Algorithmica, 27 (1) (2000), pp. 57-86.

Shortest Path Geometric Rounding

Victor J. Milenkovic *
Department of Mathematics and Computer Science
University of Miami, Coral Gables, FL 33124
vjm@cs.miami.edu, http://www.cs.miami.edu/ṽjm

July 27, 2000

Abstract

Exact implementations of algorithms of computational geometry are subject to exponential growth in running time and space. In particular, coordinate bit-complexity can grow exponentially when algorithms are *cascaded*: the output of one algorithm becomes the input to the next. Cascading is a significant problem in practice. We propose a geometric rounding technique: *shortest path rounding*. Shortest path rounding trades accuracy for space and time and eliminates the exponential cost introduced by cascading. It can be applied to all algorithms which operate on planar polygonal regions, for example, set operations, transformations, convex hull, triangulation, and Minkowski sum. Unlike other geometric rounding techniques, shortest path rounding can round vertices to arbitrary lattices, even in polar coordinates, as long as the rounding cells are connected. (Other rounding techniques can only round to the integer grid.) On the integer grid, shortest path rounding introduces less combinatorial change and geometric error than the other rounding methods. Three algorithms are given for shortest path rounding, one of which we have used in industrial application software since 1992. In combination with recent advances in exact floating point evaluation of numerical primitives, shortest path geometric rounding yields a practical solution to numerical issues in computational geometry. Geometric algorithms can be implemented exactly on floating point input coordinates; the exact output coordinates can be rounded to accurate floating point approximations; and the cost of each arithmetic operation is only a little more than if it were implemented as a single hardware floating point operation.

Keywords: Computational geometry, Robust Geometry, Numerical Analysis, Numerical Issues, Rounding.

*This research was funded by NSF grants CCR-91-157993 and 97-12401 and by the Alfred P. Sloan Foundation.

1 Introduction

A number of very useful algorithms on planar polygonal objects have been described in the computational geometry literature. Among these are union, intersection, complement, Minkowski sum, convex hull, and triangulation (or other decomposition) of polygonal regions in the plane. Given an initial set of points and lines or line segments, these algorithms use only a few binary constructive geometric primitives: 1) join two points to create a new line or line segment, 2) intersect two lines or line segments to generate a new point, and 3) add two points (coordinate-wise) together to generate a new point. Symbolically, each point or line (segment) that an algorithm outputs lies at the root of a binary "construction tree" whose leaves are the input points or lines and whose nodes are constructive primitives. For most algorithms, these trees have constant depth. However, when CAD systems and other software systems apply these algorithms, they do not apply them in isolation. Usually, the algorithms are *cascaded*: the output from one algorithm becomes the input to another. Also, these systems might construct new points and lines by translating, rotating, scaling, or applying other transformations to geometric objects. From an algorithmic point of view, transformations are trivial, yet they also add to the numerical "history" of points and lines. In practice, the height of the construction tree for a point or line can grow without bound.

This paper argues that no matter which *exact* numerical representation one chooses, the *space* required to represent a point or line (segment) grows *exponentially* with the height of its construction tree. As a consequence, the running time of geometric algorithms also grows exponentially. To avoid exponential cost, a geometric system must employ *rounding*. Rounding trades accuracy for cost. Of course, rounding is not a new idea: for more than 30 years, computers have had hardware-supported rounded floating point arithmetic. An entire mature field of discipline, numerical analysis, addresses the problems arising from implementing numerical algorithms using rounded arithmetic. Unfortunately, geometric algorithms are subtly and exquisitely sensitive to rounding in ways not addressed by numerical analysis. For many years, such numerical problems were thought to be "bugs," but it is now well understood that one cannot naively (or even not so naively) implement geometric algorithms using rounded floating point arithmetic in place of exact real arithmetic.

This paper presents a technique called *shortest path geometric rounding* for rounding sets of points and line segments. Green and Yao first introduced the idea of rounding an arrangement of line segments to an integer grid [16]. Each line segment is replaced by a polygonal curve in a way that (in some sense) preserves the topology of the arrangement. Geometric rounding has the advantage that it is applied to geometric objects *after* they have been generated by a geometric algorithm, and therefore it solves the problem of exponential cost (albeit at the price of reducing accuracy) without any modification of existing geometric algorithms. Unlike Green and Yao's algorithm and snap rounding [15, 19] (see also [18]), shortest path geometric rounding 1) introduces the minimum possible geometric error, 2) introduces the minimum combinatorial change, and 3) can round vertices to *any* rounding lattice with connected rounding cells. The other methods can only round to the integer lattice. We argue that statistically, shortest path rounding causes 1/6 to 1/4 the combinatorial damage and introduces about 1/3 the excess geometric error introduced by snap rounding. Shortest path rounding is also easy to use in practice, we have used it for industrial layout algorithms since 1992. As we describe later in the paper, these layout algorithms use a variety of geometric algorithms and have arbitrary cascading, even for a single layout problem.

The following section argues that exact methods are likely to have exponential cost no matter how one implements them. Section 1.2 compares shortest path rounding to other geometric rounding algorithms and other techniques for implementing geometric algorithms using rounded arithmetic. Section 1.3 gives an outline for the rest of the paper.

1.1 The Cost of Cascading

This section examines the cost of cascading geometric algorithms without rounding. The basic problem is that the number of bits in each output coordinate can be two or more times the number of bits in each input coordinate. Cascading causes this *bit-complexity* to grow exponentially with the number of operations. One might think that this worst case is difficult to attain or that there is some special way to represent derived coordinates that avoids exponential growth. Unfortunately, these are vain hopes. This section shows that *any* exact arithmetic calculation can be encoded as cascaded geometry, and it is easy to give an example of exact arithmetic with exponential growth in bit complexity. In short, eliminating rounding from the field of *computational geometry* is equivalent to eliminating rounding from the field of *numerical analysis*.

To simplify the analysis, we consider only *straight-edge constructions*: 1) join two points to generate a line and 2) intersect two lines to generate a point.[1] As previously stated, these are not the only way to generate new points; however, almost every geometric algorithm uses these two. Goodrich, Pollack, and Sturmfels [14] describe arrangements of points and lines whose realizations on the integer grid must have exponential bit-complexity. Their technique uses a geometric implementation of repeated squaring. In fact any arithmetic operation can be implemented given a straight-edge and a finite collection of reference points [2]:

Lemma 1.1 *Given the integer grid points (x, y), $|x|, |y| \in \{0, 1\}$, the following transformations can be done using* $\mathrm{O}(1)$ *straight-edge constructions:*

$$\begin{aligned} (a, b) &\rightarrow (a, 0), (b, 0); && (1)\\ (a, 0), (b, 0) &\rightarrow (a + b, 0), (a - b, 0), (a \cdot b, 0), (a/b, 0), (a, b). && (2) \end{aligned}$$

Lemma 1.1 implies that an exact geometric system with cascading can perform any exact calculation on integers or exact rational numbers. The following lemma shows that exact arithmetic necessarily has exponential growth in bit-complexity.

Lemma 1.2 *Given the set $\{1\}$ and the operations $+$ and $\cdot$, the set of all integers in the range from 1 to 2^{2^k} can be constructed in $2k$ generations, for all $k > 0$.*

Proof: Clearly $1 = 1$, $2 = 1 + 1$, $3 = (1 + 1) + 1$, $4 = (1 + 1) \cdot (1 + 1)$, and so the lemma holds for $k = 1$. Assume the lemma holds for $k = l - 1$. Any number in the range 1 to 2^{2^l} can be written as,

$$\left(2^{2^{l-1}} \cdot m\right) + n,$$

where m and n are in the range 1 to $2^{2^{l-1}}$. By the assumption, each number in this expression is in generation $2(l - 1)$. Performing the multiplication and addition only adds two generations, and therefore the number is in generation $2l$. This proves the lemma for $k = l$ and finishes the induction. ∎

[1] These are classical straight-edge and compass constructions–but without the compass!

Keep in mind that $2k$ is the height of the construction tree, not the number of nodes, and therefore the number of operations might be much larger than the number of generations. However, repeated squaring can generate large numbers using very few operations. In particular, repeated squaring can generate 2^{2^k} in $k+1$ generations and $k+1$ operations. Every realization of the corresponding geometric structure (Lemma 1.1) in the integer grid requires bit-complexity at least $\lg 2^{2^k} = 2^k$. This is essential the result of Goodrich *et al.* This section has shown that this result is not an isolated case. Any large number that can arise in a non-geometric calculation can also arise in a geometric calculation.

1.2 Related Work

The previous section establishes that rounding is unavoidable in general. This section discusses methods for implementing algorithms of computational geometry in the presence of rounding. Section 1.2.1 discusses four approaches, including geometric rounding. Section 1.2.2 discusses techniques for geometric rounding.

1.2.1 Robust Geometry

Many geometric systems achieve a high level of robustness though the application of tolerances and heuristics, usual over a long period of testing and use in practice. However, these systems are not provably correct. There are essentially four theoretically sound approaches to including rounding into geometric algorithms. These approaches are generally called "robust geometry."

Data Normalization [32, 24]: Carry out computations using rounded floating point arithmetic. Alter the geometry and combinatorial structure to eliminate ill-conditioned computations. For example, if vertex c is too close to edge ab to determine on which side it lies, then "crack" ab into ac and cb. **Advantages**: Uses hardware floating point and generates explicit geometric structures. **Disadvantages**: Requires modification of geometric algorithms and has unbounded geometrical and combinatorial error.

Consistent (Stable) Computation [24, 28, 22, 1, 4, 9, 10, 12, 20, 22, 38]: Use hardware floating point and make consistent symbolic decisions in the case of an ambiguous numerical tests. **Advantages**: uses hardware floating point and sometimes has better bounds on error than data normalization. **Disadvantages**: Decisions have implicit rather than explicit realizations which makes geometric reasoning difficult. Only works with specific algorithms, and sometimes requires subtle changes. Proving existence of consistent, numerical accurate, realizations is tedious and difficult.

Combinatorics-Preserving Geometric Rounding: Use exact arithmetic with any algorithm. Round output geometric structure to lower precision without changing combinatorial structure. In the example with ab and c, round a, b, and c in such a way that ab and c move apart. **Advantages**: can use any algorithm and has no combinatorial error. **Disadvantages**: no one knows how to do it. Might have large geometric error. Some versions are known to be NP-hard [31].

Geometric Rounding [16, 25, 26, 27, 35, 18]: Use exact arithmetic with any algorithm. Round output to lower precision, changing combinatorial structure if necessary. For example, if c rounds to the other side of ab, spit ab into ac and cb. **Advantages**: works with any algorithm on roundable objects and has bounded error. **Disadvantages**: changes combinatorial structure.

1.2.2 Geometric Rounding

There are currently five techniques for geometric rounding.

Green-Yao [16]: This was the earliest geometric rounding technique, and it rounds line segments to the integer lattice. Treat the line segment ab as a flexible elastic string. Pull a and b to the nearest lattice points $\rho(a)$ and $\rho(b)$. For every other vertex v, if the segment $v\rho(v)$ intersects ab, pull the segment to $\rho(v)$. Do not allow the rest of string to move past any integer lattice point. **Advantages**: bounded error, good for graphics applications, might be generalizable to other lattices. **Disadvantages**: introduces $\Omega(n \log |ab|)$ "excess" lattice points onto the segment, where n is the number of vertices to which the segment it pulled.

Snap Rounding [15, 19]: Various researchers have discovered this technique for rounding line segments to the integer grid.[2] Each vertex rounds to the nearest lattice point. To round ab, determine rounding cells of rounded vertices that intersect ab. Replace ab by the polygonal curve that visits the lattice points of these rounding cells. **Advantages**: very simple, bounded error, does not introduce any extra lattice points. **Disadvantages**: does not appear to generalize to other lattices. Introduces more vertices on polygonal curve than necessary.

