Computing Common Tangents Without a Separating Line *

David Kirkpatrick Jack Snoeyink
Department of Computer Science
University of British Columbia

Abstract

Given two disjoint convex polygons in standard representations, one can compute outer common tangents in logarithmic time without first obtaining a separating line. If the polygons are not disjoint, there is an additional factor of the logarithm of the intersection or union size, whichever is smaller.

1 Introduction

In this paper, we revisit an old problem: computing a tangent line common to two disjoint polygons, P and Q, that are represented by ordered lists of n vertices stored in arrays or balanced binary trees.

Because a tangent to a polygon P through a given point q can be found in $\Theta(\log n)$ time by binary search, there is an easy $O(\log^2 n)$ time algorithm for finding a tangent common to P and Q that uses nested binary search. Overmars and van Leeuwen [7], as part of a data structure for dynamic convex hulls, gave a logarithmic-time algorithm for the special case in which P and Q have a known vertical separating line. Because one can compute a separating line for disjoint polygons in logarithmic time—by finding the shortest segment joining them [3] or using hierarchical representations [2, 6]—the Overmars/van Leeuwen algorithm gives a complete solution.

For three reasons, however, it is still interesting to ask whether a common tangent can be computed without the knowledge of the separator. First, there are common tangent problems (and intersection problems, which are their duals) that cannot be solved by one level of binary search. Guibas et al. [5] have shown that to compute an outer common tangent to intersecting polygons P and Q requires $\Omega(\log^2 n)$ time, even if points in $P - Q$ and $Q - P$ are given. Second, Overmars and van Leeuwen's data structure has been adapted for other purposes that do not have a vertical bias—including implicit storage of arrangements [4, 5], ray shooting [1], etc.—so that an affirmative answer simplifies and speeds up these applications by a constant factor. Third, it is natural to look for common tangents in situations where no separating line exists.

In the next section, we show that tangents for disjoint convex polygons can be computed in logarithmic time by using a tentative prune-and-search technique [6]. C code is given in an appendix. The approach is much like Overmars and van Leeuwen's [7]—starting with lists of vertices for P and for Q that are known to contain the tangent vertices, attempt to discard half of some list by doing a constant-time local test. Without a separating line, however, some tests do not give sufficient information. One can proceed by making tentative discards that are later

*Both authors supported in part by NSERC Research Grants. The second was also supported by a fellowship from the B.C. Advanced Systems Institute.

certified or revoked; the analysis uses a potential function to show that the amount of work done is still logarithmic. We also extend our approach to the case of intersecting polygons.

2 The algorithm

Our algorithm `Tang`(P, Q) takes as input two disjoint convex polygons whose vertices are stored in arrays in counter-clockwise (ccw) order. It finds vertices $p_i \in P$ and $q_j \in Q$ such that no vertex of P or Q lies to the right of the oriented line $\overrightarrow{p_i q_j}$. In case of degeneracy, p_i is chosen as the furthest such cw and q_j as the furthest ccw. Thus, `Tang`(P, Q) produces an outer common tangent that leaves P ccw and goes to Q. The call `Tang`(Q, P) produces the other outer common tangent.

We describe state variables and the invariants that the algorithm maintains. Then we initialize the variables and show how the `Refine`() procedure preserves the invariants while refining intervals that contain the common tangent vertices.

2.1 State information and invariants

The algorithm maintains several pieces of state information for each polygon. For P, we store the vertices p_0 to p_{n-1} in ccw order, and their number $P.n = n$ so that all access can be performed modulo $P.n$. For each vertex $p_k \in P$, we choose a canonical tangent τ_k to be the oriented tangent line at p_k that is furthest ccw: $\tau_k = \overrightarrow{p_k p_{k+1}}$. (We can use the orientation to speak about the right and left sides of τ_k and to order points along τ_k.) We also store three indices, $0 \le P.st \le P.tent < P.end \le 2P.n$, that satisfy two invariants below. Finally, we store a boolean variable $P.wrap$ that is defined in section 2.3.

