Constant Time Parallel Indexing of Points in a Triangle[*]

Simon Kahan and Pierre Kelsen

Abstract

Consider a triangle whose three vertices are grid points. Let k denote the number of grid points in the triangle. We describe an *indexing* of the triangle: a bijective mapping from $\{0, \ldots, k-1\}$ to the grid points in the triangle. Computing such a mapping is a fundamental subroutine in fine-grained parallel computation arising in graphics applications such as ray-tracing. We describe a very fast indexing algorithm: after a preprocessing phase requiring time proportional to the number of bits in the vertices of the triangle, a grid point in the triangle can be computed in constant time from its index. The method requires only constant space.

1 Introduction

Consider a triangle whose vertices are points on a two-dimensional grid. Let k be the number of grid points, or "pixels", that lie in the interior of the triangle or on its boundary. By an indexing of the points in a triangle we mean a labeling of the points in the triangle with the integers from 0 to $k-1$. Given an integer in this range an *indexing algorithm* determines the coordinates of the corresponding point in the triangle. The standard indexing algorithm used in computer graphics is the so-called *scan-conversion* ([2]). Figure 1(a) shows the order in which this routine encounters the pixels of a triangle as it is filled or drawn into a graphical frame buffer. The intended purpose of scan-conversion is to render a triangle, not to determine an indexing; but because it is so simple, scan-conversion is probably the fastest sequential algorithm for listing the indices of all points in a general triangle.

However, scan-conversion falls short as an indexing algorithm for two reasons. First, to index a particular point, say the kth, requires time at least linear in k because it entails scanning the previous $k-1$ points. Even if only one point is desired, the cost could be the same as that to scan the entire triangle! Fast indexing of single points is key to parallel applications such as ray-tracing on fine-grained machines like the Connection Machine [4] where each point contained in a triangle is to be assigned to a distinct processor. The available processors may assign themselves unique integer indices using parallel prefix or some other machine specific load distribution technique, and subsequently compute the corresponding points via an indexing algorithm.

Let k be the number of points in the triangle. If each of k processors uses scan-conversion, then the total parallel work is $\Omega(k^2)$. Even if one processor is dedicated to the distribution of points, and computes the scan-conversion indexing only once, the delay before the last processor receives its point is the time of scan-conversion, $\Omega(k)$. Ideally we would like to have an indexing that could be computed in constant time so that the total parallel work is $\Theta(k)$,

[*]The authors were supported by the Natural Sciences and Engineering Research Council of Canada. Address of the authors: Department of Computer Science, University of British Columbia, 6356 Agricultural Road, Vancouver, B.C. CANADA V6T 1Z2; e-mail: skahan@cs.ubc.ca, kelsen@cs.ubc.ca.

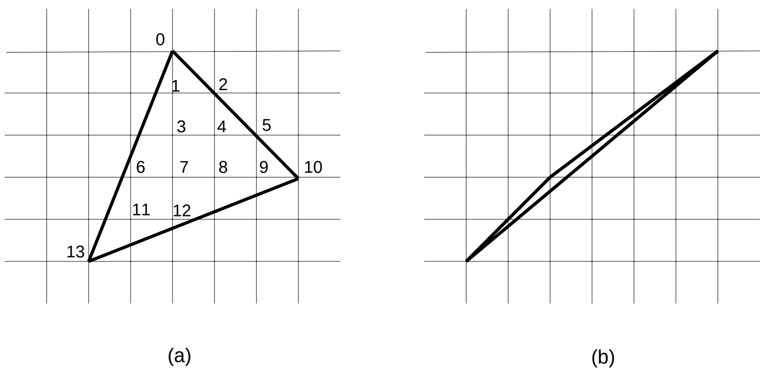

Figure 1: (a) Indexing of points by scan-conversion; (b) skinny triangle on which scan-conversion performs poorly.

and the maximum delay is constant. Unfortunately, the scan-conversion algorithm cannot be adapted to such a parallel implementation.

Another drawback of scan-conversion is that it performs poorly on certain skinny triangles such as the one shown in Figure 1(b). In fact, scan-conversion is extremely inefficient even as a sequential algorithm for filling such triangles! The reason is that scan-conversion's running time is proportional not just to the number of points in the triangle, but to the maximum of that quantity and the number of scan-lines intersected by the triangle. Therefore, a triangle that is very skinny and angled 45 degrees to the scan-lines results in a running time proportional to the length of its longest side, even when it contains few points. Consequently, scan-conversion is ill-suited to indexing — and to filling — these skinny triangles.