Shortest Path Rounding [26, 25, 35]: This technique replaces ab by the shortest path that keeps all other rounded vertices to the "correct" side. This paper describes a somewhat more general version than we have previously described in conference papers (Section 4.5) and used in practice. **Advantages**: can handle any lattice with connected rounding cells (Section 3.1). Introduces minimum geometric and combinatorial error. Simple to use in practice. **Disadvantages**: not clear how to generalize to three dimensions.

CSG Rounding [37]: Given a CSG (constructive solid geometry) representation of an object, round CSG primitives and then reconstruct the tree. **Advantages**: works in three dimensions. **Disadvantages**: suitable for set operations and transformations, not decompositions, convex hull, or Minkowski sum. Topology might be unpleasantly altered.

Manifold Rounding [11]: Given a manifold representation of a polyhedral solid, round equations of faces. If rounded solid is self-intersecting, retain only the "unburied" portion of the boundary. **Advantages**: three dimensional, intuitive topology change, bounded error. **Disadvantages**: suitable for set operations and transformations, not decomposition, convex hull, or Minkowski sum.

1.3 Outline

Section 2 describes different possible numerical representations, algorithms for exact arithmetic, and lattices including the *floating point lattice* and the *homogeneous coordinate* lattice. These are perhaps more useful in practice than the integer lattice. Section 3 gives a rigorous definition of geometric rounding in general and shortest path rounding in particular. Section 4 gives three algorithms for shortest path rounding, including the specialized version we use most often in practice. Section 5 proves the correctness of shortest path rounding and the algorithms for constructing it. Section 6 discusses our experiences in practice. It sketches an algorithm for layout of polygons which has unbounded cascading, and it shows how geometric rounding has made it possible for us

[2] At the 1989 Canadian Conference on Computational Geometry, this author discussed it as a alternative to shortest path rounding (on the integer grid) which introduced more vertices but was somewhat simpler to implement (although this discussion did not appear in the abstract [26]). In the software we developed in 1992 for set operations on polygons, we put in a compiler directive to switch to snap rounding to check to see if there was a significant difference in running time. There is no significant difference.

to create and license to industry a working implementation of this algorithm. Finally, Section 7 shows that shortest path rounding introduces less combinatorial and geometric error than other rounding methods on integer lattices, and it discusses why it is useful

2 Numerical Issues in Computational Geometry

This section discusses numerical issues in computational geometry. Section 2.1 discusses representations for single or multiple precision integer, rational, and floating point numbers and techniques for performing exact arithmetic in these domains. Section 2.2 describes several common representations for geometric points. Section 2.3 shows how geometric primitives can be reduced to polynomial expressions on the point coordinates. Finally, Section 2.4 puts these techniques together to implement geometric algorithms. These implementations 1) perform arithmetic almost as fast as using hardware floating point for all operations, 2) calculate the correct combinatorial structure corresponding to exact arithmetic on the input coordinates, and 3) generate accurate output coordinates rounded to floating point, to lower precision integers, or to lower precision rational numbers.[3] Shortest path rounding, as described in subsequent sections, then "sensibly" alters the geometry and combinatorial structure to make it consistent with these rounded coordinates.

2.1 Numerical Representations

Computer arithmetic is discrete, and essentially the only type of number which can be represented on a computer is an integer. All modern computers also support a floating point representation with an integer mantissa and exponent. Most computers have hardware devoted for floating point operations which make floating point operations faster than integer operations.

The most common hardware integer type is 31 bits plus a sign bit[4], and the most common floating point ("double precision") has a 53-bit mantissa and 10-bit exponent plus sign bits for each. It is often convenient to use the floating point data type to store integers with up to 53 bits.

A rational number can be represented as an ordered pair of integers. There are a number of representations for integers or rational numbers with a more than 53 bits. Traditionally, a "BIGNUM" is a list of integers. Fortune and van Wyk [13] survey some of these representation. Shewchuk [36] has more recently presented a simple, elegant multiple precision floating point representation. Each number is represented as a list of hardware double precision values whose sum is the number being represented. This representation can handle any number with 1023 bits both to the right and to the left of the "binary point".

Both Fortune and van Wyk and Shewchuk show that geometric primitives of bounded depth, such as the circulation test (Section 2.3) can be carried out to arbitrary precision in practice using only a little more time than that required for 53 bits of precision. The trick is that they quickly calculate the first 53 significant bits and only calculate more bits if necessary. In practical applications, calculation of primitives rarely require multiple precision. In other words, it rarely happens that a vertex c is so close to an edge ab that more than 53 bits of precision are required to determine on which side of ab it lies.

[3] In order for the implementation to be almost as fast a native floating point, the output coordinates lose as much accuracy as predicted by numerical analysis. Greater accuracy has greater computational cost but still modest if Shewchuk's techniques are used (see below).

[4] The actual representation is 2's complement, but this does change the available precision.

2.2 Coordinate Representations

A geometric system can use either points or lines as primitive elements. In this paper, we will consider only representations of points. Geometric points can be represented using an ordered pair (x, y) in Cartesian coordinates, using *homogeneous coordinates* (W, X, Y), or using *polar coordinates* (r, θ), where

$$x = \frac{X}{W} = r\cos\theta \qquad \text{and} \qquad y = \frac{Y}{W} = r\sin\theta.$$

In the case of polar coordinates, one can avoid the transcendental functions sin and cos by using a *rational parameterization* of the unit circle,

$$\omega(t) = \left(\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}\right).$$

For $-1 < t < 1$, $\omega(t)$ nearly uniformly ($|\omega'(t)| \leq 2$) covers the unit circle to the right of the y-axis. The left half is parameterized by reflection. Canny, Donald, and Ressler [3] describe how to generate a dense set of rational values for t to uniformly cover the unit circle to any desired degree of precision.

Different numerical representations can be used with the geometric representations. The coordinates of (x, y) can be integers, rationals (ordered pairs of integers), or floating points. The coordinates of (W, X, Y) can be integers or floating points. Rationals are not necessary since one can always "clear the denominators" of the homogeneous coordinates. In the rational parameterization of polar coordinates, r can be integer, rational, or floating point, and t can rational or floating point.

2.3 Geometric Primitives

Only three numerical geometric primitives are required to implement most algorithms on points, lines, and line segments: circulation, point sum, and segment intersection. It is most convenient to use homogeneous coordinates because these allow the primitives to be expressed using only addition, subtraction, and multiplication of integers or floating point numbers. These operations can be carried out exactly and quickly using the techniques cited in Section 2.1.

Converting Cartesian coordinates to homogeneous is trivial $(x, y) \to (1, x, y)$. If x and y are rational, then one can clear the denominator,

$$\left(\frac{x_n}{x_d}, \frac{y_n}{y_d}\right) \to (x_d y_d, x_n y_d, y_n x_d).$$

Assuming, r and t have rational values, polar coordinates can also be easily converted to homogeneous coordinates,

$$\left(\frac{r_n}{r_d}, \frac{t_n}{t_d}\right) \to (r_d(t_d^2 + t_n^2), r_n(t_d^2 - t_n^2), 2r_n t_d t_n).$$

All division, rounding, and converting back to Cartesian or polar coordinates can be postponed until after the algorithm has executed. This is discussed in the next section.

Circulation: We assume all negative W coordinates are made positive: $(W, X, Y) \to (-W, -X, -Y)$ if $W < 0$. The circulation $[a, b, c]$ of points a, b, and c is a 3x3 determinant,

$$[a, b, c] = \begin{vmatrix} a_W & a_X & a_Y \\ b_W & b_X & b_Y \\ c_W & c_X & c_Y \end{vmatrix}.$$

Triangle abc winds counterclockwise if $[a, b, c] > 0$ and winds clockwise if $[a, b, c] < 0$. If $[a, b, c] = 0$, then the three points are collinear.

Point Sum: Adding two homogeneous points by coordinate,

$$a + b = (a_W + b_W, a_X + b_X, a_Y + b_Y),$$

yields a point on the line ab, but it is not the same as the sum in Cartesian coordinates. In homogeneous coordinates, we denote the Cartesian sum by $\oplus$,

$$a \oplus b = \left(\frac{a_X}{a_W}, \frac{a_Y}{a_W}\right) + \left(\frac{b_X}{b_W}, \frac{b_Y}{b_W}\right) = (a_W b_W, a_X b_W + a_W b_X, a_Y b_W + a_W b_Y).$$

Line Intersection: In homogeneous coordinates, the intersection point of lines ab and cd is

$$[b, c, d]a - [a, c, d]b,$$

where scalar multiplication and vector addition (subtraction) is carried out in the usual fashion. This intersection point lies on both *edges* ab and cd if and only if acd and bcd have opposite circulation and cab and dab have opposite circulation.

Other primitives can be calculated from these three. For example, if $[a, b, c] = 0$, then the three points are collinear. To determine if b is between a and c, select a point d not on the line and check if abd and cbd have opposite circulations. For most geometric algorithms, the depth of the calculation is bounded. For example, computing an arrangement of line segments requires the following primitive: what is the circulation $[h, e, f]$ where h is the intersection of ab and cd? This primitive is a concatenation of the line intersection and circulation primitive,

$$[h, e, f] = [a, e, f][b, c, d] - [a, c, d][b, e, f].$$

Note that any primitive can be reduced to a polynomial expression on the input coordinates.

2.4 Rounding Coordinate Representations

In fact, in order to have an efficient numerical implementation of a geometric algorithm, all primitives must be reduced to polynomial expressions on the inputs. In this way, as Fortune and van Wyk and Shewchuk have demonstrated, it is possible to compute the correct combinatorial structure of the output without explicitly computing the coordinates of its points and lines. Shewchuk's method allows one to compute the value of coordinates to any degree of accuracy.

Hence, using Shewchuk's numerical methods and the conversions and primitives defined above, it is possible to 1) start with floating point (or integer) input coordinates, 2) compute the exact combinatorial output of any geometric algorithm on those inputs 3) calculate the nearest floating point approximations to the output coordinates. Applying shortest path rounding yields a combinatorial structure consistent with these approximations.

There are also methods for finding good integer approximations, if that is desired. As previously mentioned, Canny *et al.* [3] give a method for finding accurate, low precision rational coordinates on the unit circle. Part of their method involves using *continued fractions* to find good rational approximations to real numbers. Hence, their techniques can be used to round to exact rational polar coordinates or exact rational Cartesian coordinates. In other work [34], we have shown how to use *basis reduction* to find good rational orthonormal (rotation) matrices in three dimensions. Part of this work involves finding good integer approximations to homogeneous coordinates in three dimensions. The same technique also works in two dimensions to generate accurate lower precision integer approximations to homogeneous coordinates.

