◊ I.2

Testing the Convexity of a Polygon

Peter Schorn
Institut für Theoretische Informatik
ETH, CH-8092 Zürich, Switzerland
schorn@inf.ethz.ch

Frederick Fisher
2630 Walsh Avenue
Kubota Pacific Computer, Inc.
Santa Clara, CA
fred@kpc.com

◊ Abstract ◊

This article presents an algorithm that determines whether a polygon given by the sequence of its vertices is convex. The algorithm is implemented in C, runs in time proportional to the number of vertices, needs constant storage space, and handles all degenerate cases, including non-simple (self-intersecting) polygons.

Results of a polygon convexity test are useful to select between various algorithms that perform a given operation on a polygon. For example, polygon classification could be used to choose between point-in-polygon algorithms in a ray tracer, to choose an output rasterization routine, or to select an algorithm for line-polygon clipping or polygon-polygon clipping. Generally, an algorithm that can assume a specific polygon shape can be optimized to run much faster than a general routine.

Another application would be to use this classification scheme as part of a filter program that processes input data, such as from a tablet. Results of the filter could eliminate complex polygons so that following routines may assume convex polygons.

◊ Issues in Solving the Problem ◊

The problem whose solution this article describes started out as a posting on the USENET bulletin board 'comp.graphics' which asked for a program that could decide whether a polygon is convex. Answering this question turned into a contest, managed by Kenneth Sloan, which aimed at the construction of a correct and efficient program. The most important issues discussed were:

- Correctness, especially in degenerate cases. Many people quickly succeeded in writing a program which could handle *almost* all cases. The challenge was a program which works in all, even degenerate, cases. Some degenerate examples are depicted in Figure 1.

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

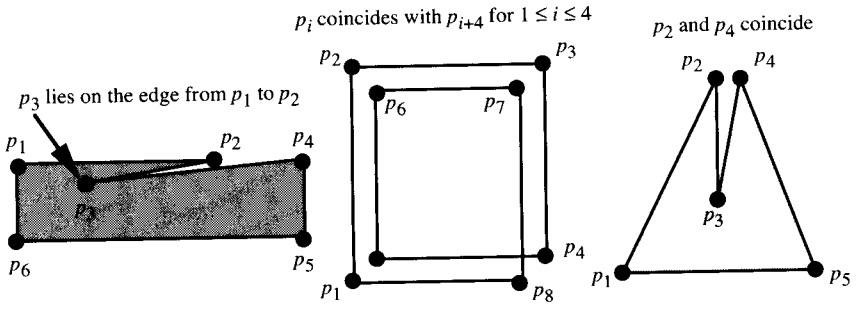

Figure 1. Some degenerate cases.

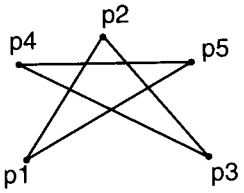

Figure 2. Non-convex polygon with a right turn at each vertex.

Although the first two examples might be considered convex (their interior is indeed convex), a majority of the participants in the discussion agreed that these cases should be considered not convex. Further complications are "all points collinear" and "repeated points."

- What is a convex polygon? This question is very much related to correctness and a suitable definition of a convex polygon was a hotly debated topic. When one thinks about the problem for the first time, a common mistake is to require a right turn at each vertex and nothing else. This leads to the counterexample in Figure 2.
- Efficiency. The program should run in time proportional to the number of vertices. Furthermore, only constant space for the program was allowed. This required a solution to read the polygon vertices from an input stream without saving them.
- Imprecise arithmetic. The meaning of "three points are collinear" becomes unclear when the coordinates of the points are only approximately correct or when floating-point arithmetic is used to test for collinearity or right turns. This article assumes exact arithmetic in order to avoid complications.

◇ What Is a Convex Polygon? ◇

Answering this question is an essential step toward the construction of a robust program. There are at least four approaches:

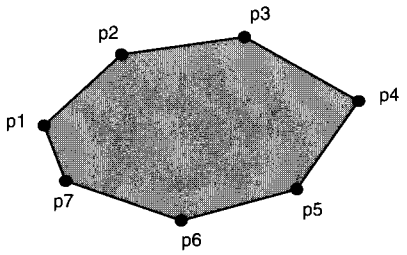

Figure 3. An undisputed convex polygon.

- The cavalier attitude: I know what a convex polygon is when I see one. For example the polygon in Figure 3 is clearly convex.
- The "what works for me" approach: A polygon P is convex if my triangulation routine (renderer, etc.) which expects convex polygons as input can handle P.
- The "algorithm as definition" approach: A polygon is convex if my convexity testing program declares it as such.
- A more abstract, mathematical approach starting with the definition of a convex set: A set S of points is convex $\Leftrightarrow$

$$(p \in S) \wedge (q \in S) \Rightarrow \forall \lambda : 0 \leq \lambda \leq 1 : \lambda \cdot p + (1 - \lambda) \cdot q \in S$$

  This roughly means that a set of points S is convex iff for any line drawn between two points in the set S, then all points on the line segment are also in the set.