The algorithm presented in this paper has neither of the deficiencies of scan-conversion. It provides an indexing computable in constant time per point and is thus ideally suited to parallel applications. Also, it performs filling in time proportional to the number of points drawn instead of to the triangle dimensions, so it is well suited to filling skinny triangles. Furthermore, our algorithm is only slightly more complicated than scan-conversion; full pseudo-code is included in this paper. Our algorithm has some other advantages over scan-conversion: it can easily be inverted, i.e., given the coordinates of a point, we can compute its index in constant time. This is useful for compact storage of point sets. Also our method allows one to generate a point in the triangle uniformly at random in constant time. Note that it is easy enough to generate approximately uniform pseudo-random points having floating-point coordinates and then to round to the nearest integer coordinates ([3]). However, once rounded, the points are not generated uniformly at all.

The remainder of this paper is organized as follows. The next section introduces preliminary definitions and notation. In Section 3 we solve the indexing problem for a special subclass of triangles. In Section 4 we show that the ideas used in the special case can be modified to yield a fast and simple indexing algorithm for general triangles.

2 Preliminaries

The notation (ab), (abc) and $(abcd)$ is used to denote the line segment (ab) (including endpoints), the triangle with vertices a, b, c and the parallelogram with vertices a, b, c and d. The vertices of a triangle or parallelogram are assumed not to be colinear. These vertices are assumed to be grid points on a two dimensional grid. Throughout the remainder of this paper the term *point* refers to a grid point. We describe the position of points with respect to a fixed Cartesian coordinate system. Such a system is defined by taking a (grid) point p to be the origin and assigning coordinates $(1, 0)$ and $(0, 1)$ to the two neighboring (grid) points on the two grid lines that intersect at p.

We say that a point is *in* the triangle (abc) (in the parallelogram $(abcd)$) if it either lies in the interior of the triangle (parallelogram) or it lies on the boundary. Let k denote the number of points in triangle (abc). We want a fast algorithm for computing a bijective mapping from $\{0, \ldots, k-1\}$ to the (grid) points in the triangle. This algorithm has as input an integer in the range $0 \ldots k-1$ and returns the coordinates of the corresponding point in the triangle. We denote the coordinates of a point P by a pair (x_P, y_P).

We shall use a few facts from elementary number theory. We first note that for any two distinct points a and b the line segment (ab) contains $gcd(|x_b - x_a|, |y_b - y_a|) + 1$ evenly spaced points. The following result will also be needed ([5]):

Theorem 1 *Let u, v and w be integers with $uv \neq 0$. The linear diophantine equation $ux + vy = w$ has an integer solution (x, y) if and only if $gcd(|u|, |v|)$ divides w. If this is the case then all solutions of the equation are of the form $x = x_0 + \lambda v / gcd(|u|, |v|)$ and $y = y_0 - \lambda u / gcd(|u|, |v|)$ where λ is an arbitrary integer and (x_0, y_0) is a particular solution to the equation.* $\square$

We can compute a particular solution to the linear diophantine equation by modifying the Euclidean algorithm. We first observe that a particular solution can easily be expressed in terms of a solution to an equation of the form $ux + vy = gcd(u, v)$ where u and v are positive integers. The following procedure computes a solution (x, y) to this equation; we assume that x, y and r are global variables. An alternative nonrecursive formulation of this algorithm is given in [1, page 301].

```
$solve(u, v)\{$
if $v = 0$ then $\{x := 1; y := 0;\}$
else { $solve(v, u \bmod v)$;
        $r := x$;
        $x := y$;
        $y := r - y\lfloor u/v \rfloor;\}\}$
```

The algorithm terminates because the second argument decreases at each iteration. It produces the correct result (x, y) because $xv + y(u \bmod v) = g$ implies $yu + (x - y\lfloor u/v \rfloor)v = g$ for $v > 0$. The running time is linear in the number of bits in u and v (just as for the Euclidean algorithm).

3 A Special Case

We first study the indexing problem for right triangles that have two sides lying on grid lines. The reason for doing so is two-fold: first, we shall see that the indexing problem is nontrivial even for this simple case; second, the study of this class of triangles provides a convenient vehicle for introducing the main ideas used for solving the general case.