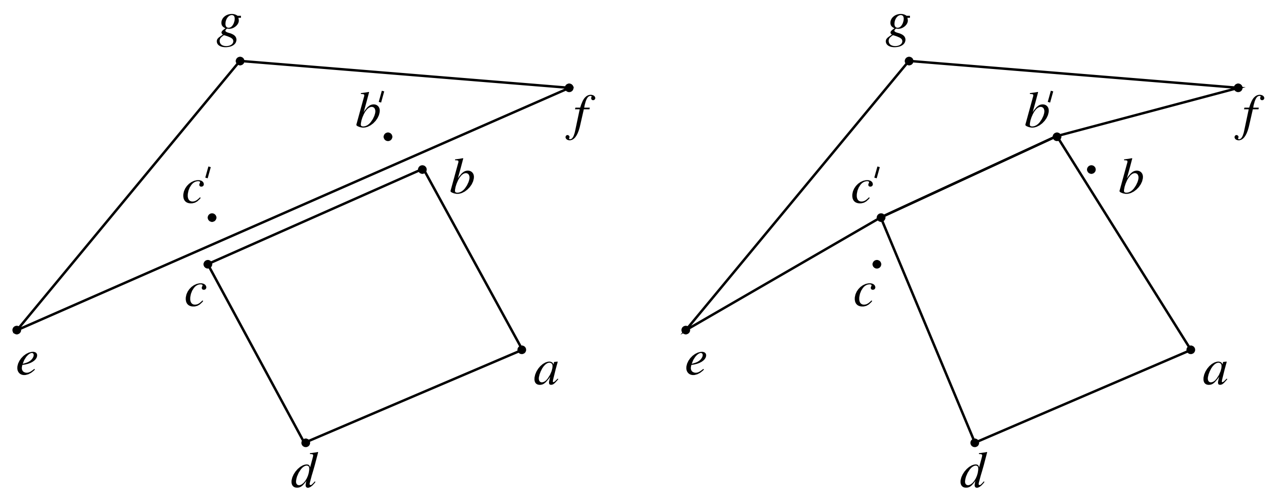

Figure 1: Shortest Path Geometric Rounding

3 Definition of Shortest Path Rounding

This section gives a mathematical definition for shortest path rounding. It will help the reader to keep in mind a simple example shown in Figure 1. Vertices b and c round to lattice points b' and c', and all other vertices are already at lattice points. Vertices b and c are not at lattice points, perhaps because they have arisen from the intersection of two line segments. This can happen if polygon $abcd$ resulted from the intersection of two other polygonal regions. As b and c are rounded to lattice points, they must not "cross into" triangle efg. Under shortest path rounding, edge ef becomes a polygonal curve $ec'b'f$.

To give a rigorous mathematical definition of what is happening in this figure, this section first defines lattices (sets of rounding sites), then straight line embeddings (the thing that is rounded), geometric rounding, and shortest path geometric rounding. It states the theorem that shortest path roundings *are* geometric roundings, but the proof is postponed until Section 5. An illustration is given of a bizarre case which might arise when rounding in polar coordinates. This case demonstrates the necessity of the careful definitions and proofs.

3.1 Lattices

Geometric rounding rounds vertices to a "lower precision lattice". Precisely speaking, there is a set of sites S in the plane and a cell $\mathrm{CELL}(s)$ associated with each site $s \in S$ such that each point p in the plane lies in exactly one cell. The rounding function ρ takes p to the unique site $s = \rho(p) \in S$ such that $p \in \mathrm{CELL}(s)$.

Shortest path rounding demands only that the cells be *connected*. However, if the rounding cell $\mathrm{CELL}(\rho(v))$ of some vertex v is not *simply connected*, then there are topologically distinct paths along which one can round v to $\rho(v)$. For example, v can travel to the left or to the right of a "hole" in $\mathrm{CELL}(\rho(v))$ on the way to $\rho(v)$ and even perhaps wind several times around the hole. This path can be represented as a *rounding curve* γ_v that takes a point v to its rounding site $\rho(v)$: $\gamma_v(0) = v$, $\gamma_v(1) = \rho(v)$, and $\gamma_v(t) \in \mathrm{CELL}(\rho(v))$, $0 \leq t \leq 1$. It will be proved that the output of shortest path rounding depends only on the topological *family* of the curve: two curves belong to the same family if one is a topological deformation of the other. If the rounding cell is simply connected, the family and hence the output of shortest path rounding is unique. If the rounding cell is connected but not simply connected, one way to ensure a unique topological family of curves is to introduce cuts in the rounding cell to eliminate holes.

Consider, for example, rounding in polar coordinates. One might choose to round all coordinates (r, θ) with $0.5 < r < 1.5$ and $-180 < \theta \leq 180$ to the point $(1, 0)$. The region $0.5 < r < 1.5$ is "cut" at $\theta = -180 = 180$. For $0 < \theta \leq 180$, θ is rounded downward to zero, and for $-180 < \theta < 0$, θ is rounded upwards to zero.[5]

3.2 Embeddings

As Figure 1 illustrates, we are not merely rounding points to a lattice. We are also rounding line segments. Precisely speaking, geometric rounding rounds a *straight line embedding* of a planar graph $G = \langle V, E \rangle$. In the figure, graph G has vertices $V = \{a, b, c, d, e, f, g\}$ and edges $E = \{ab, bc, cd, da, ef, fg, ge\}$.

In an embedding, the vertices are distinct and the edges meet only at their endpoints. In a straight line embedding, the edges must be line segments. In a general planar embedding, they can be curves. Edges in a straight line embedding can meet in a "V", but not a "T" or an "X", and they cannot share a common subsegment.

The graph G has undirected edges. To represent some geometric objects, it may be necessary to have directed edges and even multiple copies of edges. This level of combinatorial detail can be built "on top" of the undirected graph G. For example, one might choose to assign a direction to each edge of the unrounded polygon in Figure 1 so that the boundary of each polygon winds around counterclockwise. If the rounded polygon is thought of as a straight line embedding, then edge $b'c'$ is undirected and appears only once. If the figure is thought of as two abutting polygons $ab'c'd$ and $ec'b'fg$, then $b'c'$ appears twice with opposite orientation.

3.3 Geometric Rounding

As Figure 1 illustrates, geometric rounding does not preserve straight lines. It does not preserve the graph either! However, it does yield a new straight line embedding whose vertices lie at sites in S. The best way to define geometric rounding is as a *limit* of topologically equivalent embeddings.

Imagine moving b and c towards $b' = \rho(b)$ and $c' = \rho(c)$. At the same time, line segment ef deforms into $ec'b'f$. Imagine further that the deforming segment always stays a little bit ahead of the two moving vertices until the very last "moment". Until the final "click", the deforming embedding remains topologically equivalent to the original embedding. The limit is not topologically equivalent, but it is arbitrarily close to embeddings which are equivalent.

The limit must have the property that all vertices are at lattice sites and all edges become polygonal curves with vertices at the lattice sites of vertices. Each vertex v stays within its rounding cell as it moves to $\rho(v)$. We believe that this definition captures the intuitive notion of a "good rounding". Here it is more formally.

[5] An alternative to introducing cuts into a rounding cell with holes is to explicitly specify a topological family ϕ_v to which γ_v must belong. One might (bizarrely) insist that the rounding path for (r, θ) winds around the origin five times before reaching $(1, 0)$. Shortest path rounding can handle this choice, although one would probably not want to do this in practice!

Definition 3.1 *A* **deformation** *of the Euclidean plane is a continuous function,*

$$\pi : [0,1] \times \mathbf{E}^2 \to \mathbf{E}^2,$$

such that for any fixed $t \in [0,1)$, $\pi_t(p) = \pi(t,p)$ *is a bijection.*

Note that $\pi(1,p)$ is *not* a bijection: vertices may collide and edges may come in contact at the "very last moment" ($t=1$). However, the shape of the plane at $t=1$ is clearly the limit of a series of bijections.

Definition 3.2 *A* **geometric rounding** *of a straight line embedding* $G = \langle V, E\rangle$ *to a lattice* S *is a deformation* π *of the plane with the following properties:*

1. *for each* $v \in V$, $\pi(t,v) \in \mathrm{CELL}(\rho(v))$ *for* $t \in [0,1]$;
2. *for each* $uv \in E$, $\pi(1,\cdot)$ *deforms the segment* uv *into a polygonal path which has vertices only at lattice points of vertices in* V.

3.4 Shortest Path Rounding

Intuitively, the curve $ec'b'f$ is the shortest path from e to f that does not have $b' = \rho(b)$ and $c' = \rho(c)$ to the "wrong side." Unfortunately, line segments do not truly divide the plane into two sides, and so this intuitive definition is not mathematically sound. However, now that we have a proper mathematical definition of geometric rounding, it is possible to finally define shortest path rounding.

Definition 3.3 *A* **shortest path rounding** *of an edge* $uv \in E$ *is a geometric rounding, with respect to the subgraph* $G_{uv} = \langle V, \{uv\}\rangle$ *of* G, *which results in a shortest possible polygonal path (under* $\pi(1,\cdot)$*) for segment* uv.

In other words, just round each edge as if there are no other edges to worry about. The central theorem of shortest path rounding is that there exists a valid geometric rounding for G which takes each uv to the same shortest path. Also, the paths are proved to be unique. Since the paths are the only result one explicitly "sees" (π is implicit), the resulting rounding is therefore called **the shortest path rounding** of G.

Theorem 3.1
1. The path resulting from a shortest path rounding of an edge uv *is independent of the choice of rounding curve* $\gamma_w \subset \mathrm{CELL}(\rho(w))$ *for each* $w \in V$.[6]
2. The union of shortest path roundings of the individual edges of a straight line embedding of a graph G *is a geometric rounding of* G.

The proof of Theorem 3.1 is postponed until Section 5. Section 4 first gives several algorithms for computing shortest path roundings. These algorithms actually facilitate the proof because one of them clearly satisfies Part 1 and the other clearly satisfies Part 2. Proving the correctness of the theorem and the algorithms is simply a matter of showing that the algorithms generate the same output!

[6] Assuming $\mathrm{CELL}(\rho(w))$ is simply connected or some way is chosen to contrain the rounding curves to a single topological family (see Section 3.1).

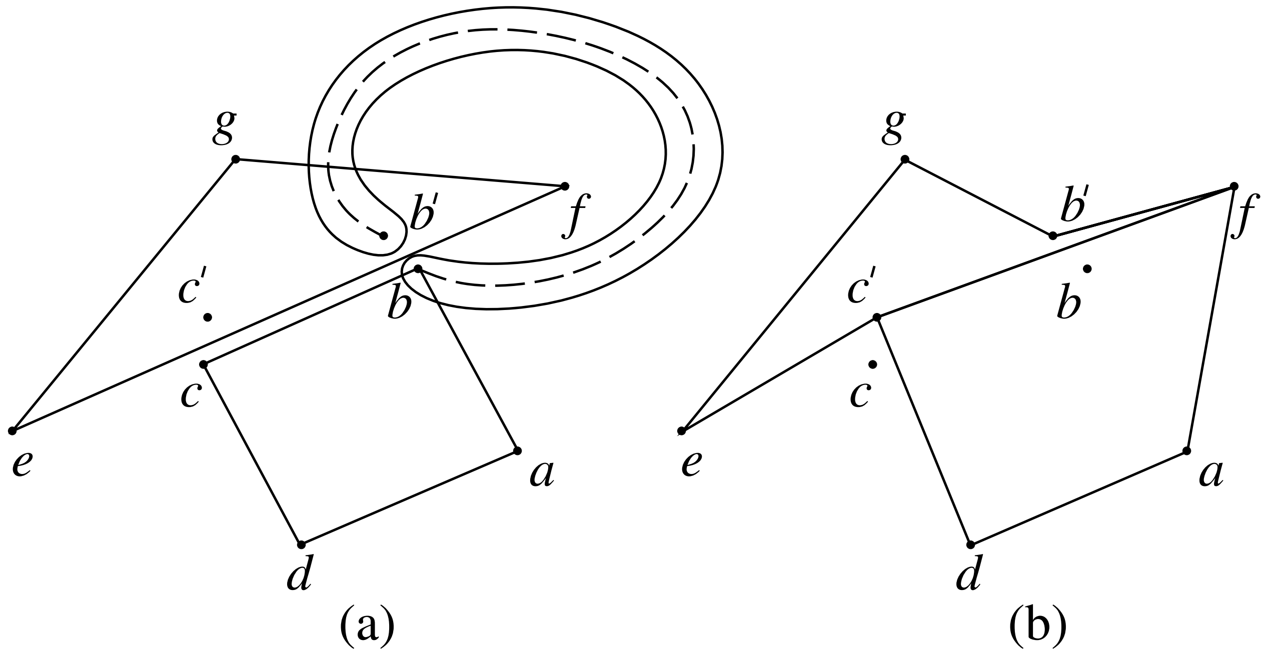

Figure 2: Shortest Path Geometric Rounding: different rounding cell for $b' = \rho(b)$. (Others remain the same.)

