produces contains h vertices, the algorithm runs in $O(n \log h)$ time in the worst case.

Because hidden surface removal is usually indispensable for realistic three-dimensional graphics, the problem has been the focus of much research, leading to numerous solutions. Solutions vary with regard to the types of scene models they accommodate, efficiency, degree of realism, and other factors. The depth-sorting method presented in this chapter is from [59]. Other well-known methods include z-buffering [16], Warnock's area subdivision method [86], the Weiler-Atherton "cookie-cutter" method [88], scanline methods [50, 87], and ray tracing. (The computer graphics texts [28, 39, 68] also provide accounts of these algorithms.) In *z-buffering*, the depth of the object displayed by each pixel is maintained in a buffer of depth values (the *z-buffer*). When a new object is to be painted, pixels are selectively updated—only those pixels displaying a more distant object are overwritten by the new object. The z-buffer is also updated with the new (closer) depth values. Because it is both simple and general (in the sense of accommodating a wide range of scene models), z-buffering has been implemented in hardware in several recent graphics systems. In ray tracing, another hidden surface removal method, simulated rays of light are cast into the scene. Ray tracing can be used to create images which include such features as transparency, reflection, specular highlights, and shadows.

The algorithm for finding the intersection of two convex polygons P and Q is presented in [62, 61], although our presentation more closely follows [66]. An earlier algorithm for the same problem is given in [75]. In this method, a vertical line is drawn through every vertex, thereby partitioning the plane into vertical slabs and each polygon into triangles and trapezoids. The intersection problem is then solved within each slab in turn, and the resulting polygonal pieces assembled. Since the intersection of two polygons of bounded size can be computed in constant time and there are no more than $|P| + |Q|$ slabs, this algorithm, like the one we have presented, runs in $O(|P| + |Q|)$ time.

The Delaunay triangulation is dual to the Voronoi diagram, a polygonal decomposition of the plane which assumes a central role in computational geometry. The connection will be explored in Chapter 8, where an algorithm for constructing Voronoi diagrams will be presented. The Delaunay triangulation algorithm presented in this chapter is based on [5, 55]. The algorithm is lifted to three-dimensional space in [24], and a data structure appropriate for lifting it to d-dimensional space is given in [12]. An $O(n \log n)$-time algorithm for constructing Delaunay triangulations in the plane using divide and conquer is presented in [36]. A survey of Voronoi diagrams and Delaunay triangulations is provided by [4].

https://hemanthrajhemu.github.io

6.8 Exercises

1. Modify program `giftwrapHull` so the vertices of the convex hull $\mathcal{CH}(S)$ it produces consist of all boundary points of set S, not just the extreme points.
2. Modify `grahamScan` to do as described in the previous question.
3. The *depth* of a point p in finite point set S is the number of convex hulls that must be removed until p becomes a boundary point. For instance, the boundary points of S are at depth zero, and those points that become boundary points when the boundary points of S are removed are at depth one. The brute-force approach to determine the depth of all points repeatedly finds the convex hull of the point set and removes the boundary points from the set, until the set is empty. Modify `giftwrapHull` so it computes the depth of every point in $O(n^2)$ time.
4. The *diameter* of a point set is the maximum distance between any pair of points.
   (a) Show that the diameter is realized by a pair of extreme points.
   (b) Give an $O(n \log n)$-time algorithm for computing the diameter of a set of n points in the plane.
5. Describe a configuration of n triangles in space which `depthSort` splits into $\Omega(n^2)$ pieces.
6. In the program `depthSort`, the function call `mayObscure(p,q)` returns `TRUE` if it is possible for triangle `p` to obscure triangle `q`. What is the effect on `depthSort` of making `mayObscure` stronger, such that it returns `TRUE` if *and only if* `p` obscures `q`? What are the advantages and disadvantages of making `mayObscure` stronger?
7. In the program `depthSort`, note that the second call to function `mayObscure` is inefficient, since tests 1, 2, and 5 are repeated unnecessarily. Rewrite the program to remove this inefficiency.
8. Consider this claim concerning the algorithm for finding the intersection polygon of two convex polygons P and Q: If the boundaries of P and Q intersect, then the algorithm finds *all* their intersection points in no more than $2(|P| + |Q|)$ iterations. Either prove this claim and modify program `convexPolygonIntersect` accordingly, or disprove the claim by giving a counterexample.
9. Characterize the inputs for which program `convexPolygonIntersect` fails, in terms of `EPSILON2`.
10. Show how the correctness proof for `convexPolygonIntersect` uses the assumption that input polygons P and Q are convex.
11. Show that if no four points of point set S are cocircular ($|S| \geq 3$), then the Delaunay triangulation of S is unique.
12. Show that any triangulation of a finite point set S contains $3|S| - 3 - h$ edges, where the boundary of $\mathcal{CH}(S)$ contains h edges. [From this it follows that a triangulation contains $O(|S|)$ edges, a fact used in our proof that the Delaunay triangulation algorithm runs in $O(n^2)$ time.]
13. Show that, over all triangulations of finite point set S, the Delaunay triangulation maximizes the minimum measure of the internal angles.

https://hemanthrajhemu.github.io