Without loss of generality the sides (ab) and (ac) lie on (orthogonal) grid lines, (bc) facing the right angle. For convenience we choose a as the origin of the coordinate system with the next grid points on (ab) and (ac) having coordinates $(1, 0)$ and $(0, 1)$. This will ensure that

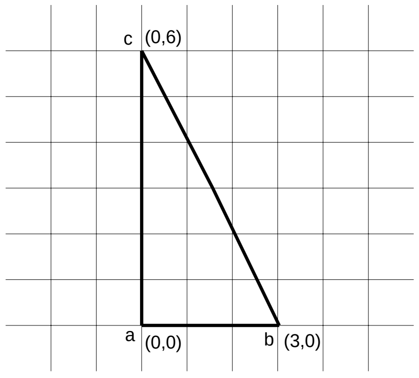

Figure 2: Triangle (abc) with sides (ab) and (ac) on grid lines.

all points in (abc) have nonnegative coordinates. Note that $y_b = x_c = 0$. An example of such a triangle is depicted in Figure 2.

We observe that any indexing of (abc) induces a linear ordering on the points in (abc) in the obvious way, i.e., a point precedes another point if it has a smaller index. We may thus rephrase our problem in terms of linear orders: we would like to determine a linear order on the points in (abc) such that a point can be computed quickly from its rank in the linear order.

A natural order on the points in (abc) is the lexicographic ordering of the points by their coordinates. Note that a and b are the first and last point in this ordering. To compute a point from its rank (index), we would need a formula for calculating the number of points P in the triangle with $x_P \leq i$. Routine calculation shows that this number is given by $\sum_{j=0}^{i} 1 + \lfloor \frac{y_c(x_b - j)}{x_b} \rfloor$. Unfortunately, there is no simple closed form for this expression (except for $i = x_b$). This fact makes the ordering unsuitable for fast indexing. The same applies to ordering the points by y coordinate first and then by x coordinate.

A new idea is needed at this point. Let d be the reflection of a about m, the midpoint of (bc) (which may not be a grid point). As we shall see shortly, it is helpful to study the problem of indexing the points in the rectangle $(abcd)$. Although the lexicographical ordering was not suitable for indexing points in (abc) it provides a very fast indexing of the points in rectangle $(abcd)$. Because the number of points P in $(abcd)$ with $x_p \leq i$ is equal to $(i+1)(y_c+1)$, the *ith* point in $(abcd)$ has coordinates: $x_p = \lfloor \frac{i}{y_c+1} \rfloor$ and $y_p = i - x_p(y_c+1)$.

The reflection about m naturally partitions the points in $(abcd)$ other than m into disjoint pairs of points that are reflections of each other. Let k denote the number of points in rectangle $(abcd)$. The lexicographic ordering of the k points has the following important property: the set R comprising the first $\lfloor k/2 \rfloor$ points in the ordering contains exactly one point of each pair. We say that a set of points in rectangle $(abcd)$ is *nice* if it has this property. Thus, R is a nice set of points. We now make two important observations. First, the indexing of the points in R using the method given in the last paragraph provides a simple indexing method for *any* nice set S: to generate the *ith* point in S, generate the *ith* point in $(abcd)$ and reflect this point about m if it lies outside S. Second, the set T of points in triangle (abc) that do not lie on line segment (mb) is a nice set of points. As we have just explained this implies a simple method for generating points in T. To generate all points in (abc) we first generate those in T and then generate the points on (bm) in the obvious way.

From this discussion we obtain the following algorithm for computing the *ith* point in triangle (abc). We assume that the quantities $k = (y_c+1)(x_b+1)$ and $gbc = gcd(|x_b|, |y_c|)$

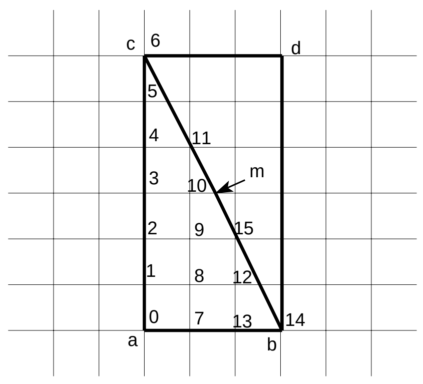

Figure 3: Indices of points generated by $Index$.

