

# A NON-TRIGONOMETRIC, PSEUDO AREA PRESERVING, POLYLINE SMOOTHING ALGORITHM

Wayne Brown and Leemon Baird  
Department of Computer Science  
The United States Air Force Academy  
2354 Fairchild Dr., Suite 6G-101  
USAF Academy, CO 80840-6208  
719 333-3590  
Wayne.Brown@usafa.edu, Leemon.Baird@usafa.edu

## ABSTRACT

A line-smoothing algorithm based on simple arithmetic is presented and its characteristics are analyzed for various implementations. The algorithm is efficient because it can be implemented using only simple integer arithmetic, with no square root or trigonometric calculations. The algorithm is applicable to graph drawing applications that require smooth polylines between graph nodes. Though the algorithm is efficient, it is sensitive to seemingly minor implementation details which make it an illustrative and illuminating student/classroom exercise for the discussion of integer round-off, integer overflow, and the importance of implementation details.

## 1 INTRODUCTION

The Game of Sprouts [1], [2] is played by two opponents who take turns connecting two free nodes in a graph with a curved line, called a polyline, that does not cross any existing line in the graph. A free node is any node that has less than three connected lines. When a player connects two nodes with a polyline, a new node is created in the middle of the polyline. A player wins when they connect two free nodes and their opponent cannot on the succeeding turn. During the implementation of this game on a Personal Digital Assistant (PDA), an algorithm is needed to spread the lines drawn by the players to allow space for future moves and, at the same time, to keep the lines smooth and the graph visually pleasing. Since the implementation is on a PDA, the line smoothing needs to be efficient and minimize expensive calculations such as square root and trigonometric functions. This paper presents a polyline smoothing algorithm and analyzes its properties. The algorithm is sensitive to implementation details: it is stable and pseudo area-preserving for some implementations but unstable for slight variations. This algorithm can be especially useful in a pedagogical setting for illustrating issues of integer round-off, integer overflow, and details of implementation.

## 2 PREVIOUS WORK

There are many algorithms for drawing smooth lines that either approximate or pass through a set of control points, such as Bezier curves, B-splines, and NURBS [3]. There are also many algorithms that will manipulate a curve's control points to meet curvature and/or other constraints to meet specific design criteria. Our algorithm differs from this previous work because it has no "end shape" goal for a polyline. The algorithm's only goal is a "smooth" polyline that is non-crossing.

Simple algorithms for polyline smoothing, such as moving-averages, Laplacian smoothing and Bilaplacian smoothing, move vertices based on the average location of their neighboring vertices. These simple methods will smooth and shrink a polyline over time, collapsing a closed polyline into a single point. Various modifications to these methods [4] can make them area-preserving at the cost of computational complexity. Arvo [5] presents a polyline smoothing technique that is "appearance preserving." Our algorithm does not attempt to maintain the original polyline shape created by the user. Lutterkort [6] presents a method for creating smooth polylines that are restricted to a "channel." Our algorithm does not place any such constraints on the resulting polyline. Our work differs from previous

work in that it can smooth polylines using only simple integer arithmetic while being pseudo area-preserving for closed-loop polylines.

### 3 THE POLYLINE SMOOTHING ALGORITHM

The purpose of this algorithm is to move the vertices of a polyline such that the polyline becomes "smooth." The algorithm is fairly simple: given five sequential vertices along a polyline,  $v_1$ ,  $v_2$ ,  $v_3$ ,  $v_4$ , and  $v_5$ , it moves the middle vertex,  $v_3$ , such that the area of the triangle formed by the three middle vertices,  $v_2$ ,  $v_3$ , and  $v_4$ , is equal to the average area of the three triangles formed by the five vertices. In addition, the middle vertex,  $v_3$ , must lie on the perpendicular bisector of the line segment between the second and fourth vertices,  $v_2$ , and  $v_4$ . For example, referring to Fig. 1, vertex  $v_3$  will move based on the average area of the triangles  $A_2$ ,  $A_3$ , and  $A_4$  and be positioned on the perpendicular bisector of  $v_2v_4$ .