For Q, the vertices are also stored in ccw order, but the tangent σ_k is chosen furthest cw: $\sigma_k = \overrightarrow{q_{k-1} q_k}$. The indices for Q have their order reversed, $0 \le Q.end < Q.tent \le Q.st \le 2Q.n$.

We would like to break P's circular list of vertices into a linear list on which we can perform binary search. No assumptions are made about the location of p_0; if one knew that p_0 would be inside the convex hull of $P \cup Q$, then this would be trivial.

Define interval I_P to be the indices of vertices of P that lie on the convex hull of $P \cup \{q_0\}$. As in figure 1, if no point of P is right of the oriented line $\overrightarrow{q_0 p_m}$ for $0 < m \le P.n$ and no point of P is right of $\overrightarrow{p_{m'} q_0}$ for $m \le m' < m + P.n$, then $I_P = [m, m']$. Notice that as index l runs over the interval $[m, m']$, we may encounter tangents τ_l that intersect Q before p_l, then tangents τ_l that do not intersect Q, and then tangents τ_l that intersect Q (equivalently, that intersect $\overline{q_0 q_j}$) after p_l. (The final tangent $\tau_{m'}$ should be limited so that q_0 does not appear to its right.) Define the interval I_Q similarly to contain indices of Q's vertices on the convex hull of $Q \cup \{p_0\}$. We never explicitly compute I_P or I_Q but we use them in the invariants.

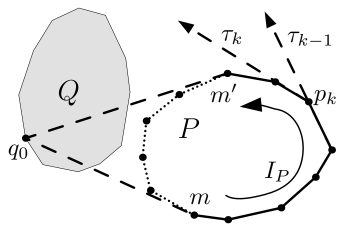

Figure 1: Defining, but not computing, I_P

Let $\overrightarrow{p_i q_j}$ be the common tangent that `Tang`(P, Q) seeks—that is, no point of P or Q is right of $\overrightarrow{p_i q_j}$. The invariants for P are:

1. The desired tangent index i is in the interval $(P.st, P.end] \cap I_P$.
2. If $P.tent \ne P.st$ then $P.tent \in (P.st, P.end] \cap I_P$ and points $q_{Q.tent}$ and $q_{Q.end}$ are left of tangent $\tau_{P.tent}$.

The invariants for Q are essentially the same:

1. The desired tangent index j is in the interval $(Q.st, Q.end] \cap I_Q$.
2. If $Q.tent \neq Q.st$ then $Q.tent \in (Q.st, Q.end] \cap I_Q$ and points $p_{P.tent}$ and $p_{P.end}$ are left of tangent $\sigma_{Q.tent}$.

2.2 Tentative prune and search

Using the invariants of the previous section we can outline the tentative prune-and-search method to compute the common tangent and prove it correct.

If $tent \neq st$ for a polygon, then we say that that polygon is *tentatively refined* or simply *refined*. We say that indices in $(tent, end]$ are *remaining* and those in $(st, tent]$, if any, have been *tentatively* discarded. If both polygons are tentatively refined and the invariants hold, then the index of the common tangent on at least one of the polygons is among those remaining. In other words, at most one tentative discard can be mistaken.

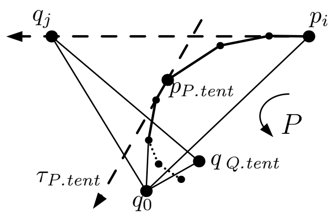

Figure 2: At most one mistaken tent. discard

Lemma 1 *Suppose that the invariants hold for disjoint polygons P and Q. If $\overrightarrow{p_i q_j}$ is the desired common tangent, then either $i \in (P.tent, P.end]$ or $j \in (Q.tent, Q.end]$.*

Proof: If one of the polygons is not refined, then $tent = st$ and the lemma follows from invariant 1. Therefore, suppose that both polygons are refined. Further suppose, for the sake of deriving a contradiction, that both tentative discards are mistaken: that is, $i \notin (P.tent, P.end]$ and $j \notin (Q.tent, Q.end]$. We make three observations.

