Proceedings of IDETC/CIE 2005
ASME 2005 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
September 24-28, 2005, Long Beach, California, USA

DETC2005-85513

POLYGON OFFSETTING BY COMPUTING WINDING NUMBERS

Xiaorui Chen **Sara McMains**

Department of Mechanical Engineering
University of California, Berkeley
Email: {xchen | mcmains} @me.berkeley.edu

ABSTRACT

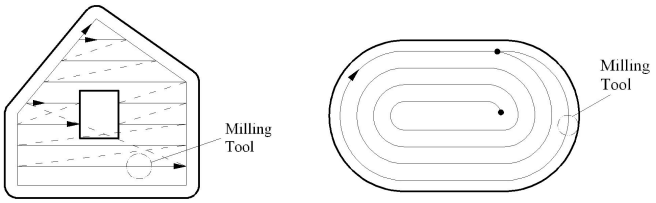
In this paper we present a simple new algorithm to offset multiple, non-overlapping polygons with arbitrary holes that makes use of winding numbers. Our algorithm constructs an intermediate "raw offset curve" as input to the tessellator routines in the OpenGL Utility library (GLU), which calculates the winding number for each connected region. By construction, the invalid loops of our raw offset curve bound areas with non-positive winding numbers and thus can be removed by using the positive winding rule implemented in the GLU tessellator. The proposed algorithm takes $O((n+k)\log n)$ time and $O(n+k)$ space, where n is the number of vertices in the input polygon and k is the number of self-intersections in the raw offset curve. The implementation is extremely simple and reliably produces correct and logically consistent results.

KEYWORDS

Polygon Offsetting, Winding Number, OpenGL, CAD/CAM

1 Introduction

Offsetting is a fundamental problem in CAD/CAM. Beyond the use of offsets as design primitives, for manufacturing they are a critical tool for analysis and process planning. In order to generate tool paths for $2\frac{1}{2}$D pocket machining, for example, the boundary of each pocket must first be offset inward by a distance equal to the radius of the cutting tool to avoid gouging [1–5]. For direction parallel tool path generation (see Figure 1(a)), the tool path includes the line segments inside the pocket generated by intersecting the offset boundaries with equidistant parallel lines; for contour-parallel tool path generation (see Figure 1(b)), the original boundaries are offset successively and the offset curves are chained together into the tool path [2, 3, 6].



(a) Direction parallel one-way cutting (b) Contour parallel spiral

Figure 1. *Tool path generation in pocket machining*

We can also use offsetting to find the accessible area for a given tool radius [7], a substep in finding an optimal set of cutter radii for pocket machining [8]. The boundary of the pocket is first offset inward by the tool radius to get the tool path boundary. We then offset the result outward by the same offset distance to find the accessible area and subtract the result from the original boundary of the pocket using a Boolean difference. If the area

of the difference is zero, we don't need a smaller tool to remove the excess material. Otherwise, a smaller tool is necessary to clean up the corners. An example is shown in Figure 2. The shaded area is the inaccessible area for a tool of radius equal to the offset distance.

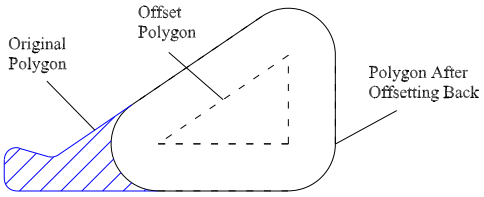
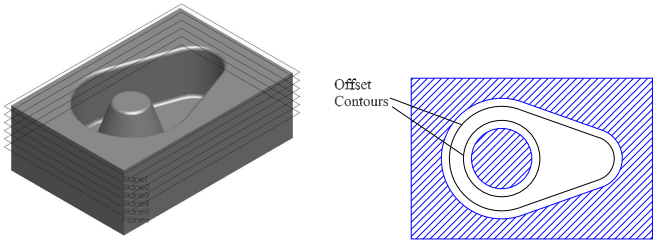


Figure 2. *Difference between the original polygon and the polygon after offsetting back*

Rough machining of molds or dies also uses offsetting. The die/mold cavity can be sliced into a set of 2D contours by a set of parallel planes perpendicular to the draw direction (see Figure 3). Then we can offset these contours inward by a distance slightly larger than the radius of the end mill to accommodate the uncut-allowance. The unwanted material in the interior of the offset polygon is removed by machining each layer from top to bottom [9].



(a) Die/Mold (b) Slice by plane 4

Figure 3. *Rough machining of dies/molds*

Offsetting is also used in rapid prototyping. For generating deposition-nozzle or laser scanning paths, a tessellated STL model is first sliced to get the external contours. For each layer, the contours are offset to construct tool paths similar to those used in pocket machining. 2D offsets of the layers, in combination with Boolean constructive solid geometry (CSG) operations, can also be used to approximate 3D offsets that hollow out objects in order to save build time and material consumption [10].

Robot motion planning is yet another area that uses offsetting [11]. For an autonomous robot moving on the ground, its work space can be modeled as a 2D polygon with arbitrary holes, where the outer polygon boundary represents the extent of the floor plan and the holes represent the obstacles to be avoided. Offsetting this general polygon inward by the dimension of the robot gives the boundary of the space in which the robot can move freely.

Our algorithm is related to the conventional offsetting approaches that offset each edge segment of the polygon and insert circular arcs to close the gaps between the offset segments (see Figure 4). The resulting curve is called a *raw offset curve*. A raw offset curve usually contains *invalid loops*, which must be removed to produce the offset polygon [2, 12–15].