3.5 An Illustration

Before seeing the algorithms, it might help the reader to examine how the shortest path rounding depends on the shape of the rounding cell CELL($\rho(v)$), even if v and $\rho(v)$ are not changed.

Figure 2(a) illustrates a crescent-shaped rounding cell for b' and a path from b to b' within this cell. If we use this path, the shortest path rounding is as shown in Figure 2(b). Edge ef becomes $ec'f$, fg becomes $fb'g$, ab becomes afb', and bc becomes $b'fc$.

Why is this rounding so different? The rounding in Figure 1 uses a rounding cell that includes the straight line segment bb'. Most rounding methods have convex rounding cells, and therefore CELL(b') would contain this segment. Assuming f does not round to b', it follows that $f \notin$ CELL(b'). A simply connected rounding cell could not contain both bb' and the curve from b to b' depicted Figure 2(a). If the cell contained both curves, it would have had a hole, and we would have cut it (Section 3.1). Examples such as this one can arise when one rounds in polar coordinates. Of course, the illustration greatly exaggerates the effect, but the principle is the same.

4 Algorithms for Shortest Path Rounding

This section presents three algorithms for shortest path rounding. For reasons that will become apparent later, we refer to these as the *locally shortest path* (*local path*) algorithm, the *globally shortest path* (*global path*) algorithm, and the *monotone shortest path* (*monotone path*) algorithm. The local and global path algorithms can handle arbitrary lattices. The monotone path algorithm is a special case of the global path algorithm, and it can only handle lattices $S = S_x \times S_y$ which are the Cartesian product of two one-dimensional lattices. Rounding in x and y are done independently.

The monotone path algorithm is very simple to implement, works for many useful lattices, and it is the one we have been using in practice for industrial applications since 1992.

4.1 Locally Shortest Path Algorithm

The locally shortest path (local path) algorithm is best described using a physical analogy. Imagine the edges of the embedding to be elastic and flexible yet impenetrable. Vertices b and c move towards edge ef, and when they meet it, the edge starts to deflect and stretch without letting them through. When b and c reach b' and c', edge ef has become the path $ec'b'f$.

4.1.1 Representation

The representation of a deformed edge is simply the list of vertices which it touches. Edge ef starts out as e, f. When b hits it (assume b hits it before c), it becomes e, b, f. When c hits it, it becomes e, c, b, f. An additional tag is added to each vertex to indicate whether it is "pushing on the edge" from the right or from the left.

4.1.2 Special Times

For each vertex $v \in V$, the algorithm selects a rounding curve (Section 3.1) $\gamma_v(t)$, $0 \leq t \leq 1$. Since the rounding curve, by definition, must stay within the rounding cell, the output of the algorithm will depend on the rounding cell, as is appropriate (see Figures 1 and 2). The state of the system corresponds to a value of t plus the list of vertices for each deformed edge. Only certain "special" values of t need to be examined, and the algorithm visits these in increasing order.

A special time of t occurs whenever $\gamma_u(t)$, $\gamma_v(t)$, and $\gamma_w(t)$ are collinear, for some $u, v, w \in V$. Changes in the representation of the deformed edges can only occur at special times. A special time is a root of the equation,

$$[\gamma_u(t), \gamma_v(t), \gamma_w(t)] = 0, \tag{3}$$

where $[u, v, w]$ is the circulation (Section 2.3),

$$[u, v, w] = \begin{vmatrix} 1 & u_x & u_y \\ 1 & v_x & v_y \\ 1 & w_x & w_y \end{vmatrix}.$$

For convex rounding cells, $\gamma_v(t) = (1-t)v + t\rho(v)$, $v \in V$, and Equation 3 is quadratic in t. For rounding in polar coordinates, one might choose to round r for $0 < t < 0.5$ and round θ for $0.5 < t < 1$. For $t < 0.5$, the rounding curves are line segments as in the convex case. For $t > 0.5$, the rounding curves are circular arcs. Assuming the algorithm uses the rational parameterization of a circle, the resulting equation is of degree six. In general, if $\gamma_v(t)$ is polygonal (piecewise linear), then the values of t at which $\gamma_v(t)$ bends must also be added to the set of special times. Piecewise polygonal or piecewise rational curves can be handled similarly. This covers the types of curves which might be used in any imaginable application.

4.1.3 Updating the Deformed Edge

At each special time t_i, the representation of an edge can change. There are four cases to consider, but there is really only one case, and the other three result from reversing time, switching left and right, or both. The two "forward time" cases have the following conditions:

- u, v, and w are collinear at time t_i and v lies between u and w;
- u and w are consecutive elements of the list of edge ab;
- the circulation $[u, v, w]$ (Equation 3) is increasing (decreasing) in some neighborhood for $t > t_i$.

In these forward time cases, the algorithm adds v to the list between u and w and tags v as "to the left" ("to the right") of edge ab. The two reverse time cases are the reverse of these. If v lies on uw and if v is tagged "left" ("right") and if the circulation is decreasing (increasing) in some neighborhood $t > t_i$, then v is removed from the list of ab and its tag is also removed.

The local path algorithm visits each special time in increasing order and makes the appropriate update to the deformed edges at each special time. When it reaches $t = 1$, the deformed edges will be the shortest path rounding. We postpone the proof of correctness until Section 5.

4.2 Globally Shortest Path Algorithm

The *globally shortest path* (*global path*) algorithm determines the topology that a path should have and then directly computes it. Before the embedding of G is rounded, each edge $uv \in E$ is a straight line segment. After the vertices of G are rounded to sites in S, the path corresponding to edge uv must "go past" the rounded vertices of G according to the same topology. Two curves from $\rho(u)$ to $\rho(v)$ have the same topology (with respect to V) if one can be continuously deformed into the other without passing through another rounded vertex $\rho(w)$, $w \in V$. The global path algorithm computes the correct topology for an edge path and then constructs the shortest path with that topology. Actually, the shortest path is allowed to "touch" lattice points, and so it might have a different topology, but it is a limit of curves with the correct topology. For example, the path $ec'b'f$ does not have the same topology as ef because to "touches" b' and c', but it is arbitrarily close to paths which do have the same topology as ef.

The following three sections describe how the global path algorithm 1) represents topologies, 2) calculates the topology of a path, and 3) constructs the shortest path which "satisfies" the topology (which is the limit of paths with the correct topology).

4.2.1 Representation

The standard method to represent a topology is through the use of simplicial complices. In this case, the simplicial complex can be any triangulation of the set $\{\rho(v) \mid v \in V\}$ of rounded vertices. However, if more than one vertex in V rounds to the same site, then that site must be replicated a like number of times. The cluster of replicated sites is assigned a degenerate triangulation.

To represent the topology of a path from site $\rho(u)$ to site $\rho(v)$ is to list the triangles through which the path passes. If two different paths have the same list of triangles, then it is clear they have the same topology. Unfortunately, two paths with the same topology may pass through a different list of triangles. However, all topologically equivalent paths can be reduced to a unique *minimal list* path through the application of the following rules:

Rule 1	$\cdots T_i T_j T_i \cdots$	$\rightarrow$	$\cdots T_i \cdots,$	
Rule 2	$T_j T_i \cdots$	$\rightarrow$	$T_i \cdots,$	where T_i and T_j have $\rho(u)$ as a vertex,
Rule 3	$\cdots T_i T_j$	$\rightarrow$	$\cdots T_i,$	where T_i and T_j have $\rho(v)$ as a vertex.

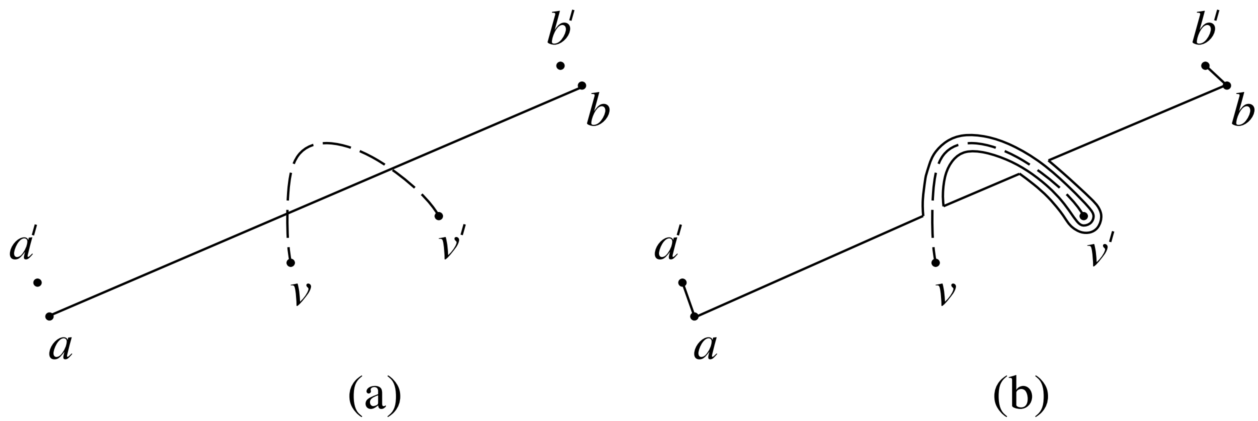

Figure 3: Computing topology for path from $a' = \rho(a)$ to $b' = \rho(b)$.

Rule 1 applies if the curve loops into triangle T_j from T_i and immediately out again. This loop can be pulled out of T_j. Rule 2 applies if the curve leaves $\rho(u)$ and passes through several triangles neighboring $\rho(u)$. All but one of these can be removed. Rule 3 is analogous to Rule 2.

This notion of a canonical (minimal list) representation of a topological path is a standard technique. It is clear that the canonical list can be generated from any list in linear time.

4.3 Computing Topologies

Suppose that a rounds to $a' = \rho(a)$ and b rounds to $b' = \rho(b)$ where $a, b \in V$ and $a', b' \in S$. This section shows how the global path algorithm computes the topology of the path from a' to b'. The idea is to generate at least one path with the correct topology, calculate its list of triangles, and then minimize that list using the rules of the previous section.

As usual, we assume that for each $v \in V$, we have chosen a rounding curve $\gamma_v(t) \subset \mathrm{CELL}(\rho(v))$ that v follows to $\rho(v)$. Given these curves, here is how the algorithm constructs a path from a to b. It starts at a', travels back along the curve γ_a to a and starts along the line segment ab. Every time it hits a rounding curve γ_v, it detours around it in the most "lazy" manner: it follows γ_v to $\rho(v)$, winds around $\rho(v)$, and then travels back along the other side of γ_v back to the line segment ab. It continues to follow ab and, if necessary, detour around rounding curves. When it reaches b, it follows γ_b to $b' = \rho(b)$. Figure 3(a) illustrates a vertex v whose rounding path crosses ab twice before reaching $v' = \rho(v)$. Figure 3(b) illustrates the resulting path from a' to b'.

The global path algorithm must compute intersections between line segments and rounding curves and must be able to compute the triangle lists for these rounding curves. Section 5 proves that the topology is independent of the choice of $\gamma_v(t) \subset \mathrm{CELL}(\rho(v))$, and therefore one can choose the curves that makes the computations most convenient. If the rounding cells are convex, as they are for most commonly used lattices, then one can choose the curve $\gamma_v(t) = (1-t)v + t\rho(v)$. The global path algorithm need only compute intersections of line segments. For rounding in polar coordinates, one can choose a curve consisting of a line segment plus a circular arc. In this case, the algorithm has to intersect line segments with line segments or arcs, requiring the solution of a quadratic equation.

Note that the local path algorithm involved roots of equations of higher degree–degree six in the case of rounding in polar coordinates–than the global path algorithm. Even though the global path algorithm is more complicated, this difference in degree might make it easier to implement in practice than the local path algorithm.