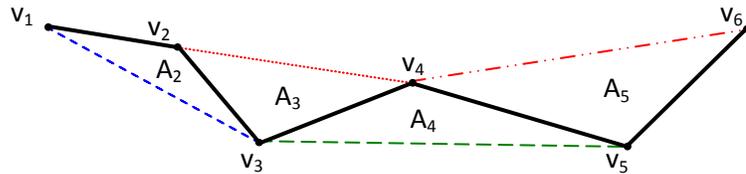


Fig. 1. Six vertices and five line segments along a polyline that is to be smoothed, along with the triangle areas used for smoothing.

#### 3.1 Implementation Details

To calculate the area of a triangle defined by three vertices  $v_1$ ,  $v_2$ , and  $v_3$ , use the "cross-product" formula shown in Eq. 1.

$$\text{TriangleArea} = (v_1.x*v_2.y - v_1.y*v_2.x) + (v_1.y*v_3.x - v_1.x*v_3.y) + (v_3.y*v_2.x - v_2.y*v_3.x) \quad (1)$$

Eq. 1 actually calculates twice the area of a triangle, which fits nicely into later equations and avoids a division by 2. The resulting area is positive if the vertices form a counter-clockwise rotation and negative if they form a clockwise rotation. This sign is important because it indicates which side of the triangle base the middle vertex should be moved.

The triangle area used to move the middle vertex is the average of three triangle areas. Using the example above from Fig. 1, this would be areas  $A_2$ ,  $A_3$ , and  $A_4$ . (Remember, Eq. 1 calculated twice the triangle area; therefore the areas are doubled.)

$$A_{\text{avg}} = (2*A_2 + 2*A_3 + 2*A_4) / 3 \quad (2)$$

Referring to Fig 2, to calculate the new location for  $v_3$ , called  $v_3'$ , start at the midpoint of  $v_2v_4$ , called  $v_m$ , and move along a vector that is perpendicular to the vector  $v_2v_4$  and that has a magnitude of  $|v_mv_3'|$ . All of these values are easily calculated using the standard formula for the area of a triangle,  $1/2*base*height$ , and vector algebra, but square root calculations are required to compute the vector magnitudes.

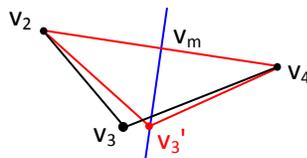


Fig. 2. Adjusting triangle  $v_2v_3v_4$  such that the location of  $v_3'$  is on the perpendicular bisector of the line segment  $v_2v_4$ .

To avoid solving for the actual length of vector  $v_2v_4$ , note that the ratio of a triangle's height to its base can be calculated from the standard triangle area formula, shown in Eq. 3, resulting in Eq. 6.

$$\text{TriangleArea} = (1/2) * \text{base} * \text{height} \quad (3)$$

$$\text{height} = (2 * \text{TriangleArea}) / \text{base} \quad (4)$$

$$\text{height} / \text{base} = (2 * \text{TriangleArea}) / \text{base}^2 \quad (5)$$

$$\text{RatioOfHeightToBase} = A_{\text{avg}} / \text{base}^2 \quad (6)$$

Therefore, the desired vertex  $v_3'$  is on a vector that is the perpendicular bisector of  $v_2v_4$ , (which is rotated 90 degrees from the vector  $v_2v_4$ ), and *RatioOfHeightToBase* times this vector's magnitude. Eqs. 7-10 show these calculations.

$$\text{base}_{dx} = v_4.x - v_2.x \quad \text{base}_{dy} = v_4.y - v_2.y \quad (7)$$

$$v_{m.x} = (v_2.x + v_4.x) / 2 \quad v_{m.y} = (v_2.y + v_4.y) / 2 \quad (8)$$

$$\begin{bmatrix} \text{Perpendicular}_{dx} \\ \text{Perpendicular}_{dy} \end{bmatrix} = \begin{bmatrix} \cos 90 & -\sin 90 \\ \sin 90 & \cos 90 \end{bmatrix} \begin{bmatrix} \text{base}_{dx} \\ \text{base}_{dy} \end{bmatrix} = \begin{bmatrix} -\text{base}_{dy} \\ \text{base}_{dx} \end{bmatrix} \quad (9)$$