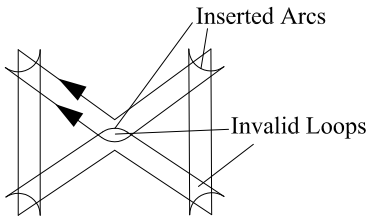


Figure 4. *Conventional pair-wise offset approach*

In this paper we present a simple new algorithm to offset multiple, non-overlapping polygons with arbitrary holes using a variation of the traditional raw offset curve and calculating the winding numbers of its connected regions. Our algorithm constructs a raw offset curve as input to the tessellator routines in the OpenGL Utility library (GLU), which calculates the winding number for each connected region. By construction, the invalid loops of our raw offset curve bound areas with non-positive winding numbers and thus can be removed by using the positive winding rule implemented in the GLU tessellator. The output is the contour(s) of the offset polygon. Using a GLU tessellator implementation that employs a sweep line algorithm, our algorithm takes $O((n+k)\log n)$ time and $O(n+k)$ space, where n is the number of vertices in the input polygon and k is the number of self-intersections in the raw offset curve. The implementation is extremely simple and reliably produces correct and logically consistent results, unlike the offset routine in the commercial ACIS geometry kernel.

We next review related work, then present assumptions and preliminary definitions, describe the algorithm, and prove its correctness. A discussion of the implementation and performance follows.

2 Related Work

One offsetting approach is to use Voronoi diagrams. Persson was the first to apply Voronoi diagrams to generating cutter paths for machining arbitrarily shaped pockets [1]. Chou and Cohen use Voronoi diagrams to create contour-parallel paths [16]. Kim gives a simple but efficient Voronoi diagram based algorithm to compute a trimmed offset of a single simple polygon consisting of line segments and arcs [17].

The computation of the Voronoi diagram is subject to numerical robustness issues, a result of generating intermediate geometric entities during its construction. Smith describes a divide and conquer algorithm, similar to that used by Held [3], to build the Voronoi diagram. Smith, however, minimizes the floating point round-off error and uses adaptive-precision arithmetic to address numerical robustness issues [18]. Though offsetting algorithms based on Voronoi diagrams are fast, creating the Voronoi diagram itself can be slow, and a robust implementation is extremely complex to implement.

Another offsetting method is pairwise intersection. Rossignac and Requicha describe a CSG-based algorithm in which the pocket boundaries are first offset and then the interfering sections are identified by calculating the distance between the offset segments and the pre-offset boundary [19]. Hansen and Arbab use interference indices to detect and delete the "gouging" sections of the offset boundaries [2]. Yang and Huang develop two criteria for loop validation by examining the geometry among the offset segments [13]. Barton calculates the inflation/deflation of a 2D polygon by using a bounding box hierarchy [12]. Choi and Park achieve a fast algorithm by removing all local invalid loops before the raw offset curve is constructed, but their method is applicable only to polygons without holes [15].

3 Assumptions and Preliminary Definitions

The input to our algorithm is a set of non-overlapping 2D polygons, each bounded by oriented straight line edges. Each polygon consists of one peripheral contour and zero or more inner contours, which form holes or "islands" in the polygon. We preprocess the input to assure that each polygon is non-self-intersecting (in other words, no edge contains a point in the interior of another). The polygons can be non-manifold in the sense that more than two edges can touch each other at some endpoint(s). We will use the right-hand rule convention that the edges of each contour are directed such that the interior of the polygon lies to the left. That is, the peripheral contour is oriented counterclockwise (CCW) and each inner contour is oriented clockwise (CW). The normal of each edge is perpendicular to the edge and points to the right, i.e., the exterior of the polygon.

In our algorithm, which we will describe in Section 4.1 in detail, we treat the vertices differently depending on whether the vertex is convex or concave.

Definition 1. *A vertex is* convex *if a left turn is made at this vertex while marching along the contour. A vertex is* concave *if a right turn is made at this vertex while marching along the contour (see Figure 5).*

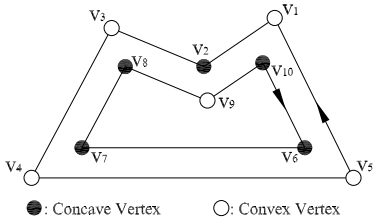


Figure 5. *Convex and concave vertices in a polygon with a hole*

Another important concept we will use is the *winding number*. While there are different approaches to defining the term [20–24], the winding numbers calculated using any of these definitions are identical. In this paper, we use the following definition:

Definition 2. *Let P be a set of oriented polygons consisting of one or more contours, q be any point of $\Re^2 \backslash P$, where $\Re^2$ is the 2D Euclidean space, and R be any ray from q to infinity that intersects no vertex of P. The* winding number $\omega(R,P)$ *of R with respect to P is:*

$$\omega(R,P) = \sum_{e_i \in P} \psi(R,e_i)$$

where, for each edge e_i, the index $\psi(R,e_i)$ is defined as follows:

$$\psi(R,e_i) = \begin{cases} 0 & \text{if } R \text{ does not intersect } e_i; \\ 1 & \text{if } e_i \text{ crosses } R \text{ in CCW direction as viewed from } q; \\ -1 & \text{if } e_i \text{ crosses } R \text{ in CW direction as viewed from } q. \end{cases}$$

From Definition 2, we have the following properties. First, $\omega(R,P)$ takes the same value for all rays R with the same start point provided that R does not intersect any vertex of P. Second, the winding number for a point q, denoted by $\omega(q,P)$, has the same value for all points in a single connected region (a connected set of points bounded by the contours of the polygons, not including its boundary). Third, the winding numbers of adjacent regions separated by a single edge will always differ by one. Fourth, the individual winding numbers with respect to each contour can be summed to get the winding number with respect to

the polygon when there are multiple contours defining the polygon.

An example of calculating the winding number for a point q using two different choices of R is shown in Figure 6. Note that winding numbers can be defined for overlapping and/or self-intersecting polygons, but only for closed contours.