4.4 Computing Shortest Paths

The algorithm has computed a triangulation on the rounding sites of V, and it has computed a list of triangles for each edge path. For a given edge $uv \in E$, the path must start at $\rho(u)$, pass through triangles $T_1, T_2, T_3, \ldots, T_m$, and end at $\rho(v)$. The path must be the shortest which does so.

Fortunately, there already exists an algorithm in the literature. Guibas *et al.* [17] give an algorithm for computing the shortest path from one vertex to another inside a simple polygon. Given a triangulation of the polygon, the algorithm runs in linear time. This algorithm works perfectly fine on a list of neighboring triangles.

It should be emphasized that we are slightly extending the grasp of the existing shortest path algorithm. The polygon in Figure 4 does not appear to be a simple polygon. However, suppose we wish to compute the shortest path from u to v which passes through the list of triangles $2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$. The algorithm *can* handle this task. As far as it is concerned, the second appearances of triangles 3 and 4 as triangles 13 and 14 are different instantiations than the first appearances. The input is a simple polygon that happens to be drawn on a spiral staircase, and we are looking at it from above. The reason that the algorithm can handle this case is that it only looks at the local interactions of the triangles.

The list of triangles should almost always be a simple polygon anyway, but strange cases like this can arise when rounding in polar coordinates. The path joining $\rho(a)$ to $\rho(b)$ cannot intersect itself, but it can enter the same triangle more than once.

4.5 Monotone Shortest Path Algorithm

This section gives a complete global path algorithm for the case in which $S = S_x \times S_y$ is the Cartesian product of two one-dimensional lattices and in which rounding is done independently for each coordinate: $\rho(v) = (\rho_x(v_x), \rho_y(v_y))$, where ρ_x and ρ_y are the rounding functions for the one-dimensional lattices. For reasons given below, this is called the *monotone shortest path* (*monotone path*) algorithm. This algorithm is very simple and easy to implement. We have used it for industrial application software since 1992.

It is easy to see that a one-dimensional lattice has connected rounding cells if and only if the rounding function is monotone: if $x_1 < x_2$, then $\rho_x(x_1) \leq \rho_x(x_2)$. This is one reason for the name of the algorithm. The other reason is that the output paths are always monotone in x and y. Specifically, let $\sigma(t)$ be the arc-length parameterization of the rounded path connecting two rounded vertices $\rho(u)$ and $\rho(v)$, $u, v \in V$. It follows that the x and y coordinates of $\sigma(t)$ are both either non-increasing or non-decreasing functions of t.[7] The path is also monotone with respect to the segment $\rho(u)\rho(v)$: each point on the path has a unique perpendicular projection onto $\rho(u)\rho(v)$.

4.5.1 High Level Algorithm

The next section gives an algorithm for

[7] An equivalent definition is that every horizontal or vertical line intersects the path in either the empty set, a single point, or a single line segment.

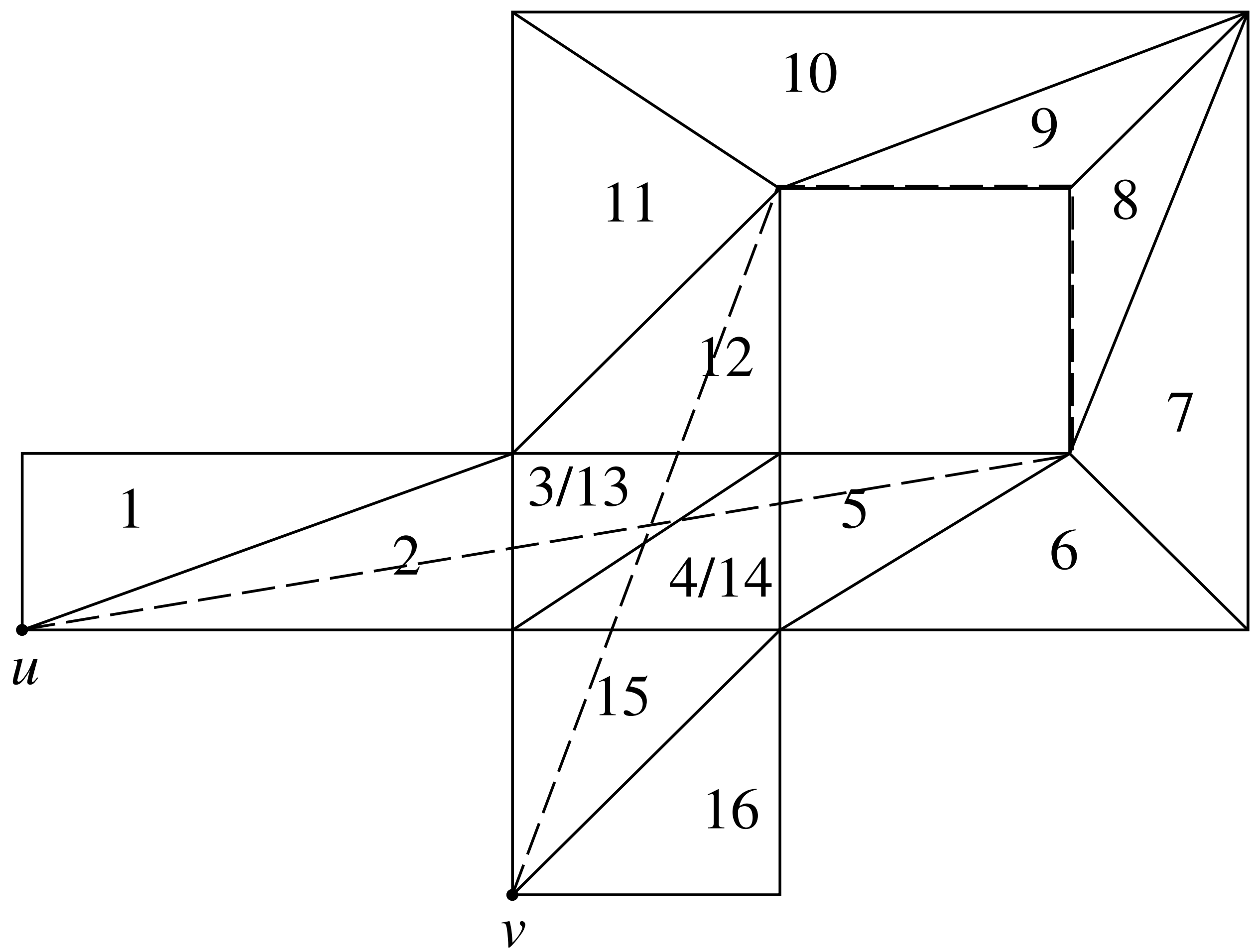

Figure 4: Shortest path from u to v through triangles 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15: dashed line.

FindShortest(A, B, P_1, P_2, ..., P_n)

which finds the shortest path from A to B which stays to the correct side of P_i, $i = 1, 2, 3, \ldots, n$. FindShortest requires that the P_i's project onto segment AB in monotone order:

$$A \cdot AB < P_1 \cdot AB \leq P_2 \cdot AB \leq P_3 \cdot AB \leq \cdots \leq P_n \cdot AB < B \cdot AB,$$

where $\cdot$ is the standard dot project and where $AB = B - A$, the vector from A to B. Each P_i is tagged LEFT or RIGHT. Points P_i and P_j cannot project to the same point on AB ($P_i \cdot AB = P_j \cdot AB$) unless $\text{TAG}(P_i) \neq \text{TAG}(P_j)$.

Given a straight line embedding of a graph $G = \langle V, E \rangle$, the monotone shortest path geometric rounding algorithm rounds an edge $ab \in E$ as follows. It sets $A = \rho(a)$ and $B = \rho(b)$. It takes the rounded vertices $\rho(v)$, $v \in V$, which project onto AB ($A \cdot AB < \rho(v) \cdot AB < B \cdot AB$) and sorts them in order of projection position $\rho(v) \cdot AB$. If two vertices $u, v \in V$ on the left of line ab ($[a, b, u], [a, b, v] > 0$) project to the same point ($\rho(u) \cdot AB = \rho(v) \cdot AB$), it discards the farther one: $\rho(u)$ is farther from AB than $\rho(v)$ if $|[A, B, \rho(u)]| > |[A, B, \rho(v)]|$. The algorithm similarly filters vertices on the right. It sets $P_1, P_2, P_3, \ldots, P_n$ equal to the sorted lattice sites (rounded vertices) and tags them LEFT or RIGHT according to the status before rounding. Finally, it calls FindShortest. The output is the shortest path rounding of edge ab.

Because rounding in each coordinate is monotonic, the monotone path algorithm only needs to consider vertices $v \in V$ inside the *bounding box* of edge ab: $\min(a_x, b_x) < v_x < \max(a_x, b_x)$ $\min(a_y, b_y) < v_y < \max(a_y, b_y)$. If v lies in the bounding box, then $\rho(v)$ will project onto $\rho(a)\rho(b)$. Also, it can eliminate vertices which lie farther than ϵ_{ab} from ab, where ϵ_{ab} is the maximum width or height of a lattice rounding cell intersecting the bounding box of ab. It is usually not difficult to calculate ϵ_{ab} or a good upper bound on it for a given choice of lattice. For example, if S_x and S_y are both the lattice of representable floating point numbers, then $\epsilon_{ab} = 2^{-\beta} \max(|a_x|, |a_y|, |b_x|, |b_y|)$, where β is the number of bits in the mantissa (53 on most computers).

Remember that that the purpose of geometric rounding is to deal the occasional near-singular cases that crash naively implemented geometric algorithms. In most circumstances, few if any vertices lie within ϵ_{ab} of any given edge ab. Furthermore, the cost of finding these vertices is modest and roughly proportional to the rate they occur. Usually, one has already constructed a trapezoidalization or some other search structure on the arrangement of line segments. Finding vertices which satisfy the epsilon test is simply a matter of detecting "flat" or "pinched" trapezoids.

4.5.2 Shortest Path Algorithm

This section gives an algorithm for FindShortest described in the previous section. This algorithm is a special case of Guibas *et al.* algorithm for shortest path in a simple polygon. The algorithm for FindShortest uses two subroutines, AddLeft and AddRight, which are defined first.

In the following, *Path*, *Left*, and *Right* are double-ended stacks. PushHead, PopHead, PushTail, and PopTail do the obvious things. *Path*.Head[0] is the current "head" of the stack. *Path*.Head[1] is the element one away from the "head" end of the stack. After FindShortest is executed, *Path* contains the desired shortest path. While the algorithm is executing, *Left* is the path that satisfies the constraints seen so far and "veers left" as much as possible. Similarly, *Right* is the path that "veers right" as much as possible. Whenever it is determined that *Left* and *Right* have a common prefix, that part is added to the end of *Path*.

```
AddLeft (P, Path, Left, Right)
    while Left.Size > 1
        A ← Left.Head[1]
        B ← Left.Head[0]
        if [A, B, P] ≤ 0
            Left.PopHead
        else
            break
    Left.PushHead (P)

    if Left.Size = 2 and Right.Size > 1
        while Right.Size > 1
            A ← Right.Tail[0]
            B ← Right.Tail[1]
            if [A, B, P] ≤ 0
                Right.PopTail
                Left.PopTail
                Left.PushTail (Right.Tail[0])
                Path.PushHead (Right.Tail[0])
            else
                break
        Right.PushHead (P)
end AddLeft
```

AddRight (P, *Path*, *Right*, *Left*) is analogous to "AddLeft" with the roles of Right and Left switched.

```
FindShortest (A, B, P_1, P_2, ..., P_n)
    Path.PushHead(A)
    Left.PushHead(A)
    Right.PushHead(A)

    for i ← 1 to n
        if TAG(P_i) = LEFT
            AddLeft (P_i, Path, Left, Right)
        else
            AddRight (P_i, Path, Left, Right)
    AddLeft (B)
    AddRight (B)
    return Path
end FindShortest
```

5 Proofs

The local path algorithm and the global path algorithm are two ways of computing shortest path roundings. They also provide two halves of the proof of Theorem 3.1, the central theorem of shortest path rounding. This section proves the correctness of these algorithms and proves the central theorem in three steps. First, it defines a *locally shortest path rounding* for an edge and proves that it is unique and equal to the shortest path rounding (Definition 3.3). Next it proves that the global path algorithm generates a shortest path rounding that satisfies Part 1 of of Theorem 3.1. Finally, it proves that the local path algorithm generates a locally shortest path rounding that satisfies Part 2 of Theorem 3.1. Taken together, these results prove the theorem.