First, $p_{P.tent}$ is in the triangle $\triangle p_i q_j q_0$: Since $P.tent \in [i, P.end] \cap I_P$, the point $p_{P.tent}$ lies on a convex curve from p_i to q_0. This curve cannot cross the segment $\overline{q_j q_0}$ because P and Q are disjoint, neither can it cross the common tangent $\overline{p_i q_j}$.

Second, $p_{P.tent}$ is in the triangle $\triangle p_i q_j q_{Q.tent}$: By invariant 2, q_0 and $q_{Q.tent}$ are both left of the tangent $\tau_{P.tent}$. By disjointness, $p_{P.tent}$ is not in $\triangle q_j q_0 q_{Q.tent}$. By the first observation, and the fact that $q_{Q.tent}$ is left of the common tangent, we know that $p_{P.tent}$ is in $\triangle p_i q_j q_{Q.tent}$.

Third, $q_{Q.tent}$ is not in triangle $\triangle p_i q_j p_{P.tent}$: This is immediate from the second observation.

The situations of P and Q are completely symmetric, however. We can derive observations that assert that $q_{P.tent}$ is in $\triangle p_i q_j p_{P.tent}$ by interchanging the roles of P and Q. This contradiction establishes the theorem. ∎

As a corollary, when only one candidate is remaining on each polygon, then we have found the common tangent.

Corollary 2 *If the invariants hold and intervals $(Q.tent, Q.end]$ and $(P.tent, P.end]$ contain one candidate each, then $q_{Q.end}$ and $p_{P.end}$ are the points of tangency desired.*

Proof: By lemma 1, we know that one of the intervals is correct: Say that $q_{Q.end}$ is one point of tangency. If P is not refined, then invariant 1 says that $p_{P.end}$ is the other. If P is refined, then invariant 2 says that $q_{Q.end}$ is left of $\tau_{P.tent}$, so the point of tangency must be after (ccw of) $p_{P.tent}$. Invariant 1 says that $p_{P.end}$ is the only candidate. ∎

Our algorithm discards indices from initial lists containing $O(n)$ indices using *refinement operations* A–C. We shall see in section 2.4 that `Refine`(P, Q) implements these refinement operations.

A. The interval $(P.tent, P.end]$ is halved by setting $P.tent$ or $P.end$ or both $P.tent$ and $P.st$ to be the midpoint of $(P.tent, P.end]$.
B. Possibly Q is *certified*—made unrefined by setting $Q.st = Q.tent$.

C. Possibly a mistake is found on Q. Then we revoke the tentative discard by $Q.end = Q.tent$, $Q.tent = Q.st$, and certify P by $P.st = P.tent$, because lemma 1 implies that the discards to P were correct.

The call `Refine`(Q, P) will handle the intervals for Q in a similar manner. We can perform a refine unless the $(tent, end]$ intervals on both polygons contain only a single index. If we alternately call `Refine`(P, Q) and `Refine`(Q, P), then we find the common tangent in logarithmic time.

Lemma 3 *If* `Refine`() *implements the refinement operations A–C, then* `Tang`(P, Q) *terminates after* $O(\log|P| + \log|Q|)$ *steps.*

Proof: We can define a potential for a polygon in terms of its indices *st*, *tent*, and *end*:

$$\Phi(P) = 2\log|P.end - P.st| + 2\log|P.end - P.tent| + (P.tent \neq P.st).$$

All logarithms are base 2 and the expression $(P.tent \neq P.st)$ equals 1 if the boolean test is true and 0 otherwise. The total potential is $\Phi = \Phi(P) + \Phi(Q)$.