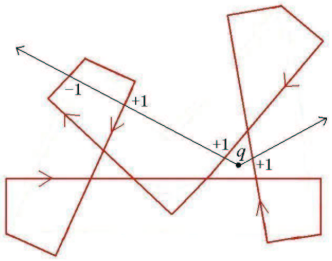
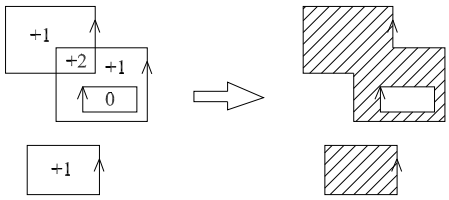


Figure 6. $\omega(q,P)$, *winding number at point q, is equal to* $+1$ *with respect to any ray from q. For the rays shown, the incremental difference* $(+/-1)$ *to the winding number as it crosses each polygon edge is indicated.*

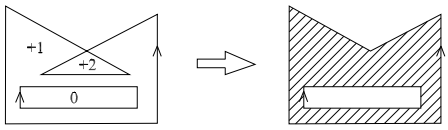
The *winding rule* defines a category, such as odd, nonzero, positive, negative, or "absolute value greater than or equal to 2", to classify a region as inside or outside. If the calculated winding number of a region falls into the chosen category, it is classified as inside. Other winding rules could also be defined, but these five are the only ones implemented in OpenGL, with the names GLU_TESS_WINDING_ODD, GLU_TESS_WINDING_NONZERO, GLU_TESS_WINDING_POSITIVE, GLU_TESS_WINDING_NEGATIVE, and GLU_TESS_WINDING_ABS_GEQ_TWO, respectively (see Figure 7). Winding rules GLU_TESS_WINDING_ODD and GLU_TESS_WINDING_NONZERO are commonly used in polygon fill procedures (only regions classified as inside are filled). Winding rules can be used to implement CSG Boolean operations such as union, difference and intersection of contours. For example, we can use the winding rule GLU_TESS_WINDING_ABS_GEQ_TWO to get the intersection of two contours [20].

In this paper, we use the positive winding rule: only regions with positive winding numbers are classified as in the interior of the polygon. We first use the positive winding rule to clean up any overlapping and/or self-intersecting polygons in a preprocessing step (see Figure 8). We later use the same winding rule to extract the offset from our raw offset curve, as we will describe in Section 4.1.

The inner offset and the outer offset of a polygon are defined as follows.



(a) Preprocessing overlapping polygons



(b) Preprocessing self-intersecting polygons

Figure 8. *Preprocessing; the winding numbers of each connected region are shown on the left and the interior of the polygon, by the positive winding rule, is shown on the right.*

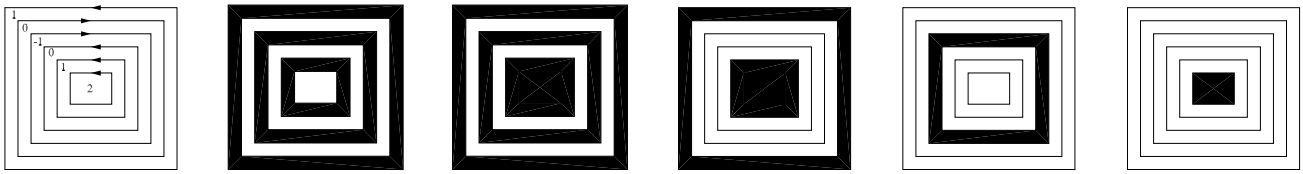
Definition 3. *The* inner offset *of the polygon P is the regularized [25] boundary of the set of points each of which lies in the interior of P and has a Euclidean distance greater than the offset distance d from the boundary of P.*

Definition 4. *The* outer offset *of the polygon P is the regularized boundary of the set of points each of which lies exterior to P and has a Euclidean distance greater than the offset distance d from the boundary of P.*

Note that the distance of a point q from the boundary of a polygon is the distance between q and the closest point on the boundary, and that there may be multiple points on the boundary that are closest to the given point. For instance, in Figure 9, the distance from interior point a (respectively, exterior point c) to the polygon is the distance from point a to point d (respectively, from point c to point g). The distance of point b from the polygon is the distance from point b to either point e or point f.

4 Winding Number Offset Algorithms

Conventional pair-wise offsetting approaches offset each edge and insert counterclockwise (respectively, clockwise) circular arcs at convex (respectively, concave) vertices between the nontangent offset segments to get the raw, self-intersecting offset curve (see Figure 4). They then calculate the self-intersections in the raw offset curve and identify and remove both local and global invalid loops [2, 12, 13, 15]. Identifying the invalid loops,



(a) Contours and winding numbers for each region (b) Odd (c) Nonzero (d) Positive (e) Negative (f) Abs. Val. ≥ 2

Figure 7. *Winding Rules*

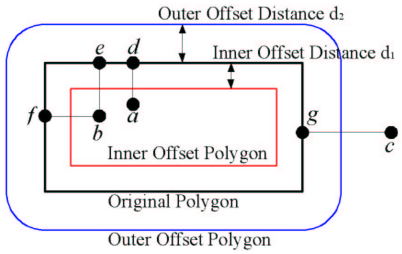


Figure 9. *Distance between a point and a polygon*

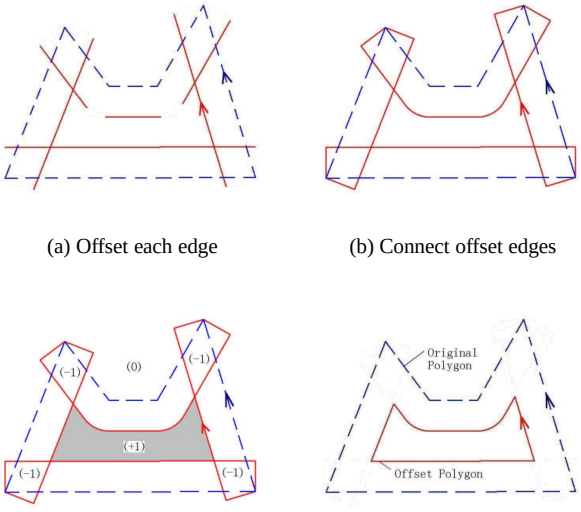
particularly those caused by global interactions, while retaining valid loops that are part of the offset boundary, lies at the heart of these algorithms.