5.1 Locally Shortest Path Rounding

Given a straight line embedding of a graph $G = \langle V, E \rangle$, let us suppose that we have a path from $\rho(a)$ to $\rho(b)$, $ab \in E$, that avoids all other rounded vertices $\rho(v)$, $v \in V$. Suppose we have triangulated the rounded vertices (Section 4.2.1), and the path passes through the (minimal) list of triangles $T_1, T_2, T_3, \ldots, T_n$. It is possible to parameterize the path $\sigma(t)$ so that $\sigma(0) = \rho(a)$, $\sigma(n+1) = \rho(b)$, and $\sigma(t) \in T_i$, $i-1 \leq t \leq i$. For $t = i$, $\sigma(t) \in T_i \cap T_{i+1}$, which means that curve crosses the edge common to T_i and T_{i+1}. Note that under this parameterization, $\sigma(t)$ might stay fixed for t in some intervals $[i-1, i]$.

To deal with the case that σ passes through $\rho(v)$, let us consider σ to be a curve,

$$(t, \sigma(t)) \subset [0, n-1] \times \mathbf{E}^2.$$

This allows us to artificially "tag" portions of the curve as being "in" different triangles, even if $\sigma(t)$ is not varying. Specifically, for $i - 1 < t < i$, $(t, \sigma(t))$ is "in" T_i and no other triangle, even though $\sigma(t)$, $t \in [i-1, i]$ might be fixed at $\rho(v)$ of T_i and therefore an element of *all* triangles which meet at this vertex. Similarly, for $t = i$, $(t, \sigma(t))$ is passing through the edge common to T_i and T_{i+1}, and no other.

Let $T_i = \rho(u)\rho(v)\rho(w)$ where $[\rho(u), \rho(v), \rho(w)] > 0$. (If T_i is degenerate, then T_i must be the limit of triangles with positive circulation.) Suppose $(t, \sigma(t))$ passes through $\rho(u)\rho(v)$ for $t = i - 1$ and

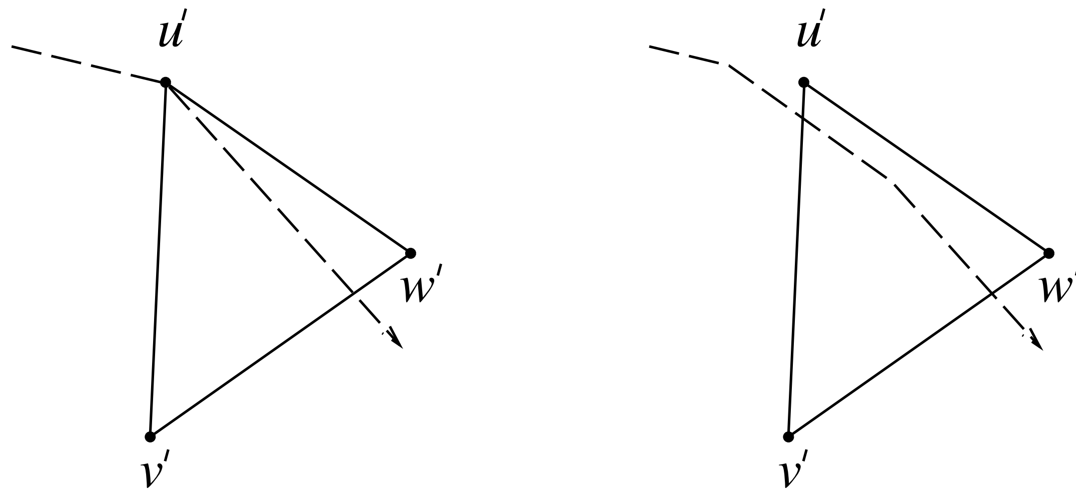

Figure 5: Curve (dashed) making a right turn at $u' = \rho(u)$ which is to its left. This curve can be made shorter without changing topology.

through $\rho(u)\rho(w)$ for $t = i$, then for $i - 2 < t < i + 1$, $\rho(u)$ lies "to the left" of σ. "To the right" is defined similarly. Since the list of triangles is minimal, σ cannot enter a triangle and then leave through the same edge. Therefore "to the left" and "to the right" are well-defined. Furthermore, for any sublist of triangles $T_i, \ldots, T_j$ which share a common vertex $\rho(v)$, the definition is consistent over the sublist.

Let $t \in [i, j]$ be a maximal interval on which $\sigma(t) = \rho(v)$ for some $v \in V$. The path $\sigma(t)$ "turns left" at $\rho(v)$ for $t = i$ if the sublist $T_i, \ldots, T_j$ winds around $\rho(v)$ at least once counterclockwise or it makes a left turn in the conventional sense (and the sublist does not wind around $\rho(v)$ at least once clockwise). "Turns right" is defined analogously.

Definition 5.1 *A* **locally shortest path geometric rounding** *of an edge ab in an embedding of $G = \langle V, E \rangle$ is a geometric rounding of ab that only turns left at $\rho(u)$, $u \in V$, to its left and that only turns right at $\rho(v)$, $v \in V$, to its right.*

Figure 5 depicts a curve that "turns right" at a vertex "to its left." This curve can be made shorter locally by taking a shortcut near the vertex.

Lemma 5.1 *The locally shortest path geometric rounding of an edge is unique.*

Proof: Suppose we have two locally shortest paths $\sigma(t)$ and $\tau(t)$ from $\rho(a)$ to $\rho(b)$. Suppose that they are not equal. This means that there is some value t_0 of t at which they diverge. Without loss of generality, $\sigma(t)$ is to the right of $\tau(t)$ in a neighborhood of $t > t_0$: the angle between the tangent vectors $\sigma'(t_0)$ and $\tau'(t_0)$ is positive (counterclockwise). The curves σ and τ cannot rejoin unless either σ makes a left turn or τ makes a right turn. However, the curves can only turn as they cross some edge $\rho(u)\rho(v)$ of the triangulation. (They must cross the same edges in the same order at the identical values of t.) Curve σ cannot pass through $\rho(u)$ because τ intersects $\rho(u)\rho(v)$ closer to $\rho(u)$, and similarly, curve τ cannot pass through $\rho(v)$. But σ can only make a right turn at $\rho(v)$ and τ can only make a left turn at $\rho(u)$ (see Figure 6). If σ makes a right turn or τ makes a left turn, then the angle of divergence becomes larger, never smaller. (Note: the angle cannot grow greater

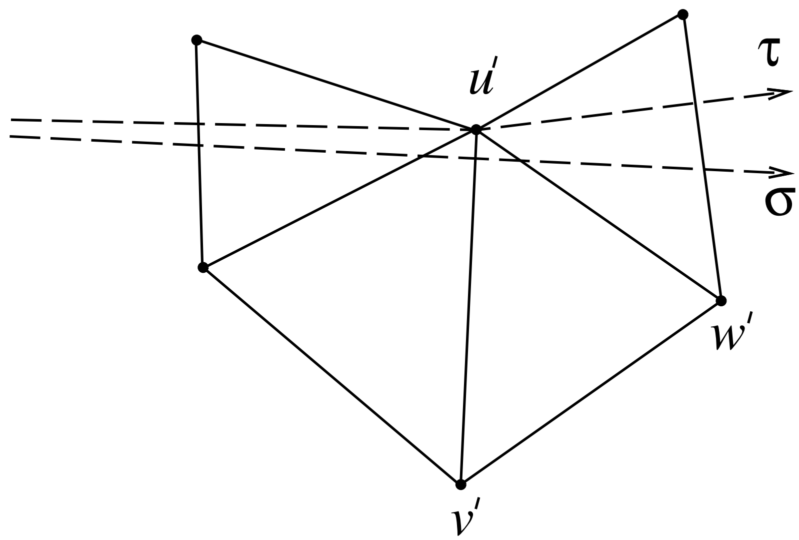

Figure 6: Curve τ makes a left turn at $u' = \rho(u)$ to its left. The divergence between curves σ and τ increases.

than 180 degrees otherwise σ and τ could not both pass through the next edge.) Therefore, if the curves diverge, they can never rejoin. This contradicts the fact that $\sigma(n+1) = \tau(n+1) = \rho(b)$. Therefore, the curves must be equal. ∎

5.2 Global Path Lemma

Lemma 5.2 *The global path algorithm generates a shortest path geometric rounding for each edge of the embedding, and that path is independent of the choice of rounding curves $\gamma_v(t)$, $v \in V$.*

Proof: The global path algorithm "loosely" threads the edge path from $\rho(a)$ to $\rho(b)$ past each rounded lattice point. To make sure it has the correct topology, it detours around each rounding curve that the edge from $\rho(a)$ to $\rho(b)$ intersects. Once it has the correct topology, it "pulls the string tight," computing the shortest path for that topology. It is clear that the topology is correct, and we will not give an explicit construction of the deformation of the plane that takes the initial embedding to the "loosely threaded" intermediate embedding. The deformation would be something like dragging fingers through frosting, where each finger starts at some $v \in V$, follows γ_v and stops at $\rho(v)$.

If any γ_v is modified continuously without leaving CELL$(\rho(v))$, then it cannot sweep through any other rounded vertex $\rho(u)$, $u \in V$, because $\rho(u) \notin$ CELL$(\rho(v))$. Therefore, the "loosely threaded" path varies continuously too, and does not sweep through any rounded vertex. This means that continuously modifying γ_v does not change the final topology. Since the output of the algorithm only depends on the topology (minimal list of triangles), the output is independent of the rounding curves γ_v, $v \in V$. ∎

5.3 Local Path Lemma

Lemma 5.3 *The local path algorithm generates a locally shortest path geometric rounding for each edge, and the union of these paths is a geometric rounding of the entire straight line embedding.*

Proof: The local path algorithm is a physical simulation. It is possible to give a mathematical proof, but a "physical" proof is simpler and more comprehensible.

The algorithm simulates the result of moving frictionless particles (vertices) pressing against flexible, elastic, impenetrable strings. The strings do not vibrate or "wave about", and therefore the system remains at (local) minimum potential energy at all times. For an elastic string, the potential energy is proportional to the length of the string. Therefore, the length of each path (representing an edge in the original embedding) is always at a local minimum for the given topology.

The shape of each string path depends only on the evolution of the vertices, not on the presence of other paths. Furthermore, each path reacts appropriately to *all* moving vertices which impinge on it. Therefore, it is not possible for a moving vertex to push one string "through" another.

We could endow each vertex and each edge (string) with a small thickness. Each vertex would be represented by its center, and each edge would be represented by its medial axis (the portion joining the centers of the two vertex endpoints). It is clear that for sufficiently small thickness, the physical motion can be made arbitrary close to the ideal zero-thickness case. The centers and medial axes are a deformation of the original embedding with the same topology: axes/centers cannot come into contact because of the "thickness" surrounding them. When each vertex in the thick model reaches its final resting place, one can shrink the thickness down to zero. At the moment the thickness reaches zero, the centers and axes become identical to the output of the local path algorithm. This demonstrates that there exists a topological deformation of the original embedding whose limit is the set of locally shortest paths.