$$v_3'.x = v_{m.x} + \text{RatioOfHeightToBase} * \text{Perpendicular}_{dx} \quad (10)$$

$$v_3'.y = v_{m.y} + \text{RatioOfHeightToBase} * \text{Perpendicular}_{dy}$$

The location for  $v_3'$  must be on the side of the triangle indicated by the sign of the average triangle area calculation. There are two possibilities, as shown in Fig. 3. In case 1, the triangle area is positive and the direction of the rotated bisector vector must be inverted. In case 2, the triangle area is negative and the direction of the bisector vector is correct, but the area needs to be positive. Therefore, if the sign of the triangle area is inverted, both cases are handled correctly.

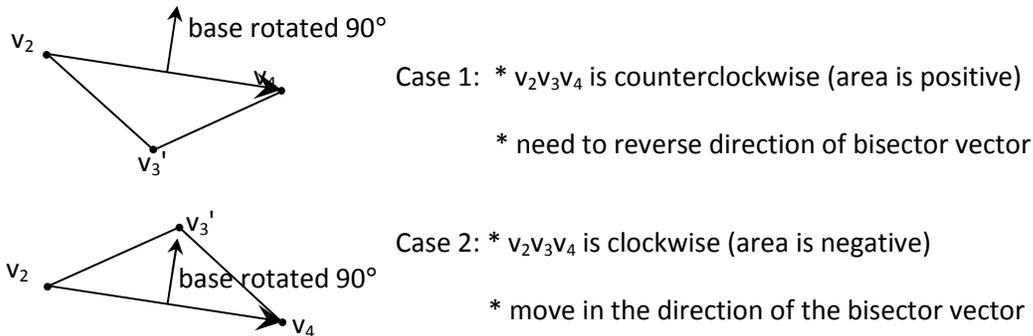


Fig. 3. Two possible directions of the perpendicular bisector vector based on the orientation of the triangle vertices.

In conclusion, the desired location of  $v_3'$  can be calculated using no trigonometric or square root calculations, as shown in Eqs. 11 and 12.

$$\text{base}^2 = \text{basedx}^2 + \text{basedy}^2 \quad (11)$$

$$v_3'.x = (v_2.x + v_4.x) / 2 + (-A_{\text{avg}} / \text{base}^2) * \text{-basedy} \quad (12)$$

$$v_3'.y = (v_2.y + v_4.y) / 2 + (-A_{\text{avg}} / \text{base}^2) * \text{basedx}$$

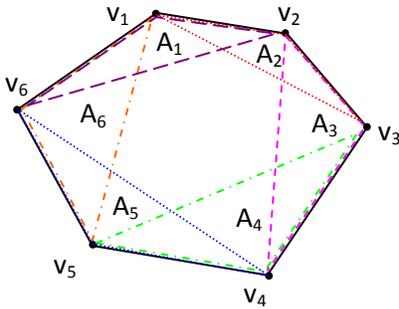
### 3.2 Applying the Algorithm to Polylines.

If this smoothing algorithm is repeatedly applied to the vertices of a closed-loop polyline, each vertex on the line is guaranteed to have two preceding and two trailing vertices (assuming the polyline contains a minimum of 3 vertices). Each vertex will cease to move when the area of its associated triangle is equal to the average area of its three sequential triangles (Eq. 13), or the average of its preceding and succeeding triangles (Eq. 14).

$$A_i = (A_{i-1} + A_i + A_{i+1})/3 \quad (13)$$

$$A_i = (A_{i-1} + A_{i+1})/2 \quad (14)$$

Therefore, referring to the closed-loop, six vertex polyline shown in Fig. 4, Eq. 15 expresses what must be true for the polyline to become stable (i.e., the vertices stop moving).



$$\frac{1}{2} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{bmatrix} \quad (15)$$

$$\frac{1}{2} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \end{bmatrix} = 0 \quad (16)$$

Fig. 4. A closed-loop polyline composed of six vertices with their associated triangle areas.