Calculating the winding numbers of the raw offset curve obtained by the conventional pair-wise offset approaches initially appears to be a valid method for differentiating invalid loops. A simple example is shown in Figure 10. For a relatively small offset distance such as shown in Figure 10(a) and (b), the boundary of the area with positive winding number is the inner offset polygon. However, for a large offset distance such as the one in Figure 10(c), the result is incorrect: the resulting inner offset polygon should be empty, but the area with positive winding number is nonempty. Therefore, we instead construct a variation on the traditional raw offset curve that allows us to use winding numbers to efficiently determine the offset polygons.

4.1 Our Algorithm

To construct our raw offset curve for the inner offset polygon, we first offset each edge opposite to its normal direction (i.e., to the left of the edge) by the offset distance d (Figure 11(a)). The offset segments have the same orientation as the original edges. If the vertex is a concave vertex, we connect the endpoints of the offset edges whose original edges share this vertex by a clockwise oriented arc centered on this shared vertex. These segments and arcs are identical to those typically used in other approaches that construct a raw offset curve. If the vertex is a convex vertex, however, we connect it with the endpoints of the offset segments whose original edges share this convex vertex with straight line segments rather than inserting an arc (Figure 11(b)). The inserted edges are oriented such that the connectivity is maintained when you go around the raw offset curve.

Then we consider the winding numbers of each region with respect to the raw offset curve. Regions with positive winding numbers are in the interior of the inner offset polygon (Figure 11(c)). The boundary of their union is the inner offset polygon (Figure 11(d)).



(a) Offset each edge (b) Connect offset edges

(c) Calculate the winding number of each region (d) Get the boundary of areas with positive winding numbers

Figure 11. *Constructing the inner offset polygon (original polygon dashed)*

To construct our raw offset curve for the outer offset polygon, we follow the same steps as for an inner offset polygon with

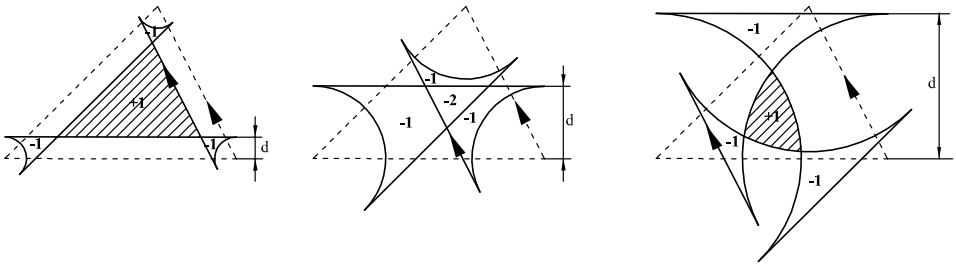
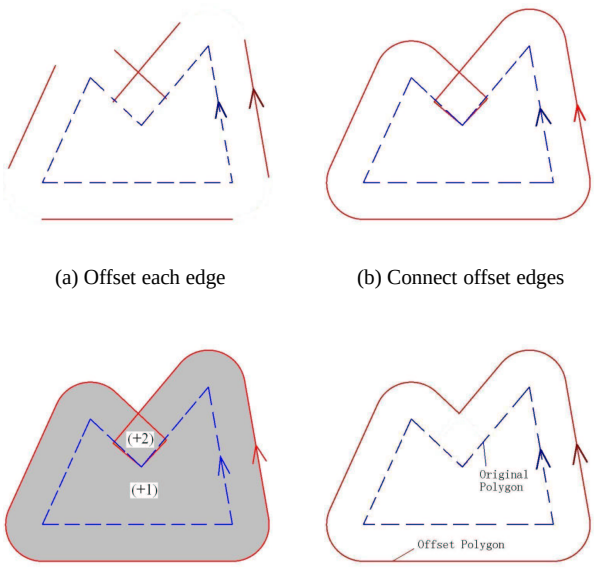


Figure 10. *Winding number calculation using conventional pair-wise inner offset raw offset curve; the shaded area in each figure has positive winding number and the original polygon is dashed.*

minor modifications. In the first step, we offset each edge *along* its normal direction (i.e., to the right of the edge) by the offset distance. In the second step, for a convex vertex, we connect the endpoints of the offset segments whose original edges share this vertex by a *counterclockwise* oriented arc centered on this shared vertex; for a concave vertex, we connect it to the endpoints of the offset edges whose original edges share this concave vertex. An example of constructing the outer offset polygon is shown in Figure 12.



(a) Offset each edge

(b) Connect offset edges

(c) Calculate the winding number of each region

(d) Get the boundary of areas with positive winding numbers

Figure 12. *Constructing the outer offset polygon (original polygon dashed)*

4.2 Correctness of the algorithm

In this section, we first show that the offset polygons can be constructed by Boolean operations, which can in turn be implemented using winding numbers. After proving the correctness of this method, we then show that our method is equivalent but takes less time and space.

Definition 5. *The* rectangular area *of an edge segment is the area swept out when sweeping the edge segment along or opposite to its normal direction until reaching its offset segment. (The sweep direction is along the normal for an outer offset, opposite to the normal for an inner offset.)*

Definition 6. *The* fan-shaped area *of a vertex is the fan-shaped area formed by the vertex and the inserted arc centered at the vertex and connecting its two offset points. The fan-shaped area is only defined on concave vertices for an inner offset and on convex vertices for an outer offset.*

We illustrate Definition 5 and Definition 6 with the examples shown in Figure 13.

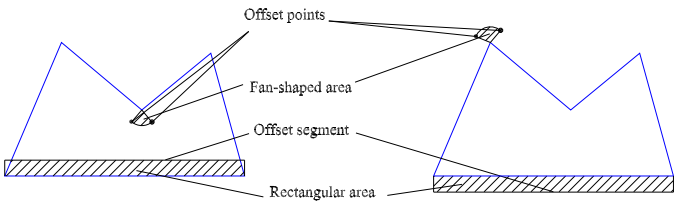


Figure 13. *Definitions of rectangular areas and fan-shaped areas*

4.2.1 Inner Offset To construct the inner offset polygon by Boolean subtraction, we need to subtract all points within

distance d of the boundary, that are also inside the boundary, from the original polygon. This is accomplished by subtracting the rectangular area for each edge segment and the fan-shaped area for each concave vertex. We orient the boundary clockwise for each contour we wish to subtract. Combining these contours with the contours defining the original oriented polygon P and using the positive winding rule to evaluate the resulting polygon set P' corresponds to a Boolean subtraction [20]. The boundary of the resulting set of regions with positive winding numbers is the inner offset polygon of P.