This physical argument demonstrates that the local paths satisfy the definition of a geometric rounding. Since we have shown they have (local) minimum length, this proves the lemma. ■

5.4 Central Theorem of Shortest Path Rounding

This section proves Theorem 3.1, the central theorem of geometric rounding in Section 3.3 (page 10).

Proof: By Lemma 5.3, the local path algorithm generates a geometric rounding of the entire straight line embedding. However, since this is a physical simulation algorithm, the output might only be a local minimum of the path length, and it might depend on the choice of rounding curves γ_v, $v \in V$. However, Lemma 5.2 shows that the topology of the global minimum length geometric rounding for each individual edge is independent of the choice of rounding curves. Finally, Lemma 5.3 proves that there is only one local or global minimum length path for a given topology. Therefore, the local path algorithm and the global path algorithm generate the same output, and these are a geometric rounding of the entire embedding. ■

6 Applications

This section illustrates one of the ways we use shortest path rounding in practice. Since 1991, we have been developing algorithms for layout in the apparel industry. The basic problem is *strip* *packing*: given polygons $P_1, P_2, \ldots, P_k$ and a rectangle of fixed width and undetermined length, find the non-overlapping layout of the polygons with minimum length. In apparel applications, fabric has a grain, and thus each polygon has between one and eight valid orientations. The general strip packing problem is NP-hard. In practice, k is 100 or more, and the problem is intractable. However, we have shown that *translational* algorithms for modest values of k ($1 \leq k \leq 10$), are very useful in the development of heuristics or approximate algorithms for much larger values of k with multiple allowed orientations.

One useful algorithm is *translational minimum area enclosure*: given $P_1, P_2, \ldots, P_k$, find the layout under translation with the minimum area bounding rectangle. Aside from its usefulness in practice, this algorithm is an "acid test" for geometric rounding. It cascades algorithms for polygon decomposition, Minkowski sum, union, intersection, complement, convex hull, and linear programming. These algorithms do not apply transformations to the coordinates, but they use all three construction primitives mentioned in Section 1: 1) join points to make lines, 2) intersect lines to make points, and 3) add two points. In addition, the linear programming algorithm generates new point coordinates by solving a system of linear equations: in this case, $2(k+1)$ equations whose coefficients are linear or quadratic in the input point coordinates. The depth of computation (number of cascades) is arbitrary, even for a single problem instance. Furthermore, the minimum enclosure algorithm "deliberately seeks" degenerate cases.

We have described most of our layout algorithms in journal and conference papers and technical reports [21, 7, 23, 5, 30, 29, 6, 8, 33]. We have also licensed the implementations to industry. This section summarizes the minimum enclosure algorithm to give the reader an idea of how the problem of cascading can arise in practice.

6.1 Displacement Spaces

To solve the minimum enclosure problem, we solve a set of decision problems: does there exist a rectangle of area A which can contain the polygons? This decision problem is reduced to a displacement equation,

$$t_j - t_i \in U_{ij}, \qquad 0 \leq i < j \leq k+1, \tag{4}$$

where $t_1, t_2, \ldots, t_k$ are the translations applied to the polygons and where t_0 and t_{k+1} are the lower-left and upper-right corners of the enclosing rectangle.[8] Except for $U_{0,k+1}$, the U_{ij} regions are bounded depth constructions on $P_1, P_2, \ldots, P_k$. (For instance, $U_{ij} = \overline{P_i \oplus -P_j}$, $1 \leq i < j \leq k$, where $\oplus$ is the Minkowski sum.) In theory, $U_{0,k+1} = \{(x, y) \mid x{\cdot}y {\leq} A\}$ is bounded by a hyperbola. However, we use a polygonal approximation to the hyperbola. As a result, only polygonal operations are required, but the algorithm generates an enclosure whose area is an approximation to the minimum.

The minimum enclosure algorithm uses binary search to find the minimum area. Whenever the decision problem has a solution, the algorithm applies *compaction* [21, 23] (actually, a slight generalization), which moves the layout to a local minimum area. This greatly speeds up the binary search. However, since compaction involves no cascading, we will not summarize it here.

[8]We always set $t_0 = (0, 0)$, but it is easier to describe mathematically and implement in this more general form.

6.2 Solving the Displacement Equations

We refer to the list $\mathcal{U} = \langle U_{ij} \mid 0 \leq i < j \leq k+1 \rangle$ of displacement spaces as a *hypothesis* because it corresponds to the hypothesis that there exists a layout with that particular area. To solve Equation 4, the minimum enclosure algorithm applies three operations to a hypothesis: *restriction*, *evaluation*, and *subdivision*. *Restriction* replaces one or more U_{ij} by a subset without changing the truth value of the hypothesis (without throwing away any valid solutions). *Evaluation* attempts to find a solution within a given hypothesis. *Subdivision* selects one pair i, j and splits U_{ij} into U_{ij}^{+} and U_{ij}^{-}. Replacing U_{ij} by either of these generates two sub-hypotheses $\mathcal{U}^{+}$ and $\mathcal{U}^{-}$. The hypothesis $\mathcal{U}$ is true if and only if $\mathcal{U}^{+}$ is true or $\mathcal{U}^{-}$ is true. Evaluation is not constructive. Subdivision only involves intersection with two half-planes, although the overall depth of subdivision is arbitrary. Restriction can involve unbounded cascading.

The minimum enclosure algorithm employs two types of restriction: geometric restriction and linear programming (LP) restriction. Geometric restrict performs the following substitution,

$$U_{ij} \leftarrow U_{ij} \cap (U_{ih} \oplus U_{hj}), \qquad 0 \leq i < j \leq k+1, \qquad 0 \leq h \leq k+1, h \neq i, j,$$

where U_{ih} is defined to be $-U_{hi}$ if $h < i$. The algorithm applies this restriction repeatedly until a "steady state" is reached. In practice, we stop when the decrease in area drops below a fixed fraction. Geometric restriction arbitrarily cascades the operations of intersection and Minkowski sum.

Linear programming restriction shrinks each U_{ij} in a different way. It first constructs an outer convex approximation to the displacement equation,

$$t_j - t_i \in \mathrm{CH}(U_{ij}), \qquad 0 \leq i < j \leq k+1,$$

where $\mathrm{CH}(U_{ij})$ is the convex hull of U_{ij}. Using an adaptation of the simplex method, it constructs the *range* of $t_h - t_g$ under this convex approximation. The range is a convex polygonal region C_{gh}. The following substitution is a valid restriction,

$$U_{gh} \leftarrow U_{gh} \cap C_{gh}.$$

It applies this substitution for each pair g, h until a steady state is reached. Again, the cascading is arbitrary. In addition to straight-edge constructions, linear programming generates new point coordinates by solving $2(k+1)$ linear equations in $2(k+1)$ variables. The coefficients of these equations are linear or quadratic in the coordinate of the vertices of the displacement spaces.

6.3 Results

Figure 7 illustrates the minimum enclosure algorithm on five input polygons. Thanks to compaction, the algorithm can apply a very "lop-sided" binary search. Iteration 1 is a square container with compaction applied. For iterations 2-5, the algorithm set the target area to be 1% less than the previous layout after compaction. Iteration 5 was infeasible. The algorithm set the target for iteration 6 to be 0.01% smaller than the area of iteration 4, and similarly, iteration 7 and 8 have targets 0.01% than the previous layouts after compaction. Iteration 8 was infeasible, and therefore iteration 7 is within 0.01% of optimum. The polygonal approximation to the hyperbola had 100 vertices, which introduces an additional error of at most 0.01%.

The running times ranged from 2 minutes for iteration 1 to 45 minutes for iteration 8. Total time on a DEC Alpha 3000/700 is about 2.5 hours.[9] Solving iteration 8 required 318 subdivisions and

[9] This computer is advanced 1994 technology. A 1998 PC (400MHz Pentium II) is about twice as fast.

Figure 7: Minimal enclosing rectangle of five polygons with 55, 61, 66, 65 and 72 vertices.

steady state restrictions. Each of these 318 steady state restrictions involves a considerable amount of cascading, and in addition, the depth of subdivision averaged about 9. Iteration 7 required 246 subdivisions and 34 minutes. These iterations bracket the tightest possible layout.

Without geometry rounding, this type of calculation would simply be impossible. With geometry rounding, we see no numerical problems at all, even for this near-degenerate and highly cascaded construction.

7 Analysis and Conclusions

The previous section demonstrates that geometric rounding is an absolute necessity, at least for some important applications. However, shortest path rounding is not the only type of geometric rounding. For the integer grid, Section 1.2.2 discussed Greene-Yao rounding and snap rounding as alternative types of rounding. This section presents the ways in which shortest path geometric rounding is a better choice than these other two rounding techniques.

It should be emphasized that running time is not a critical issue in the choice of rounding technique. All three rounding techniques essentially run in time linear in the number of vertices added to each rounding path. As indicated in Section 4.5.1, the lattice point $\rho(v)$ for vertex $v \in V$ can only appear on the rounding path for edge $ab \in E$ if it is very "near" to ab, and this should be a rare occurrence.

The shortest path algorithm in Section 4.5.2 performs a number of circulation tests which is proportional to the number of distance calculations required for snap rounding. For programs in which snap rounding is applicable, switching from snap rounding to shortest path rounding only increases a small fraction of the running time by a small factor. It may not even be possible to measure the overall difference in running time.

The more important issue is how much "damage" rounding inflicts when it does become necessary. Sections 7.1 and 7.2 show that on integer grids, shortest path rounding introduces less geometric and combinatorial error than the other rounding methods. Another issue is the generality of the rounding technique. Section 7.3 gives several reasons why it is useful to round on a non-uniform grid, which only shortest path rounding can handle.

7.1 Rounding on the Integer Grid

All three rounding methods replace a line segment ab by a polygonal path from $\rho(a)$ to $\rho(b)$. The vertices of the path are at lattice points. If the lattice is the integer grid, then a lattice point p can be a vertex of the path only if ab intersects $\mathrm{CELL}(\rho(p))$, in which case we say that p is *near ab*. If p is $\rho(v)$ for some $v \in V$, then we say that p is a *vertex lattice point.* Snap rounding replaces ab by a path joining all *near vertex lattice points.*

Greene-Yao rounding does not necessarily "tie" the path to all near vertex lattice points. It ties the path to $\rho(v)$ only if $v\rho(v)$ crosses ab. However, it does not permit the path to "sweep past" *any* lattice point, even non-vertex lattice points, as it is "pulled" to these special lattice points. As a result, it ends up adding $\Omega(\log |ab|)$ extra non-vertex lattice points to the path for every vertex lattice point on the path. For a grid of pixels, typically about 1000 by 1000 on current graphics displays, $\ln |ab|$ is perhaps not too large. Other industrial applications generally require higher accuracy. Using a 10^6 by 10^6 grid might mean that 10 to 20 non-vertex lattice points are added for each vertex lattice point on the path. For such applications, this number of "extra" vertices would make Greene-Yao rounding an impractical choice.

Shortest path rounding does not "tie" the path to any vertices, except of course $\rho(a)$ and $\rho(b)$. Even if $v\rho(v)$ crosses ab, $\rho(v)$ might not be a vertex on the rounded path. Vertex v "pushes" on the path as v rounds to $\rho(v)$, but other vertices might push it past $\rho(v)$. In particular, u and w, $u, w \in V$ might be near to v and on the same side of ab. If v and $\rho(v)$ lie on the same side of segment $\rho(u)\rho(w)$, then as u and w push on the path, they will push it "past" $\rho(v)$, and $\rho(v)$ will not lie on the path. Unlike Greene-Yao rounding, shortest path rounding only puts *vertex* lattice points on the path, and these path vertices are a subset of the points on the snap rounding path.[10]

7.2 Analysis of Error

Snap rounding and shortest path rounding both add many fewer vertices to the rounded paths than Greene-Yao rounding, and shortest path rounding adds somewhat fewer vertices than snap rounding. Also, the shortest path has the minimum possible deviation from the original line segment. How significant is the difference between snap rounding and shortest path rounding? The difference is "merely" a constant factor. However, cascading can multiply these constants into exponential differences. This section argues that for each vertex that snap rounding puts on a rounding path, shortest path rounding will put that vertex on the path with a probability between 1/6 and one-quarter. Shortest path rounding introduces a geometric error with a standard deviation about 1/3 of the error introduced by snap rounding. The standard deviation of the appropriate measure of error: cascaded rounding is essentially a random walk, and the result of a random walk is a Gaussian who standard deviation is proportional to the standard deviation of the rounding distribution.