To make analysis easier, we simplify the algorithm in a way that can only make the running time worse. We call `Refine`() on any unrefined polygon until both polygons are refined. Thus, an "unsuccessful" refine decreases Φ by 4; a successful one decreases Φ by 1. Then we call `Refine`(P, Q) and `Refine`(Q, P) alternately and either certify all tentative discards on one polygon and revoke those on the other or else extend the tentative discard (as if the index changes were always $P.tent$). Extending the tentative discard decreases Φ by 2. Certifying P after i refine steps decreases $\Phi(P)$ by $2i + 1$ and revoking Q after j steps increases $\Phi(Q)$ by at most $2j - 1$. Because of the alternation, $j \leq i + 1$ so the net change in Φ is at most zero. Note that certification can happen only after two successful refines, so every three steps Φ decreases by at least 2.

Since the initial potential $\Phi = O(\log P.n + \log Q.n)$ and Φ cannot be negative, the lemma is established. ∎

2.3 Initialization

To initialize P, if q_0 is not left of tangent τ_0, then we know that the interior of $\overline{p_0p_1}$ is inside the convex hull of $P \cup \{q_0\}$. We break P at p_0 by setting $st = tent = 0$, $end = n$, and $wrap = F$. Otherwise, we start at p_0 and wrap around P twice, as illustrated in figure 3, by setting $st = 0$, $tent = n$, $end = 2n$, and $wrap = T$.

We initialize Q in a similar manner: if p_0 is not left of tangent σ_0, set $st = tent = n$, $end = 0$, and $wrap = F$. Otherwise, we start at q_0 and wrap around Q twice by setting $st = 2n$, $tent = n$, $end = 0$, and $wrap = T$.

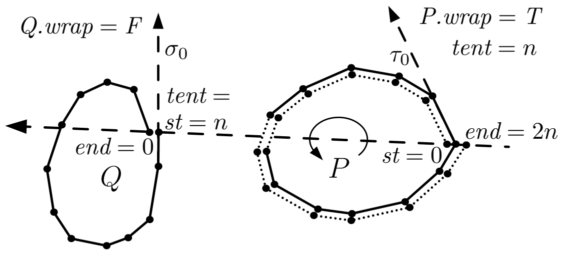

Figure 3: Initializing P, with $P.wrap = T$, and Q, with $Q.wrap = F$.

Lemma 4 *Initially, the two invariants hold for P and Q.*

Proof: We prove this for P. If $P.wrap$ is false, then $I_P \subset (0, P.n]$ and invariants 1 and 2 are trivial.

If $P.wrap$ is true, then the base segment $\overline{q_0p_0}$ intersects some edge $\overline{p_kp_{k+1}}$ of P where q_0 is not left of τ_k and $0 \leq k < P.n$. The index of the common tangent vertex p_i can be chosen from

$I_P \subset (k, k + P.n]$ to satisfy part of invariant 1. The remaining conditions of invariants 1 and 2 are trivial. ∎

2.4 Refining the intervals

Finally, we show that `Refine()` implements the refinement operations A–C listed in section 2.2.

Our most basic test determines whether a point (X, Y) is right or left of an oriented line $\overrightarrow{pq}$ by evaluating sign of the determinant

$$\begin{vmatrix} px & py & 1 \\ qx & qy & 1 \\ X & Y & 1 \end{vmatrix} = X(py - qy) - Y(px - qx) + (pxqy - qxpy).$$

Points to the left of $\overrightarrow{pq}$ make this determinant positive; those to the right make it negative. For a detailed treatment of signed homogeneous coordinates see Stolfi [8].

Suppose that there are candidates remaining on P: that $(P.tent, P.end]$ contains more than one index. Choose *mid* to be a median index in $(P.tent, P.end]$.