To see why this gives the inner offset, consider calculating the winding numbers first with respect to each contour of P'. Since the original polygon is not self-intersecting, the winding number is $+1$ in its interior, and 0 in its exterior. The points inside each subtracted rectangular or fan-shaped clockwise contour have a winding number of -1 relative to that contour since the boundary of any rectangular or fan-shaped area will not be self-intersecting. The points outside each subtracted contour have a winding number of 0 relative to that contour.

Now we calculate the winding numbers with respect to P'. Since the winding number of a point with respect to P' equals the sum of the winding numbers of this point with respect to each contour of P', we know a point o outside the original polygon P will have a winding number of at most 0 with respect to P'. The winding number with respect to P' of o equals 0 when o does not lie in any of the rectangular or fan-shaped area; the winding number with respect to P' of o is less than 0 if o lies in one or more of the rectangular or fan-shaped areas. In either case o will be classified as outside the evaluated polygon P' using the positive winding rule. For a point that lies in at least one rectangular or fan-shaped area, it will have a winding number of at most $(-1)+(+1)=0$ (when the point lies both in one of the rectangular or fan-shaped area and in the interior of the polygon P). So these points will also be classified as outside the evaluated polygon P' using the positive winding rule. Now we can express the process as a subtraction, from the interior of the original polygon P, of the union of all the rectangular areas and fan-shaped areas. The result is the set of points that lie in the interior of P and have a distance of at least the offset distance d from P. The boundary of this region is thus the inner offset polygon.

4.2.2 Outer Offset The construction of the outer offset polygon by Boolean union (addition) and its proof are similar to that for the inner offset polygon. To construct the outer offset polygon, we first construct the rectangular area for each edge segment and the fan-shaped area for each *convex* vertex. Then we orient the boundary *counterclockwise* for each rectangular or fan-shaped area. Combining these contours with those defining the original polygon P and again using the positive winding rule to evaluate the new polygon P' corresponds to a Boolean union (because this time the boundaries are counterclockwise) [20]. Since the winding number of a point in the interior (respectively, exterior) of the original polygon P is 1 (respectively, 0) with respect to P and the winding number of a point inside (respectively, outside) a rectangular or fan-shaped area is 1 (respectively, 0) with respect to that rectangular or fan-shaped area, the winding number of a point is at least 1 with respect to the polygon P' after the winding numbers with respect to each contour are summed if and only if the point lies in the interior of the original polygon P and/or in one or more rectangular or fan-shaped areas. Therefore, the point is in the interior of the new polygon P' using positive winding rule if and only if it is in the interior of the original polygon P and/or in one or more rectangular or fan-shaped areas. This process is equivalent to adding the rectangular areas and fan-shaped areas to the interior of the original polygon P. The boundary of the resulting set of regions with positive winding numbers is thus the outer offset polygon.

4.2.3 Simplifying the Raw Offset Curve Using straight Boolean operations to obtain the offset polygons as detailed in Section 4.2.1 and Section 4.2.2, we need to construct a new contour with four edges for each edge in the original polygon, as well as a new contour with two straight edges and one curved edge for each concave or convex vertex. We can reduce the number of contours and edges, leaving only those used in our algorithm's raw offset curve, by eliminating the edges with the same geometry but opposite directions by applying the following theorem:

Theorem 1. *If there are two edges in a polygon having the same geometry and opposite directions, the winding number of any point is invariant after we remove these two edges.*

Proof. Suppose there are two edges e_1 and e_2 in a polygon P with the above properties (see Figure 14). After removing the two edges from the polygon P, we get a new polygon P'. The polygons P and P' have the same vertices. For any ray R with an endpoint q that contains no vertex of P and hence contains no vertex of P', either it crosses both edges e_1 and e_2 or it crosses neither. If it does not cross either edge, the winding number of point q is equal with respect to the polygon P or the polygon P' since the edges that contribute to the winding number of point q are the same for both the polygons P and P'. If the ray crosses both e_1 and e_2 in the polygon P, their contributions to the winding number of point q are $+1$ (or -1) and -1 (or $+1$) respec-

tively. For any ray R_q from q, we have

$$\begin{aligned}\omega(q,P) &= \sum_{e_i \in P} \psi(R_q, e_i) \\ &= \sum_{e_i \in P \setminus \{e_1,e_2\}} \psi(R_q,e_i) + \psi(R_q,e_1) + \psi(R_q,e_2) \\ &= \sum_{e_i \in P \setminus \{e_1,e_2\}} \psi(R_q,e_i) + 1 - 1 \\ &= \sum_{e_i \in P \setminus \{e_1,e_2\}} \psi(R_q,e_i) \\ &= \omega(q,P')\end{aligned}$$

Therefore, the winding number for any point is invariant after e_1 and e_2 are removed.