7.2.1 Combinatorial Error

We simplify the analysis[11] by considering only the case in which exactly one vertex lattice point $\rho(v)$ lies near to ab (ab intersects $\mathrm{CELL}(\rho(v))$). Snap rounding *always* puts $\rho(v)$ onto the path. Shortest path rounding puts $\rho(v)$ on the path only if v and $\rho(v)$ lie on opposite side of ab. Assuming that the distance from $\rho(v)$ to ab is uniformly distributed and that v is uniformly distributed in $\mathrm{CELL}(\rho(v))$, one can show that v lies in the portion of $\mathrm{CELL}(\rho(v))$ on the side of ab opposite from $\rho(v)$ with probability 1/6 if ab has 45 degree slope and with probability one-quarter if ab has 0 degree or 90 degree slope.

7.2.2 Geometric Error

In analyzing the standard deviation, we will consider only the case in which ab either rounds to $\rho(a)\rho(b)$ or $\rho(a)\rho(v)\rho(b)$. The question is, what is the standard deviation of the error introduced by vertex v? Since we are only concerned with the ratio of standard deviations, we will call the maximum deviation one "unit". If ab has 45 degree slope, one unit of error is actually $\sqrt{2}/2$. If ab has 0 or 90 degree slope, one unit of error is $1/2$.

If $\rho(v)$ lies near ab, then snap rounding will always snap to $\rho(v)$. The deflection along the path $\rho(a)\rho(v)\rho(b)$ is uniformly distributed from 0 to the distance δ from $\rho(v)$ to $\rho(a)\rho(b)$. Therefore

[10] The subset can be improper: the two paths might be the same.

[11] For this reason, we call this section an "argument", not a "proof".

the distribution of deflection varies according to the following distribution: select δ uniformly from $[0,1]$ and then select ϵ uniformly from $[0,\delta]$. The error ϵ has the distribution $-\ln\epsilon$. The standard deviation is $1/3$.

If $\rho(v)$ lies near ab and it is the only vertex which does so, then shortest path rounding will only snap the path to $\rho(v)$ if $v\rho(v)$ intersects ab. For this to happen, v must lie in the portion of CELL($\rho(v)$) that is on the opposite side of ab from $\rho(v)$. If ab has 45 degree slope, this event has probability $(1-\delta)^2/2$, and if ab has 0 or 90 degree slope, the probability is $(1-\delta)/2$. For the 45 degree case, the distribution of error along the segment is constructed as follows: select δ uniformly from $[0,1]$ and then select ϵ uniformly from $[0,\delta]$ with probability $(1-\delta)^2/2$ but set $\epsilon = 0$ with probability $1-(1-\delta)^2/2$. The 0 or 90 degree is analogous. The 45 degree distribution is,

$$-\frac{1}{4}\epsilon^2 + \epsilon - \frac{1}{2}\ln\epsilon - \frac{3}{4},$$

and the 0 or 90 degree distribution is,

$$\frac{1}{2}\epsilon - \frac{1}{2}\ln\epsilon - \frac{1}{2}.$$

Surprisingly, these both have the same standard deviation: $1/\sqrt{72}$. This is $1/2\sqrt{2} \approx 1/3$ times the standard deviation of snap rounding.

7.3 Non-Uniform Grids

The most common non-uniform numerical representation is floating point: mantissa plus exponent. Obviously floating point would not be as popular as it is if it did not have many technical advantages. One advantage is that is seamlessly handles changes in scale or unit. We first licensed our layout software to Microdynamics, Inc., which used a unit of 0.01 inch. Gerber Garment Technologies (GGT) bought Microdynamics and took over the license. GGT uses a unit of 0.001 inch. Our software used integer arithmetic and the integer grid. As a consequence, it had some absolute rather than relative tolerance values. Unfortunately, accommodating GGT was not simply a matter of changing constants and recompiling: they still had to service the former Microdynamics customers. We could have avoided all of this inconvenience if we had used floating point computations and rounded to the floating point grid. Of course, Shewchuk's work (Section 2.1) was not available at that point in time.

We emphasize again that it is not necessary to compute the nearest floating point coordinate to every exact coordinate. Shortest path rounding can round to *any* lattice set S as long as the rounding cells are connected. The easiest way to accomplish this is to use a round-to-nearest strategy: the rounding cells are simply the convex Voronoi cells of S. Since the rounding cells are convex, the rounding curves can be straight line segments. When a point needs to be rounded, the system can check to see if it is close enough to an existing vertex lattice point in S. If not, a new lattice point can be added to S. This approach involves point location and update of a Voronoi diagram, both very well understood problems. It would use the global path algorithm.

If the rounding needs to be independent of a specific xy coordinate frame, then one would use the approach in the previous paragraph. Otherwise, it is even simpler to use the local path algorithm. Maintain a set S_x of x lattice coordinates and a set S_y of y lattice coordinates. These sets can be built incrementally. To round a point (x, y), round each coordinate to the nearest existing x and y lattice coordinates in S_x and S_y. If the nearest is too far, compute an approximation to x and/or

y and add these to S_x and/or S_y. Maintaining an ordered set of numbers with find-nearest and insert operations is a very well understood problem. This approach would use the monotone path algorithm. This is the approach we would recommend in practice.

7.4 Conclusion

Many common, useful, and practical applications of computation geometry have exponential space and time owing to numerical issues and cascading. Geometric rounding reduces the cost to what it would be in the absence of cascading. Numerical error is the price paid for this reduced cost, but this tradeoff is a reasonable and well-understood principle of numerical computing. Geometry rounding does not require any modification of the geometric algorithm or their exact arithmetic implementation. The algorithm can even use symbolic perturbation.

Shortest path geometric rounding introduces the minimum geometric deviation of any method that introduces only vertex lattice points (meaning that the only vertices in the output are rounded locations of vertices in the input). On the integer grid, it introduces many fewer vertices that Greene-Yao rounding and fewer vertices and less deflection than snap rounding.

Unlike other rounding methods, shortest path rounding can handle any connected lattice, even rounding in polar coordinates, but most importantly, the floating point lattice. The *global path* algorithm uses standard, easily implemented algorithms of computation geometry: triangulation, segment intersection, and shortest path in a simple polygon. The *monotone path* algorithm is even simpler to implement and use, and a complete implementation is given in this paper.

Shortest path rounding has very low overhead. In combination with numerical techniques for exact floating point computation, it offers the ideal implementation for any algorithm on polygonal regions: floating point input, exact computation, cost per arithmetic operation a little more than hardware floating point, rounded floating point output. Finally, its use is well-established in licensed industrial application software.

References

[1] C. Bajaj and T. K. Dey. Polygon nesting and robustness. *Inform. Process. Lett.*, 35:23–32, 1990.

[2] Behnke, Bachmann, Fladt, and Kunle. *Fundamentals of Mathematics, Volume II: Geometry.* MIT Press, Cambridge, MA, 1974.

[3] J. Canny, B. R. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 251–260, 1992.

[4] Wei Chen, Koichi Wada, and Kimio Kawaguchi. Parallel robust algorithms for constructing strongly convex hulls. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 133–140, 1996.

[5] K. Daniels. *Containment algorithms for nonconvex polygons with applications to layout.* Ph.D. thesis, Harvard University, Cambridge, MA, 1995.

[6] K. Daniels and V. Milenkovic. Column-based strip packing using ordered and compliant containment. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, pages 33–38, 1996.

[7] K. Daniels, V. Milenkovic, and Z. Li. Multiple containment methods. Technical Report 12-94, Center for Research in Computing Technology, Division of Applied Sciences, Harvard University, Cambridge, MA, 1994.

[8] K. Daniels and V. J. Milenkovic. Multiple Translational Containment, Part I: An Approximate Algorithm. *Algorithmica*, 19:148–182, 1997.

[9] T. K. Dey, K. Sugihara, and C. L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Comput. Aided Geom. Design*, 9:457–470, 1992.

[10] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1):193–213, 1995.

[11] S. Fortune. Polyhedral modeling with multiprecision integer arithmetic. *Comput. Aided Design*, 29(2):123–133, 1997.

[12] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 334–341, 1991.

[13] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[14] J. E. Goodman, R. Pollack, and B. Sturmfels. Coordinate representation of order types requires exponential storage. In *Proc. 21st Annu. ACM Sympos. Theory Comput.*, pages 405–410, 1989.

[15] D. H. Greene. Integer line segment intersection. unpublished manuscript.

[16] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 143–152, 1986.

[17] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[18] Leonidas Guibas and David Marimont. Rounding arrangements dynamically. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 190–199, 1995.

[19] J. D. Hobby. Practical segment intersection with finite precision output. submitted for publication.

[20] H. Inagaki and K. Sugihara. Numerically robust algorithm for contructing constrained Delaunay triangulation. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 171–176, 1994.

[21] Z. Li. *Compaction algorithms for nonconvex polygons and their applications.* Ph.D. thesis, Harvard University, Cambridge, MA, 1994.

[22] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. *Algorithmica*, 8:345–364, 1992.

[23] Z. Li and V. Milenkovic. Compaction and separation algorithms for nonconvex polygons and their applications. *European Journal of Operations Research*, 84:539–561, 1995.

[24] V. Milenkovic. *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic.* Phd thesis, Carnegie Mellon University, 1988.

[25] V. Milenkovic. Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 500–505, 1989.

[26] V. Milenkovic. Rounding face lattices in the plane. In *Abstracts 1st Canad. Conf. Comput. Geom.*, page 12, 1989.

[27] V. Milenkovic. Rounding face lattices in d dimensions. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 40–45, 1990.

[28] V. Milenkovic. Robust polygon modeling. *Comput. Aided Design*, 25(9):546–566, 1993. (special issue on Uncertainties in Geometric Design).

[29] V. Milenkovic. Translational polygon containment and minimal enclosure using linear programming based restriction. In *Proc. 28th Annu. ACM Sympos. Theory Comput. (STOC 96)*, pages 109–118, 1996.

[30] V. Milenkovic and K. Daniels. Translational polygon containment and minimal enclosure using geometric algorithms and mathematical programming. Technical Report 25-95, Center for Research in Computing Technology, Division of Applied Sciences, Harvard University, Cambridge, MA, 1995.

[31] V. Milenkovic and L. R. Nackman. Finding compact coordinate representations for polygons and polyhedra. *IBM J. Res. Develop.*, 34:753–769, 1990.

[32] V. J. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artif. Intell.*, 37:377–401, 1988.

[33] V. J. Milenkovic. Multiple Translational Containment, Part II: Exact Algorithms. *Algorithmica*, 19:183–218, 1997.

[34] V. J. Milenkovic and V. Milenkovic. Rational orthogonal approximations to orthogonal matrices. *Computational Geometry: Theory and Applications*, 7:25–35, 1997.

[35] Victor J. Milenkovic. Practical methods for set operations on polygons using exact arithmetic. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 55–60, 1995.

[36] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[37] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *J. Inform. Proc.*, 12(4):380–393, 1989.

[38] Kokichi Sugihara. A robust and consistent algorithm for intersecting convex polyhedra. *Comput. Graph. Forum*, 13(3):45–54, 1994. Proc. EUROGRAPHICS '94.