In the following we propose a different, formal approach, which has the following advantages:

- It captures the intuition about a convex polygon.
- It gives a reasonable answer in degenerate cases.
- It distinguishes between clockwise- and counterclockwise orientations.
- It leads to a correct and efficient algorithm.

Classification: Given a sequence $P = p_1, p_2, \ldots, p_n$ of points in the plane such that

1. n is an integer and $(n > 0)$.
2. Consecutive vertices are different. $p_i \neq p_{i+1}$ for $1 \leq i \leq n$ (we assume $p_{n+1} = p_1$).
3. We restrict consideration to sequences where p_1 is lexicographically the smallest, i.e., $p_1 < p_i$ for $2 \leq i \leq n$ where $p < q \Leftrightarrow (p_x < q_x) \vee ((p_x = q_x) \wedge (p_y < q_y))$.
4. All convex polygons are monotone polygons, that is the x-coordinate of the points increases monotonically and then decreases monotonically. p_j is the "rightmost vertex."
   $\exists j : 1 < j \leq n : p_i < p_{i+1}$ for $1 \leq i < j$ and $p_{i+1} < p_i$ for $j \leq i \leq n$

Then if $p_i = [X_i, Y_i]$, and

$$d(i) = (X_{i-1} - X_i) \cdot (Y_i - Y_{i+1}) - (Y_{i-1} - Y_i) \cdot (X_i - X_{i+1})$$

P denotes a left- (counterclockwise) convex polygon $\Leftrightarrow$

$$(\forall i : 1 \leq i \leq n : d(i) \leq 0) \wedge (\exists i : 1 \leq i \leq n : d(i) < 0)$$

P denotes a right- (clockwise) convex polygon $\Leftrightarrow$

$$(\forall i : 1 \leq i \leq n : d(i) \geq 0) \wedge (\exists i : 1 \leq i \leq n : d(i) > 0)$$

P denotes a degenerate-convex polygon $\Leftrightarrow$

$$\forall i : 1 \leq i \leq n : d(i) = 0$$

P denotes a non-convex polygon $\Leftrightarrow$

$$(\exists i : 1 \leq i \leq n : d(i) < 0) \wedge (\exists i : 1 \leq i \leq n : d(i) > 0)$$

This classification of vertex-sequences agrees with our intuition for convex polygons (see Figure 3). For clockwise convex polygons there is a right turn at each vertex, and for counterclockwise convex polygons there is a left turn at each vertex. If the points satisfy condition 4 but lie on a line, the polygon is classified as degenerate-convex.

For purposes of simplifying the classification, conditions 2, 3, and 4 constrain the possible polygons. However, the classification can be extended to sequences not satisfying conditions 2, 3, or 4. Any sequence can easily meet conditions 2 and 3 if we remove consecutive duplicate points and perform a cyclic shift, moving the lexicographically smallest point to the beginning of the sequence. If condition 4 cannot be met, the sequence denotes a non-convex polygon.

♢ Implementation in C ♢

The following C program shows how the classification scheme can be turned into a correct and efficient implementation. The program accepts lines which contain two numbers, denoting the x- and y-coordinates of a point (see the function `GetPoint`). Duplicate points are removed on the fly (see the function `GetDifferentPoint`).

Since we do not want to store more than a constant number of points, we cannot perform a cyclic shift of the input vertices in order to assure condition 3. Instead, the program counts how often the lexicographic order of the input vertices changes. If this number exceeds two, the input polygon is definitely not convex.

In addition to the four cases distinguished in the classification scheme, the program introduces a fifth case (`NotConvexDegenerate`) for polygons whose vertices all lie on a line but do not satisfy condition 4.