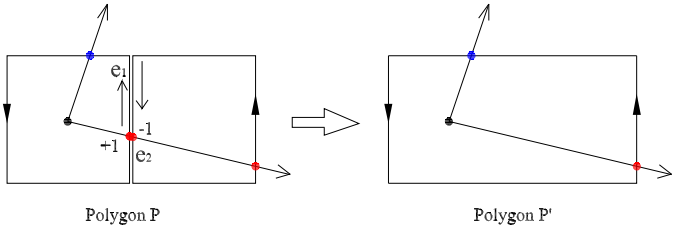
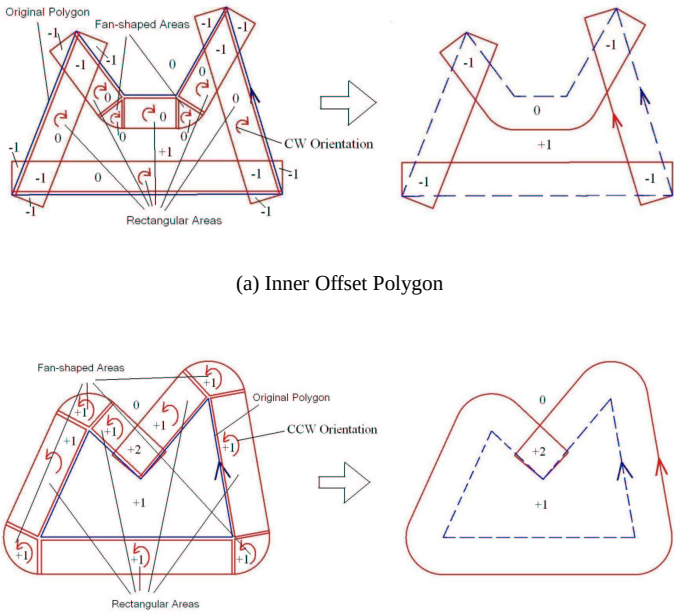


Figure 14. *Winding number invariance. Edges e_1 and e_2 are coincident but we move them a little apart from each other for visualization purposes.*

The reduction from the contours used for the full Boolean algorithm to the contours used for our raw offset curve is shown in Figure 15. Both the orientation of each rectangular or fan-shaped contour and the winding numbers of each single connected region are shown in the figure. We do not use the contour defining the original polygon in our raw offset curve so it is shown using dashed lines. Using our algorithm, the number of contours in the raw offset curve P' is equal to the number in the original polygon. That is, for each contour in the original polygon P, we construct only one contour in the raw offset curve. The total number of the edges in P' for our algorithm is between 28% and 75% of the number of the edges using the full Boolean subtraction/addition algorithm (both the greatest and least savings are for a convex polygon — the greatest savings for an outer offset, the least savings for an inner offset). Thus, our new algorithm takes less time and less space, with results equivalent to the full Boolean algorithm.



(a) Inner Offset Polygon

(b) Outer Offset Polygon

Figure 15. *Reducing the Boolean subtraction/addition algorithm to our winding number algorithm*

5 Implementation, Analysis and Results

Our program uses GLU 1.3 implemented by SGI, a free extension to OpenGL. The GLU tessellator takes as input the vertices of each contour defining the input polygon. After the GLU_TESS_WINDING_POSITIVE winding rule and the plane normal are set, it tessellates the polygon and calculates the winding numbers of each connected region. We set the GLU_BOUNDARY_ONLY property to GL_TRUE in order to extract the contours that separate the interior and the exterior of the resulting output polygon. The callback functions GLU_TESS_COMBINE, GLU_TESS_VERTEX, GLU_TESS_BEGIN and GLU_TESS_END are predefined and called during the tessellation. The GLU_TESS_COMBINE callback creates new vertices at self-intersections. The GLU_TESS_VERTEX callback allows us to extract the ordered vertices from the output polygons (the offsets). When the tessellator begins or ends a loop in their contour outlines, the GLU_TESS_BEGIN and GLU_TESS_END callback functions are called, telling us when to start a new loop or end an existing one in our data structure for the offset polygon.

To assure robustness, we preprocess the input to clean up any overlapping and/or self-intersecting polygons by running the winding number calculation once using the positive winding rule (see Figure 8). This clean up step is necessary with machine generated input because the round off errors that arise due

to limited numerical precision often generate microscopic self-intersections.

5.1 Space Complexity and Time Complexity

There are two main steps in our algorithm: the construction of the raw offset curve and the calculation of the winding numbers. Since each vertex in the raw offset curve is inserted only once, the first part takes $O(n)$ time and $O(n)$ space, where n is the number of line segments and arc segments in the raw offset curve. The second part depends on the GLU tessellator.

The GLU tessellator only takes polygons with straight line edges as input. Therefore, we must approximate each arc in the raw offset curve with short line segments, increasing the number of vertices passed to the GLU tessellator to some number $N > n$. We set the approximation error (the maximum distance between any approximating line segment and the original arc) to be 1% of the offset distance for the results reported here. Since the difference is bounded by the constant approximation percentage, $O(N) \in O(n)$, and the asymptotic complexity, in terms of N or n, is the same. In our analysis, we will use n, that is, the number of vertices in the raw offset curve, instead of N.

The source code for the GLU tessellator from the SGI Sample Implementation [26] reveals that they use a sweep line algorithm to subdivide the polygon into monotone regions at intersection points and classify each region as inside or outside according to the winding rule used. A sweep line intersection algorithm that stores intersections has $O((n+k)\log n)$ time complexity and $O(n+k)$ space complexity, where n is the number of vertices in the input, the raw offset curve in this instance, and k is the number of self-intersections; thus we assume that the GLU tessellator we are using also runs in $O((n+k)\log n)$ time and uses $O(n+k)$ space.

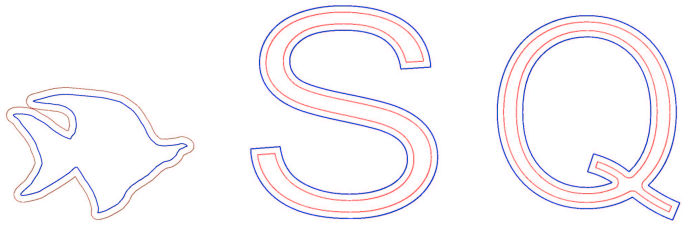
We have experimently confirmed our implementation's space and time complexity for several examples, shown in Figure 16. Their curved surfaces were approximated to several different levels of precision for comparison. We ran the tests on a dual-boot PC with an AMD Athlon(tm) XP 2500+ 1.8GHz processor under Linux using single user mode for repeatability.

We experimently confirmed the space complexity by plotting the memory usage of the program versus $n+k$ (see Figure 17). We can see that the space usage is linear in $n+k$, which is consistent with the space complexity of the sweep line algorithm used in the GLU tessellator.