have been precomputed.

```
Index(i){
if $i < \lfloor k/2 \rfloor$ then
   /* generate $ith$ point in $T$ */
   {$x_p := \lfloor \frac{i}{y_c+1} \rfloor$;
   $y_p := i - x_p(y_c + 1)$;
   if $P$ lies outside $(abc)$ reflect $P$ about $m$; }
else /* generate the point with index $i - \lfloor k/2 \rfloor$ on $(bm)$ */
      {$j := i - \lfloor k/2 \rfloor$;
      $x_p := x_b - j \cdot x_b/gbc$;
      $y_p := j \cdot y_c/gbc$; }}
```

Figure 3 shows the ordering in which the points in the triangle of Figure 2 are generated by $Index$. Note that the points with indices 12 and 13 are obtained by reflecting the points with coordinates $(1, 5)$ and $(1, 6)$ in R.

This indexing method is very fast: precomputing $gcd(|x_b|, |y_c|)$ (using the Euclidean algorithm) takes time proportional to the number of bits in the binary representations of the integers x_b and y_c. The number of steps required by $Index$ is a small constant (depending on the particular implementation), assuming $\lfloor \frac{i}{y_c+1} \rfloor$ can be computed in constant time.

4 The General Case

We now consider a triangle (abc) in general position. Again we let a be the origin of a Cartesian coordinate system with the neighboring grid points to the right and above a having coordinates $(1, 0)$ and $(0, 1)$. Figure 4 shows an example of such a triangle. The possibility that none of the three sides lie on a grid line seems to complicate the indexing problem. Fortunately, many of the ideas introduced in the last section can be applied here after some modifications.

Let d be the reflection of a about the midpoint m of (bc). We shall first study the indexing problem for the parallelogram $(abcd)$. The points in $(abcd)$ other than m are naturally partitioned into disjoint pairs of points that are reflections of each other about m. As before we define a set of points to be *nice* if it contains exactly one point from each pair. Note that the set of points in (abc) that do not lie on (bm) is again a nice set of points. By an

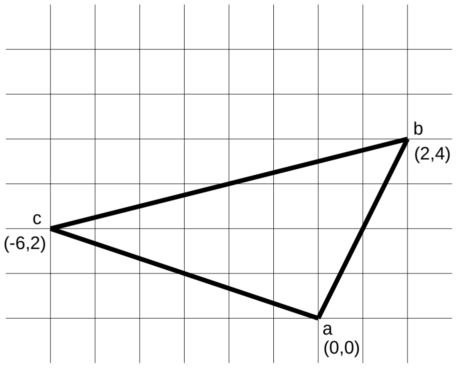

Figure 4: A triangle in general position.

observation in the last section all that remains to be done is to develop a fast indexing method for some fixed nice set of points.

It turns out, somewhat surprisingly, that there is such a set of points having a very simple structure. This is best understood by considering a new coordinate system having a as its origin and basis vectors ab and ac. We call the coordinates of a point in this system *skew coordinates*. Note that the skew coordinates of points b, c and d are $(1, 0)$, $(0, 1)$ and $(1, 1)$, respectively.

Consider the lexicographic ordering of the points in $(abcd)$ by their skew coordinates. Let k be the number of points in $(abcd)$ and let R denote the set comprising the first $\lfloor k/2 \rfloor$ points in the lexicographic ordering. It is not difficult to see that R is a nice point set.

It is less obvious that the points in R can be indexed quickly. This can be seen by carefully examining the distribution of those points. First we prove that all points in $(abcd)$ lie on equidistant lines parallel to (ac). For this we use basic analytic geometry and some elementary number theory. The equation of a line parallel to (ac) is of the form

$$x_c y - y_c x = \alpha. \tag{1}$$

Since we are interested only in those lines containing grid points, we may assume that α is an integer. Let $gc = gcd(|x_c|, |y_c|)$. By Theorem 1, Equation 1 has a solution if and only if gc divides α. In this case we may rewrite Equation 1 as