We are going to consider making $P.end = mid$ or $P.tent = mid$. Thus, if Q is refined, we test if p_{mid} is left of $\sigma_{Q.tent}$ to preserve invariant 2 for Q. If the oriented tangent line $\sigma_{Q.tent}$ intersects $\overline{p_0 p_{mid}}$ after $q_{Q.tent}$, then the point of tangency cannot be ccw of $q_{Q.tent}$. We can certify the tentative discard to Q by $Q.st = Q.tent$. If $\sigma_{Q.tent}$ intersects $\overline{p_0 p_{mid}}$ before $q_{Q.tent}$, then the point of tangency cannot be cw of $q_{Q.tent}$. We can revoke tentative discards on Q by the assignments $Q.end = Q.tent$ and $Q.tent = Q.st$ and certify those on P by $P.st = P.tent$.

Next we check if $mid \in I_P$ as follows: If τ_{mid} intersects $\overline{q_0 p_0}$ after p_{mid} or if $P.wrap$ and $mid > P.n$ and p_{mid} is not right of the base line $\overline{q_0 p_0}$, then nothing ccw of p_{mid} is in I_P. We can set $P.end = mid$. If τ_{mid} intersects $\overline{q_0 p_0}$ before p_{mid} or if $P.wrap$ and $mid < P.n$ and p_{mid} is right of the base line $\overline{q_0 p_0}$, then nothing cw of p_{mid} is in I_P. We can set $P.st = mid$ and $P.tent = mid$.

In a similar way, if τ_{mid} intersects $\overline{q_0 q_{Q.end}}$ or $\overline{q_0 q_{Q.tent}}$ after p_{mid} then the point of tangency cannot be ccw of p_{mid}. We set $P.end = mid$. If τ_{mid} intersects $\overline{q_0 q_{Q.end}}$ or $\overline{q_0 q_{Q.tent}}$ before p_{mid} then we set $P.st = mid$ and $P.tent = mid$. Finally, if none of these actions occur, we set $P.tent = mid$. This preserves the invariants for P.

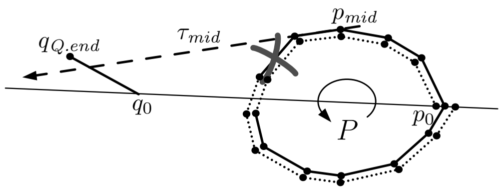

Figure 4: A situation causing $P.end = mid$

Therefore, `Refine()` implements the refinement operations needed for lemma 3. Since we can also initialize by lemma 4, we conclude with theorem 5.

Theorem 5 *The algorithm* `Tang`(P, Q) *computes a common tangent to disjoint convex polygons P and Q in $O(\log|P| + \log|Q|)$ time.*

3 Intersecting polygons

One can extend this analysis to intersecting polygons and obtain a theorem that covers the gap between the logarithmic-time algorithm for disjoint polygons and the $\Omega(\log^2 n)$ worst-case bound for intersecting polygons. We consider the case where polygons P and Q have at most two common tangents and where *helper points* in their differences are given: $p_0 \in P \setminus Q$ and $q_0 \in Q \setminus P$. Notice that these helper points certify that there is a common tangent; without them it would take $\Omega(n)$ time to determine if one polygon contained the other.

Theorem 6 *Let P and Q be two convex polygons whose boundaries intersect at most twice and let $p_0 \in P \setminus Q$ and $q_0 \in Q \setminus P$. One can compute the common tangent from P to Q in $O(\log(|P| + |Q|) \log K)$ time, where $K = \min\{|P \cap Q|, |P \cup Q|\}$.*

Proof: We sketch the proof for $O(\log(|P| + |Q|) \log |P \cap Q|)$ and leave many details to the reader. Begin by using the helper points to define the intervals I_P and I_Q as before and, in addition, compute these intervals in logarithmic time. One can check whether the tangents to p_0 and q_0 that define these intervals are the desired common tangent. We assume that they are not.

Even if P and Q intersect, if some point of Q is found to the right of tangent τ_{mid}, the tangent to P at p_{mid}, then by tests similar to that in figure 4, we can discard a portion of P so as to preserve the common tangent. Only when the inspected points of Q are left of τ_{mid} and the inspected points of P are left of $\sigma_{mid'}$ does local information fail to eliminate half of one of the polygons. Then there are the two cases, depicted in figure 5a and 5b, to consider: either the segments $\overline{p_0 p_{mid}}$ and $\overline{q_0 q_{mid'}}$ are disjoint, or they intersect.