◇ Program to Classify a Polygon's Shape ◇

```
#include <stdio.h>

typedef enum { NotConvex, NotConvexDegenerate,
               ConvexDegenerate, ConvexCCW, ConvexCW } PolygonClass;

typedef struct { double x, y; } Point2d;

int WhichSide(p, q, r)          /* Given a directed line pq, determine  */
Point2d       p, q, r;          /* whether qr turns CW or CCW.          */
{
    double result;
    result = (p.x - q.x) * (q.y - r.y) - (p.y - q.y) * (q.x - r.x);
    if (result < 0) return -1;  /* q lies to the left  (qr turns CW).   */
    if (result > 0) return  1;  /* q lies to the right (qr turns CCW).  */
    return 0;                   /* q lies on the line from p to r.      */
}

int Compare(p, q)               /* Lexicographic comparison of p and q  */
Point2d     p, q;
{
    if (p.x < q.x) return -1;   /* p is less than q.                    */
    if (p.x > q.x) return  1;   /* p is greater than q.                 */
    if (p.y < q.y) return -1;   /* p is less than q.                    */
    if (p.y > q.y) return  1;   /* p is greater than q.                 */
    return 0;                   /* p is equal to q.                     */
}

int GetPoint(f, p)              /* Read p's x- and y-coordinates from f */
FILE    *f;                     /* and return true, iff successful.     */
Point2d *p;
{
    return !feof(f) && (2 == fscanf(f, "%lf%lf", &(p->x), &(p->y)));
}

int GetDifferentPoint(f, previous, next)
FILE    *f;                     /* Read next point into 'next' until it */
Point2d previous, *next;        /* is different from 'previous' and     */
{                               /* return true iff successful.          */
    int eof;
    while((eof = GetPoint(f, next)) && (Compare(previous, *next) == 0));
    return eof;
}

/* CheckTriple tests three consecutive points for change of direction
 * and for orientation.
 */
#define CheckTriple                                                     \
        if ( (thisDir = Compare(second, third)) == -curDir )            \
            ++dirChanges;                                               \
        curDir = thisDir;                                               \
        if ( thisSign = WhichSide(first, second, third) ) {             \
```

```
            if ( angleSign == -thisSign )                                   \
                return NotConvex;                                           \
            angleSign = thisSign;                                           \
        }                                                                   \
        first = second; second = third;

/* Classify the polygon vertices on file 'f' according to: 'NotConvex'   */
/* 'NotConvexDegenerate', 'ConvexDegenerate', 'ConvexCCW', 'ConvexCW'.   */
PolygonClass ClassifyPolygon(f)
FILE                        *f;
{
    int          curDir, thisDir, thisSign, angleSign = 0, dirChanges = 0;
    PolygonClass result;
    Point2d      first, second, third, saveFirst, saveSecond;

    if ( !GetPoint(f, &first) || !GetDifferentPoint(f, first, &second) )
        return ConvexDegenerate;
    saveFirst = first;  saveSecond = second;
    curDir = Compare(first, second);
    while( GetDifferentPoint(f, second, &third) ) {
        CheckTriple;
    }
    /* Must check that end of list continues back to start properly. */
    if ( Compare(second, saveFirst) ) {
        third = saveFirst; CheckTriple;
    }
    third = saveSecond;  CheckTriple;

    if ( dirChanges > 2 ) return angleSign ? NotConvex : NotConvexDegenerate;
    if ( angleSign  > 0 ) return ConvexCCW;
    if ( angleSign  < 0 ) return ConvexCW;
    return ConvexDegenerate;
}

int main()
{
    switch ( ClassifyPolygon(stdin) ) {
        case NotConvex:           fprintf( stderr,"Not Convex\n");
                                  exit(-1); break;
        case NotConvexDegenerate: fprintf( stderr,"Not Convex Degenerate\n");
                                  exit(-1); break;
        case ConvexDegenerate:    fprintf( stderr,"Convex Degenerate\n");
                                  exit( 0); break;
        case ConvexCCW:           fprintf( stderr,"Convex Counter-Clockwise\n");
                                  exit( 0); break;
        case ConvexCW:            fprintf( stderr,"Convex Clockwise\n");
                                  exit( 0); break;
    }
}
```

♢ Optimizations ♢

The previous code was chosen for its conciseness and readability. Other versions of the code were written which accept a vertex count and pointer to an array of vertices. Given this interface, it is possible to obtain good performance measurements by timing a large number of calls to the polygon classification routine.

Variations of the code presented have resulted in a two to four times performance increase, depending on the polygon shape. Optimizations for a particular machine or programming language will undoubtedly produce different results. Some considerations are:

- Convert each of the routines to macro definitions.
- Instead of keeping track of the first, second, and third points, keep track of the previous delta (second − first), and a current delta (third − second). This will speed up parts of the algorithm: The macro `Compare` needs only compare two numbers with zero, instead of four numbers with each other; the routine for getting a different point calculates the delta as it determines if the new point is different; the cross product calculation uses the deltas directly instead of subtracting vertices each time; the comparison for the `WhichSide` routine may be moved up to the `CheckTriple` routine to save a comparison at the expense of a little more code; and preparing to examine the next point requires three moves instead of four.
- Checking for less than three vertices is possible, but generally slows down the other cases.
- Every time the variable `dirChanges` is incremented, it would be possible to check if the number is now greater than two. This will slow down the convex cases, but makes it possible to exit early for polygons which violate classification condition 4. If it is important to distinguish between `NotConvex` and `NotConvexDegenerate`, this optimization may not be used.