$$\frac{x_c}{gc} y - \frac{y_c}{gc} x = r \tag{2}$$

for some integer r. Let $D = x_c y_b - y_c x_b$. Note that $|D|$ is the area of parallelogram $(abcd)$. As we shall see this quantity will play an important role in the indexing procedure. Substituting coordinates for a and b in Equation 2 we deduce that the lines containing points of $(abcd)$ correspond to the integer values of r between $\frac{D}{gc}$ and 0. Note that the distance separating two consecutive points on such a line is no larger than the distance from a to c. Thus each such line contains at least one (grid) point in $(abcd)$. We have thus shown that the points of $(abcd)$ lie on a family of equidistant lines parallel to (ac). Let L denote the set of these lines. From the above discussion it follows that $|L| = \frac{|D|}{gc} + 1$. Figure 5 shows the lines in L for the parallelogram associated with the triangle in Figure 4. Note that in this example $|L| = 15$ in accordance with our formula.

Next we examine the distribution of points on these lines. We distinguish two types of lines in L: those intersecting each of (ab) and (cd) in a (grid) point, termed of type 1, and those

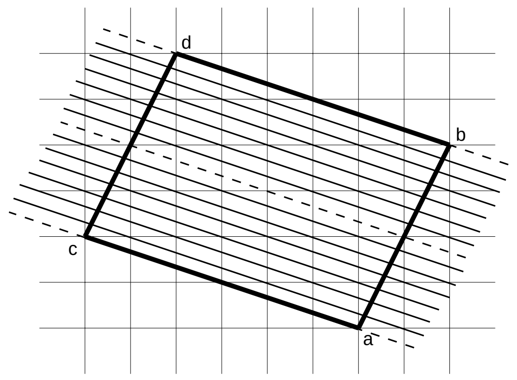

Figure 5: Equidistant lines in L containing the points in $(abcd)$.

that intersect neither one in a grid point, termed of type 2. Note that each line in L is either of type 1 or type 2. Let $gb = gcd(|x_b|, |y_b|)$. There are exactly $gb+1$ equally spaced lines of type 1 corresponding to $gb+1$ equally space points on (ab). Each type 1 line contains $gc+1$ points while each line of type 2 contains gc points. Since there are $gb+1$ lines of type 1, the total number of points in $(abcd)$ is $k = (gb+1)(gc+1)+(|L|-gb-1)gc = gb+gc+1+|x_c y_b - y_c x_b|$. The lines of type 1 are represented by dashed line segments in Figure 5 while those of type 2 are depicted by solid line segments. In this example there are three lines of type 1 containing three points each. The remaining twelve lines of type 1 have 2 points each. Thus we have $k = 33$ in this example which agrees with our formula.

Type 1 and type 2 lines alternate in a regular pattern. We order the lines in L in increasing order of their distance from (ac). Any two consecutive lines of type 1 are separated by the same number of type 2 lines. Let us call the set consisting of a type 1 line and the following type 2 lines up to (excluding) the next type 1 line a *group*. We note that one group consists of line segment (bd) containing $gc+1$ points while the other gb groups contain $(k-gc-1)/gb$ points each.

Ordering the lines in L by increasing distance from (ac) induces an ordering of the groups. The points on each line are ordered according to the lexicographic order of their skew coordinates or, equivalently, in increasing order of their distance from (ab). Based on these orderings we number the groups, the lines within each group, and the points on each line with nonnegative integers starting at 0. Each point in $(abcd)$ is thus uniquely determined by the group index, the line index within this group, and the point index of the point on this line. Let the variables *group*, *line* and *point* denote these quantities. As an example consider the center of the parallelogram depicted in Figure 5. It is the second point of the first line in the second group and thus satisfies $group = 1$, $line = 0$ and $point = 1$.

In view of the above discussion these quantities are easily computed from the index i of a point in $(abcd)$:

```
$group := \lfloor i \cdot gb/(k-gc-1) \rfloor;$
$j := i - group \cdot (k-gc-1)/gb;$
if $j \leq gc$ then { $line := 0;$
                      $point := j;$}
else { $line := \lfloor (j-gc-1)/gc \rfloor + 1;$
       $point := (j-gc-1) \bmod gc;$ }
```

Having computed *group, line* and *point* for the point P with index i in $(abcd)$ we still have to determine the actual coordinates of P. If F is the first point having the same *group* and *line* index as P (thus satisfying $point = 0$) then we can express the coordinates of P simply as $x_P = x_F + point \cdot x_c/gc$ and $y_P = y_F + point \cdot y_c/gc$. We are thus left with the problem of computing the coordinates of F. For this we take another look at Equation 2. The *jth* line in L (in increasing order of distance from (ac)) corresponds to setting $r = sign(D) \cdot j$. Since there are $(|L| - 1)/gb = \frac{|D|}{gb \cdot gc}$ lines per group, the index of the unique line with given *group* and *line* values is $j = group \cdot \frac{|D|}{gb \cdot gc} + line$ and the corresponding r-value in Equation 2 is $r = sign(D) \cdot (group \cdot \frac{|D|}{gb \cdot gc} + line) = group \cdot \frac{D}{gb \cdot gc} + sign(D) \cdot line$.