The coefficient matrix of Eq. 15 is singular and therefore the equation has an infinite number of solutions (as intuition would tell us). Subtracting the right hand side of Eq. 15 from both sides and using Gaussian Elimination to reduce the resulting matrix produces Eq. 16. This matrix shows that the algorithm will cease to change the location of vertices when triangle areas 1 through 5 are all equal to area 6. Given that the shape is closed and every triangle shares a "non-base" edge with more than one triangle, all the triangles must have the same base and height. Therefore, a closed-loop polyline will only become a stable shape if it is transformed into a circular shape where all sides and all interior angles are equal (or all the triangle areas become zero and the shape collapses to a single point). These equations do not prove that the algorithm will transform any arbitrary closed polyline into a circular shape, but they do prove that a closed polyline will never reach stability until it becomes a circular shape.

A mathematical analysis similar to the one presented for closed-loop polylines can be performed for a polyline that is not closed and the results are identical. Given the polyline shown in Fig. 1, v1 and v6 are assumed to be stationary and the algorithm is executed on the interior vertices. Vertices v2 and v5 must be treated as special cases since v2 has no "preceding" triangle and v5 has no "succeeding" triangle. One option for the movement of v2 is to use the average triangle area of only two triangles (i.e., A2 and A3). Another option is to use the average area of the three triangles A2, A3, and A4. Both options produce the same results: the polyline will cease to move under this algorithm if all the triangle areas are equal. However, since the "non-base" edges are not all interdependent, the final shape of the polyline will vary from case to case based on the initial conditions of the polyline.

### 3.3 Using Integer Coordinates and Integer Smoothing Calculations.

As stated in the introduction, our goal was the implementation of a Sprouts game on a PDA using only integer arithmetic. The smoothing algorithm works well using only integers if the implementation is done carefully to account for round-off and overflow.

**Round-off.** If Eq. 12 is implemented as the sum of two fractions (which is consistent with the logic of the equations) and both fractions are rounded to their closest integer result, a closed-loop polyline will sometimes converge to its circular shape but then continually move along the positive x and/or positive y axis. To eliminate this undesired motion, Eq. 12 must be implemented as a single fraction, as shown in Eq. 17.

$$\begin{aligned}v3'.x &= ( (v2.x + v4.x) * \text{base}^2 + 2 * (-A\text{avg}) * \text{-basedy} ) / (2 * \text{base}^2) \\v3'.y &= ( (v2.y + v4.y) * \text{base}^2 + 2 * (-A\text{avg}) * \text{basedx} ) / (2 * \text{base}^2)\end{aligned}\quad (17)$$

**Overflow.** Our algorithm was implemented in Java, which has no mechanism for detecting integer overflow. Therefore, special care must be taken to insure that the numerator of Eq. 17 does not become too large. Eq. 17 calculates the midpoint of vector v2v4 in global coordinates. To calculate a bound for the numerator, the midpoint needs to be calculated in local coordinates, as shown in Eq 18.

$$\begin{aligned}v3'.x &= v2.x + ( (v4.x - v2.x) * \text{base}^2 + 2 * (-A\text{avg}) * \text{-basedy} ) / (2 * \text{base}^2) \\v3'.y &= v2.y + ( (v4.y - v2.y) * \text{base}^2 + 2 * (-A\text{avg}) * \text{basedx} ) / (2 * \text{base}^2)\end{aligned}\quad (18)$$

To find an upper bound for Eq. 18, assume that **L** is the length of the longest segment of a polyline. The base of a triangle using two polyline segments is bounded by  $2 * L$  and the largest possible triangle area is bounded by  $L^2/2$ . Therefore, to guarantee that the numerator of Eq. 18 is less than a 32-bit signed integer, the length of polyline segments must be less than 812 (using Eq. 19). To eliminate this limit, vertex coordinates can be scaled down to this bound before the smoothing calculations are performed and then re-scaled after the calculations are complete. However, our experiments produced unacceptable precision loss using this technique.

$$L * (2 * L)^2 + 2 * (0.5 * L^2) * (2 * L) \leq 2^{31} - 1 \quad (19)$$

## 4 IMPLEMENTATION RESULTS AND VARIATIONS

A test-bed system was implemented in Java to validate our algorithm. Fig. 6 shows a randomly generated, 20-segment, closed-loop polyline being changed into a final circular shape. Each new picture represents 185 iterations of the algorithm, where each vertex moved a maximum of one pixel per iteration.