♢ Reasonably Optimized Routine to Classify a Polygon's Shape ♢

```
/*

. . . code omitted which reads polygon, stores in an array, and calls
        classifyPolygon2()
*/

typedef double  Number;          /* float or double */

#define ConvexCompare(delta)                                          \
    ( (delta[0] > 0) ? -1 :      /* x coord diff, second pt > first pt */\
```

```
      (delta[0] < 0) ?  1 :      /* x coord diff, second pt < first pt */\
      (delta[1] > 0) ? -1 :      /* x coord same, second pt > first pt */\
      (delta[1] < 0) ?  1 :      /* x coord same, second pt > first pt */\
      0 )                        /* second pt equals first point */

#define ConvexGetPointDelta(delta, pprev, pcur )                          \
    /* Given a previous point 'pprev', read a new point into 'pcur' */    \
    /* and return delta in 'delta'.                                 */    \
    pcur = pVert[iread++];                                                \
    delta[0] = pcur[0] - pprev[0];                                        \
    delta[1] = pcur[1] - pprev[1];                                        \

#define ConvexCross(p, q) p[0] * q[1] - p[1] * q[0];

#define ConvexCheckTriple                                                 \
    if ( (thisDir = ConvexCompare(dcur)) == -curDir ) {                   \
          ++dirChanges;                                                   \
          /* The following line will optimize for polygons that are  */ \
          /* not convex because of classification condition 4,       */ \
          /* otherwise, this will only slow down the classification. */ \
          /* if ( dirChanges > 2 ) return NotConvex;                 */ \
    }                                                                     \
    curDir = thisDir;                                                     \
    cross = ConvexCross(dprev, dcur);                                     \
    if ( cross > 0 ) { if ( angleSign == -1 ) return NotConvex;           \
                       angleSign = 1;                                     \
                     }                                                    \
    else if (cross < 0) { if (angleSign == 1) return NotConvex;           \
                          angleSign = -1;                                 \
                        }                                                 \
    pSecond = pThird;            /* Remember ptr to current point. */     \
    dprev[0] = dcur[0];          /* Remember current delta.        */     \
    dprev[1] = dcur[1];                                                   \

classifyPolygon2( nvert, pVert )
int     nvert;
Number  pVert[][2];
/* Determine polygon type. return one of:
 *      NotConvex, NotConvexDegenerate,
 *      ConvexCCW, ConvexCW, ConvexDegenerate
 */
{
    int      curDir, thisDir, dirChanges = 0,
             angleSign = 0, iread, endOfData;
    Number   *pSecond, *pThird, *pSaveSecond, dprev[2], dcur[2], cross;

    /* if ( nvert <= 0 ) return error;       if you care */

    /* Get different point, return if less than 3 diff points. */
    if ( nvert < 3 ) return ConvexDegenerate;
    iread = 1;
    while ( 1 ) {
        ConvexGetPointDelta( dprev, pVert[0], pSecond );
```

```
        if ( dprev[0] || dprev[1] ) break;
        /* Check if out of points. Check here to avoid slowing down cases
         * without repeated points.
         */
        if ( iread >= nvert ) return ConvexDegenerate;
    }

    pSaveSecond = pSecond;

    curDir = ConvexCompare(dprev);      /* Find initial direction */

    while ( iread < nvert ) {
        /* Get different point, break if no more points */
        ConvexGetPointDelta(dcur, pSecond, pThird );
        if ( dcur[0] == 0.0  &&  dcur[1] == 0.0 ) continue;

        ConvexCheckTriple;              /* Check current three points */
    }

    /* Must check for direction changes from last vertex back to first */
    pThird = pVert[0];                  /* Prepare for 'ConvexCheckTriple' */
    dcur[0] = pThird[0] - pSecond[0];
    dcur[1] = pThird[1] - pSecond[1];
    if ( ConvexCompare(dcur) ) {
        ConvexCheckTriple;
    }

    /* and check for direction changes back to second vertex */
    dcur[0] = pSaveSecond[0] - pSecond[0];
    dcur[1] = pSaveSecond[1] - pSecond[1];
    ConvexCheckTriple;                  /* Don't care about 'pThird' now */

    /* Decide on polygon type given accumulated status */
    if ( dirChanges > 2 )
        return angleSign ? NotConvex : NotConvexDegenerate;

    if ( angleSign > 0 ) return ConvexCCW;
    if ( angleSign < 0 ) return ConvexCW;
    return ConvexDegenerate;
}
```

◇ Acknowledgments ◇

We are grateful to the participants of the electronic mail discussion: Gavin Bell, Wayne Boucher, Laurence James Edwards, Eric A. Haines, Paul Heckbert, Steve Hollasch, Torben Ægidius Mogensen, Joseph O'Rourke, Kenneth Sloan, Tom Wright, and Benjamin Zhu.