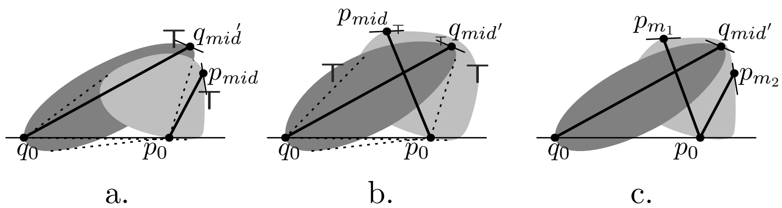

Figure 5: Two tentative discard cases and their combination

If they are disjoint, then lemma 1 implies that we can tentatively discard the indicated "outer" portions of P and Q. If they intersect, then we can prove that tentatively discarding the "inner" portions leads to at most one mistake. In either case, we can continue the computation in tentative mode until both "inner" and "outer" discards have been applied, as in figure 5c. In the illustrated case P is refined twice, at p_{m_1} and p_{m_2}, and all of Q has been tentatively discarded. There is a symmetric case in which Q is refined twice and all of P is discarded.

We can determine whether $q_{mid'}$ is inside or outside of P by searching between p_{m_1} and p_{m_2} for the edge of P that intersects the ray $\overrightarrow{q_0 q_{mid'}}$. Finding $q_{mid'}$ outside P allows the deletion of the portions of P and Q that are left of $\overrightarrow{q_0 q_{mid'}}$, because P is inside Q to the left of this ray. Finding $q_{mid'}$ inside P allows the deletion of the portions of P and Q that are right of $\overrightarrow{q_0 q_{mid'}}$, because cutting both polygons along $\overrightarrow{q_0 q_{mid'}}$ leaves a tangent from P to Q that is to the left of this ray. We can use similar analyses on the ray $\overrightarrow{p_0 p_{m_1}}$.

If edge $\overline{p_j p_{j+1}}$ is the edge of P that intersects $\overrightarrow{q_0 q_{mid'}}$ with $j \in [m_1, m_2)$, then we can find this edge in $O(\log(m_2 - j))$ steps by using an increasing-increment search from p_{m_2}—testing the 1st, 2nd, 4th, etc. vertex from p_{m_2} until a vertex passes the ray $\overrightarrow{q_0 q_{mid'}}$, then applying binary search. We can use a simultaneous increasing-increment searches from an end of I_Q clockwise towards $q_{mid'}$ for the edge of Q that intersects $\overrightarrow{p_0 p_{m_1}}$. When one of these searches succeeds, we delete portions of P and Q and escape this mode.

If p_{m_1} is found to be outside of Q or if $q_{mid'}$ is found outside of P, then the searches on Q or on P, respectively, walked only on portions on the boundary of $P \cap Q$. On the other hand, if the search on Q found that p_{m_1} was inside Q, then the unsuccessful search on P walked on $P \cap Q$.

Similarly, if the search on P found $q_{mid'}$ inside P, then the search on Q walked on $P \cap Q$. Thus, one of the two searches succeeds in $O(\log |P \cap Q|)$ steps. (For the lemma, we simultaneously search from p_{m_1} and q_0; one of these succeeds in $O(\log |P \cup Q|)$ steps.)

A potential function analysis similar to that of lemma 3 shows that we perform $O(\log(|P| + |Q|))$ steps, each costing $O(\log \min\{|P \cap Q|, |P \cup Q|\})$. This completes our sketch of the proof. ∎

References

[1] Bernard Chazelle and Leonidas J. Guibas. Visibility and intersection problems in plane geometry. *Discrete & Computational Geometry*, 4:551–581, 1989.