Running times, taken as the average of 100 runs, are plotted versus $(n+k)\log n$ in Figure 18. These plots are consistent with the theoretical asymptotic time complexity of $O((n+k)\log n)$.

5.2 Comparison with ACIS

5.2.1 Running Time We compared the running time of our offsetting algorithm to the running time of the offset routine provided in the ACIS kernel, the geometric kernel used by



(a) Fish (272 × 186 bounding box)

(b) BigS (117 × 134 bounding box)

(c) BigQ (116 × 130 bounding box)

Figure 16. *Examples, with the original polygons in (darker) blue and the offset polygons in (lighter) red; the dimensions of the axis aligned bounding box for each example are indicated in parentheses.*

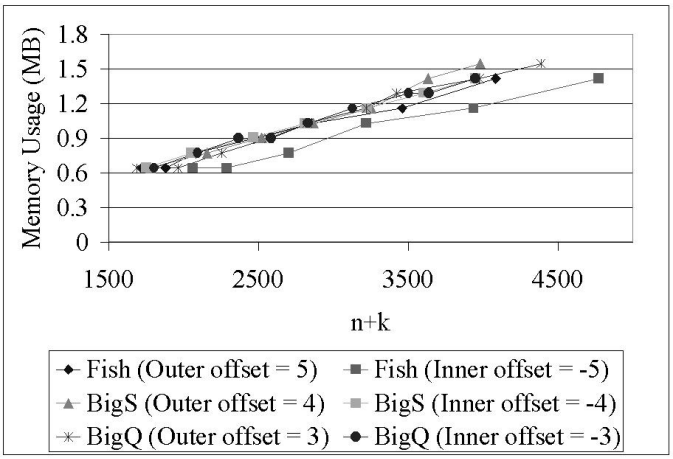


Figure 17. *Space usage vs.* $(n+k)$

many commercial CAD package such as AutoCAD, 3D Studio, Autodesk Inventor, IronCAD, and CADkey. We ran the tests on the same dual-boot 1.8GHz PC under Windows 2000 (see Figure 19). Our algorithm is 30 to 150 times faster than the ACIS offsetting package on these examples.

5.2.2 Robustness One of our motivations for starting this work was an observed lack of robustness in the ACIS routine for input polygons with features of size exactly equal to twice the inner offset distance. Therefore we also compared the robustness of the ACIS offsetting routine and our algorithm. While ACIS gives qualitatively incorrect results in some cases, our algorithm always gave correct results for the hundreds of cases we tested. A typical example is shown in Figure 20, in which the offset polygon should consist of two contours that touch each other at a point on the symmetric centerline. ACIS gives only one of

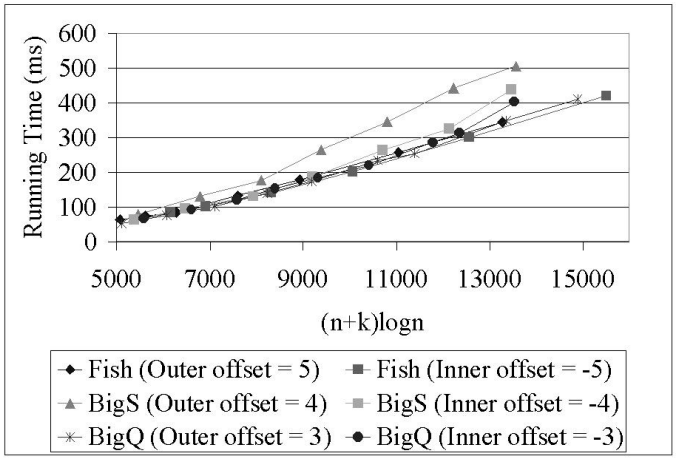


Figure 18. *Running time vs.* $(n+k)\log n$

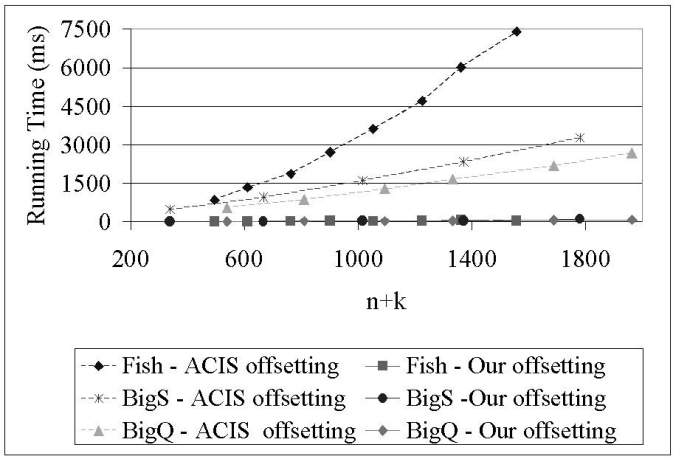
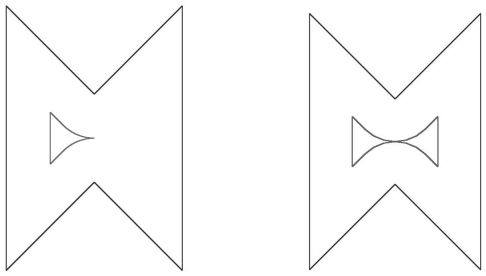


Figure 19. *Running time comparisons*

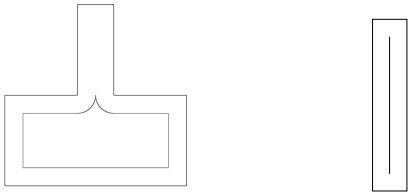
these contours, while the other contour is missing entirely. Our algorithm gives both contours correctly.

We also found out that the ACIS offsetting routine gives inconsistent results corresponding to different mathematical definitions of the offset polygon, depending on the input. The issue arises when two offset segments coincide with each other (an example is shown in Figure 21). ACIS gives two different, mathematically inconsistent results for the offsets of these two polygons, one containing the coincident offset segments and the other omitting them. Our algorithm does not display the coincident offset segments at any time, consistent with our definition of offset as the boundary of a regularized set (r-set) [25]. Having an offset algorithm that implements a consistent mathematical definition of offsetting is essential if we are to use it as a subroutine for well-defined, provably correct geometric algorithms.