To compute the point F we first compute a particular solution of Equation 2 for this value of r using the procedure *solve* described in Section 2. Having computed a particular point (x_0, y_0) on the given line, we can compute F as follows. If y_1 is the y-coordinate of (x_0, y_0) in the skew coordinate system then the point F satisfies $x_F = x_0 - \lfloor y_1 \cdot gc \rfloor \cdot x_c/gc$ and $y_F = y_0 - \lfloor y_1 \cdot gc \rfloor \cdot y_c/gc$. Note that $y_1 = \frac{x_b y_0 - x_0 y_b}{x_b y_c - y_b x_c}$.

In summary we have described how to compute the *group*, *line* and *point* values corresponding to the *ith* point in $(abcd)$ and how to compute the coordinates of the point specified by these values. We have thus given a method for indexing the points in a particular nice set, namely the set of the first $\lfloor k/2 \rfloor$ points in $(abcd)$ arranged in lexicographic order of their skew coordinates. To compute the *ith* point in triangle (abc) we now proceed exactly as described in the last section. The resulting procedure has the same complexity as in the special case: after precomputation time proportional to the number of bits in the vertices of the triangle, the point with a given index can be computed in constant time (assuming the floor function takes constant time).

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[2] J.D. Foley et al., Computer Graphics: Principles and Practice, Addison-Wesley, 1990.

[3] A. S. Glassner,ed., Graphics gems, 1990.

[4] J. Hughes, Personal Communication, 1993.

[5] I. Niven and H. S. Zuckerman, An Introduction to the Theory of Numbers, John Wiley & Sons, 1980.

Appendix

We now give the full pseudo-code for the function *Index* derived in the last section. We assume that the following quantities have been precomputed: $k = gb + gc + 1 + |D|$ (number of points in $(abcd)$), $gb = gcd(|x_b|, |y_b|)$, $gc = gcd(|x_c|, |y_c|)$, $gbc = gcd(|x_c - x_b|, |y_c - y_b|)$, $D = x_c y_b - y_c x_b$, a solution (x, y) to the equation $\frac{x_c}{gc} y - \frac{y_c}{gc} x = 1$.

```
Index(i){
if $i < \lfloor k/2 \rfloor$ then {/* generate $ith$ point in triangle $(abc)$ */
  $group := \lfloor i \cdot gb/(k - gc - 1) \rfloor$;
  $j := i - group \cdot (k - gc - 1)/gb$;
  if $j \leq gc$ then { $x_P = group \cdot x_b/gb + j \cdot x_c/gc$;
                        $y_P = group \cdot y_b/gb + j \cdot y_c/gc$};
  else { $line := \lfloor (j - gc - 1)/gc \rfloor + 1$;
         $point := (j - gc - 1) \bmod gc$;
         $r = group \cdot \frac{D}{gb \cdot gc} + sign(D) \cdot line$;
         $x_0 := rx$;
         $y_0 := ry$;
         $x_P = x_0 - \lfloor \frac{x_0 y_b - y_0 x_b}{D} \cdot gc \rfloor \cdot x_c/gc + point \cdot x_c/gc$;
         $y_P = y_0 - \lfloor \frac{x_0 y_b - y_0 x_b}{D} \cdot gc \rfloor \cdot y_c/gc + point \cdot y_c/gc$};
  if $P$ lies outside $(abc)$ reflect $P$ about $m$ };
else {/* generate the point with index $i - \lfloor k/2 \rfloor$ on $(bm)$ */
     $j := i - \lfloor k/2 \rfloor$;
     $x_P := x_b + j \cdot (x_c - x_b)/gbc$;
     $y_P := y_b + j \cdot (y_c - y_b)/gbc$; }}
```