[2] David P. Dobkin and David G. Kirkpatrick. Determining the separation of preprocessed polyhedra: A unified approach. In *Seventeenth International Colloquium on Automata, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 400–413. Springer-Verlag, 1990.

[3] H. Edelsbrunner. Computing the extreme distances between two convex polygons. *Journal of Algorithms*, 6:213–224, 1985.

[4] Herbert Edelsbrunner, Leonidas Guibas, John Hershberger, Raimund Seidel, Micha Sharir, Jack Snoeyink, and Emo Welzl. Implicitly representing arrangements of lines or segments. *Discrete & Computational Geometry*, 4:433–466, 1989.

[5] Leo Guibas, John Hershberger, and Jack Snoeyink. Compact interval trees: A data structure for convex hulls. *International Journal of Computational Geometry & Applications*, 1(1):1–22, 1991.

[6] David Kirkpatrick and Jack Snoeyink. Tentative prune-and-search for computing Voronoi vertices. In *Proceedings of the Ninth Annual ACM Symposium on Computational Geometry*, pages 133–142, 1993.

[7] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[8] Jorge Stolfi. *Oriented projective geometry: A framework for geometric computations.* Academic Press, 1991.

Appendix A: C code for computing a common tangent to disjoint polygons

This code implements the common tangent algorithm described above.

```
/* main.h   Jack Snoeyink   March 94
 * tangent without a separating line
 */

#pragma once

#include <stdio.h>
#include <math.h>

#define MAXPTS 2000                       /* Maximum number of points per polyline */
#define EPSILON 1.0e-14                   /* Approximation of zero */

typedef double COORD;
typedef COORD POINT[2];                   /* Most data is Cartesian points */
typedef COORD HOMOG[3];                   /* Some partial calculations are homogeneous */
#define XX 0
#define YY 1
#define WW 2

typedef struct Polygon {
  int n,                                  /* Number of vertices in polygon */
  ccw,                                    /* 1 = ccw -1 = cw */
  st, end,                                /* Tangent is in (st, end] */
  tent,                                   /* Index of tentative refinement if tent ≠ st */
  wrap;                                   /* Boolean indicates wraparound */
  HOMOG tang;                             /* Tangent τ_tent when refined (tent ≠ st) */
  POINT v[MAXPTS];
} Polygon;

#define DET2(p, q) DET2x2(p,q, XX,YY)     /* Determinants */
#define DET2x2(p, q, i, j) ((p)[i]*(q)[j] - (p)[j]*(q)[i])
#define DET3C(p, q, r) DET2(q,r) - DET2(p,r) + DET2(p,q)

#define DOTPROD_2CH(p, q)                 /* 2-d Cartesian to Homogeneous dot product */
  ((q)[WW] + (p)[XX]*(q)[XX] + (p)[YY]*(q)[YY])

#define CROSSPROD_2SCCH(s, p, q, r)       /* 2-d Cart to Homog cross prod w/ sign */
  (r)[XX] = s * (- (q)[YY] + (p)[YY]);\
  (r)[YY] = s * (  (q)[XX] - (p)[XX]);
  (r)[WW] = s * ((p)[XX] * (q)[YY] - (p)[YY] * (q)[XX]);\

#define ASSIGN_H(p, op, q)                /* Homogeneous assignment */
  (p)[WW] op (q)[WW];  (p)[XX] op (q)[XX];  (p)[YY] op (q)[YY];

#define LEFT(x) (x > EPSILON)             /* Sidedness tests */
#define RIGHT(x) (x < -EPSILON)
#define LEFT_PL(p, l) LEFT(DOTPROD_2CH(p, l))
#define RIGHT_PL(p, l) RIGHT(DOTPROD_2CH(p, l))
#define LEFT_PPP(p, q, r) LEFT(DET3C(p, q, r))
#define RIGHT_PPP(p, q, r) RIGHT(DET3C(p, q, r))
```

```