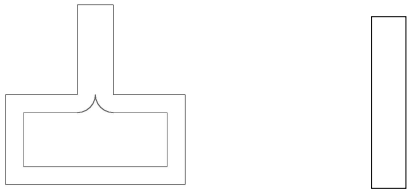


(a) ACIS result (b) Our result

Figure 20. *Robustness comparison: ACIS inner offset and our algorithm's inner offset*



(a) Polygon 1 by ACIS (b) Polygon 2 by ACIS



(c) Polygon 1 by our algorithm (d) Polygon 2 by our algorithm

Figure 21. *Consistency of the definition of the offset polygon*

6 Conclusion

Our algorithm for calculating offset polygons using winding numbers is fast, accurate, and extremely easy to implement. The simple-to-construct raw offset curve is processed by the built-in winding number capabilities of the robust, widely available and free GLU tessellator. The algorithm's scalability is excellent, with space complexity of $O(n+k)$ and running time of $O((n+k)\log n)$, where n is the number of input vertices and k is the number of self-intersections in the raw offset curve. The algorithm gives mathematically well-defined and internally consistent results, even for input that contains multiple, overlapping

and/or self-intersecting polygons with arbitrary holes.

ACKNOWLEDGMENTS

The authors were supported in part by a Junior Faculty Research Grant and MICRO grant 04-079.

REFERENCES

[1] Persson, H., 1978. "NC machining of arbitrarily shaped pockets". *Computer-Aided Design*, **10**(3), pp. 169–174.
[2] Hansen, A., and Arbab, F., 1992. "An algorithm for generating NC tool paths for arbitrarily shaped pockets with islands". *ACM Trans. Graph.*, **11**(2), pp. 152–182.
[3] Held, M., 1991. *On the Computational Geometry of Pocket Milling*. Lectures Notes in Computer Science. Springer-Verlag, Berlin.
[4] Manuel, D., Liang, M., and Kolahan, F., 1996. "A dynamic offsetting approach to tool path generation for machining convex pockets". *Computers & Industrial Engineering*, **31**(1-2), October, pp. 135–138.
[5] Choi, B. K., and Kim, B. H., 1997. "Die-cavity pocketing via cutting simulation". *Computer-Aided Design*, **29**(12), pp. 837–846.
[6] Park, S. C., 2004. "Sculptured surface machining using triangular mesh slicing". *Computer-Aided Design*, **36**(3), pp. 279–288.
[7] D'Souza, R., Wright, P., and Séquin, C., 2002. "Handling tool holder collision in optimal tool sequence selection for 2.5-D pocket machining". *Journal of Computing and Information Science in Engineering*, **2**(4), pp. 345–349.
[8] Nadjakova, I., and McMains, S., 2004. "Finding an Optimal Set of Cutter Radii for 2.5D Pocket Machining". In ASME International Mechanical Engineering Congress and Exposition.
[9] Choi, B. K., and Jerard, R. B., 1998. *Sculptured surface machining: theory and applications*. Kluwer Academic Publishers.
[10] McMains, S., Smith, J., and Sequin, C., 2003. "Thin-wall calculation for layered manufacturing". *Journal of Computing and Information Science in Engineering*, **3**(3), September, pp. 210–218.
[11] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., 2000. *Computational Geometry: Algorithms and Applications*, second ed. Springer-Verlag.
[12] Barton, E. E., and Buchanan, I., 1980. "The polygon package". *Computer-Aided Design*, **12**(1), pp. 3–11.
[13] Yang, S.-N., and Huang, M.-L., 1993. "A New Offsetting Algorithm Based On Tracing Technique". In Proceedings of the 2nd Symposium on Solid Modeling and Applications, ACM Press, pp. 201–210.
[14] Kalmanovich, G., and Nisnevich, G., 1998. "Swift and stable polygon growth and broken line offset". *Computer-Aided Design*, **30**(11), pp. 847–852.
[15] Choi, B., and Park, S., 1999. "A pair-wise offset algorithm for 2D point-sequence curve". *Computer-Aided Design*, **31**(12), pp. 735–745.
[16] Chou, J. J., and Cohen, E., 1989. Computing offsets and tool paths with Voronoi diagrams. Tech. rep., Technical Report UUCS-89-017, Department of Computer Science, University of Utah, Salt Lake City, UT 84112 USA.
[17] Kim, D.-S., 1998. "Polygon offsetting using a Voronoi diagram and two stacks". *Computer-Aided Design*, **30**(14), pp. 1069–1076.
[18] Smith, J., 2004. "Robust geometric methods for surface design and manufacturing". Phd dissertation, UC Berkeley, Department of Electrical Engineering and Computer Science, May.
[19] Rossignac, J. R., and Requicha, A. A. G., 1986. "Offsetting operations in solid modelling". *Computer Aided Geometric Design*, **3**(2), pp. 129–148.
[20] Woo, M., Neider, J., Davis, T., and Shreiner, D., 1999. *OpenGL Programming Guide*, third ed. Addison-Wesley.
[21] Grunbaum, B., and Shephard, G. C., 1990. "Rotation and Winding Numbers for Planar Polygons and Curves". *Transaction of the American Mathematical Society*, **322**(1), pp. 169–187.
[22] Hormann, K., and Agathos, A., 2001. "The point in polygon problem for arbitrary polygon". *Computational Geometry: Theory and Applications*, **20**, pp. 131–144.
[23] Sunday, D., 2001. Fast winding number test for point inclusion in a polygon. http://softsurfer.com/algorithms.htm.
[24] Ismailescu, D., 2003. "Slicing the Pie". *Discrete and Computational Geometry*, **30**(2), pp. 263–276.
[25] Requicha, A. G., 1980. "Representations for Rigid Solids: Theory, Methods, and Systems". *ACM Computing Surveys (CSUR)*, **12**(4), pp. 437–464.
[26] SGI Sample Implementation. http://www.mesa3d.org.