/* nosep.c   Jack Snoeyink  March 94   Common tangent without a separating line
 */
#include "main.h"

#define Pv(m) P->v[(m) % P->n]                   /* Indexing into polygon vertices mod n */
#define Qv(m) Q->v[(m) % Q->n]

#define CCW(x) (x->ccw == 1)                     /* Is x oriented counterclockwise? */
#define DONE(x) ((x->end - x->tent) == x->ccw)   /* Any candidates left? */
#define REFINED(x) (x->st != x->tent)            /* Is x refined? */

#define DISC_START  0                            /* Actions in Refine() */
#define DISC_END    1
#define NO_DISC     2

void Refine(P, Q)
     Polygon *P, *Q;          /* We refine polygon P checking against Q. We
{                                can assume that more than one candidate exists in
  HOMOG q0pm, mtang;             (P.tent, P.end] and the invariants hold. */
  register int mid, left_base, action = NO_DISC;
  register COORD *pm, *pm1, *qend, *qt;

  mid = P->tent + (P->end - P->tent) / 2;       /* Check mid point. Round towards P.tent */
  pm = Pv(mid); pm1 = Pv(mid + P->ccw);
  CROSSPROD_2SCCH(P->ccw, pm, pm1, mtang);      /* Generate τ_mid */
  CROSSPROD_2SCCH(1, Qv(0), pm, q0pm);
  left_base = RIGHT_PL(Pv(0), q0pm);

  if (REFINED(Q) && !LEFT_PL(pm, Q->tang)) {
    qt = Qv(Q->tent);
    if (CCW(Q) ^ LEFT_PPP(Pv(0), qt, pm))       /* Check σ_Q.tent */
      Q->st = Q->tent;                          /* Certify tentative to Q */
    else {
      Q->end = Q->tent;
      Q->tent = Q->st;                          /* Revoke tentative to Q */
      P->st = P->tent;                          /* Certify tentatve on P (if refined) */
    }
  }

  qend = Qv(Q->end);  qt = Qv(Q->tent);

  if (P->wrap && (left_base ^ (mid > P->n)))    /* Is P wrapped around? */
    action = !left_base;
  else if (!LEFT_PL(Qv(0), mtang))              /* Can we be tangent w.r.t q_0? */
    action = left_base;
  else if (!LEFT_PL(qend, mtang))               /* Can we be tangent w.r.t q_Q.end? */
    action = LEFT_PL(qend, q0pm);
  else if (REFINED(Q) && !LEFT_PL(qt, mtang))   /* Can we be tangent w.r.t q_Q.tent? */
    action = LEFT_PL(qt, q0pm);

  if (action == NO_DISC)                        /* We tentatively refine at mid */
    { P->tent = mid; ASSIGN_H(P->tang, =, mtang) }
  else if (CCW(P) ^ action) P->st = P->tent = mid;   /* A discard at P.st occurred */
  else P->end = mid;                            /* A discard at P.end occurred */
}
```

```
void Tang(P, Q)
     Polygon *P, *Q;
{                                                  /* Compute a tangent from P to Q            */
  register int n1 = Q->n - 1;

  P->ccw = 1; P->st = P->tent = 0; P->end = P->n;  /* Initialize P                             */
  CROSSPROD_2SCCH(1, Pv(0), Pv(1), P->tang);
  if (P->wrap = LEFT_PL(Qv(0), P->tang))           /* Wrap P initially                         */
    { P->tent = P->n; P->end += P->n; }

  Q->ccw = -1; Q->st = Q->tent = Q->n; Q->end = 0; /* Initialize Q                             */
  CROSSPROD_2SCCH(1, Qv(n1), Qv(0), Q->tang);
  if (Q->wrap = LEFT_PL(Pv(0), Q->tang))           /* Wrap Q initially                         */
    Q->st += Q->n;

  while (!DONE(P) || !DONE(Q)) {
    if (!DONE(P)) Refine(P, Q);
    if (!DONE(Q)) Refine(Q, P);
  }                                                /* Finished. Q.end and P.end indicate tangent */
}
```