Several implementation factors have a significant impact on how this smoothing algorithm modifies a polyline's vertices over time. These factors include:

- 1) Integer vs. floating point vertex coordinates with the algorithm calculations performed in integer or floating point arithmetic.
- 2) Vertices are moved to the exact location calculated by the algorithm or only a short distance in the direction of the algorithm's calculated location.
- 3) Each vertex is changed immediately as a polyline is processed or all calculations are performed on the entire polyline before any vertices are changed.

Our experiments show that the algorithm transforms a polyline into a stable shape using integer arithmetic while moving each vertex immediately as it is processed. The other variations can cause the algorithm to become unstable and produce infinite oscillations in the polyline.

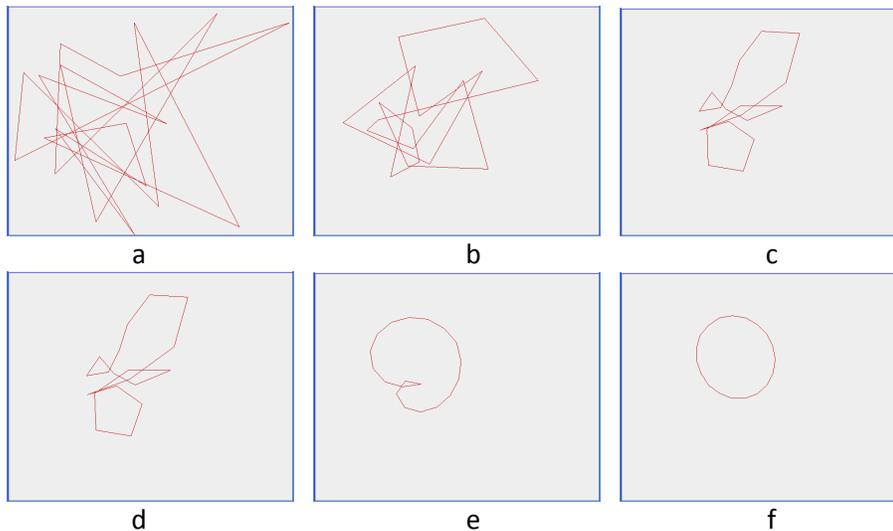


Fig. 6. A 20-segment, closed-loop polyline transformed by our algorithm using integer arithmetic, incremental vertex movement, and simultaneous vertex updates. (185 iterations per image)

## 5 CONCLUSIONS

The algorithm presented can smooth polylines using integer arithmetic without the use of trigonometric or square root calculations. This algorithm is pseudo area preserving for closed-loop polylines (which means the algorithm does not preserve the exact area inside a closed-loop polyline, but the shape does not collapse to a point over time). This smoothing algorithm is used in an implementation of a computer-based Sprouts game and produces visually pleasing graph drawings.

Our algorithm is simple, easy to implement, and easy to graphically animate. Implementation of this algorithm could be used as a computer science programming assignment for the investigation of integer overflow and integer round-off issues. It is also a good example of how small implementation details can have a large effect on an algorithm's behavior. For example, a course assignment might provide equations 1-12 and ask students to implement the algorithm. Or, in an analysis course, students could be asked to derive equations 17-19 to minimize integer round-off and prevent integer overflow.

## REFERENCES

- [1] [http://en.wikipedia.org/wiki/Sprouts\\_game](http://en.wikipedia.org/wiki/Sprouts_game)
- [2] <http://www.geocities.com/chessdp>, "World Game of Sprouts Association"
- [3] Mortenson, Michael E. Geometric Modeling, John Wiley & Sons, New York (1997), 523 pages, ISBN 978-0471129578.
- [4] "Experiments with Area Preserving Smoothing of Planar Curves" Student projects from Computer Graphics 6491 - Fall 2005 course at the College of Computing, Georgia Tech, [http://www-static.cc.gatech.edu/computing/classes/AY2006/cs6491\\_fall/Various/p1.htm](http://www-static.cc.gatech.edu/computing/classes/AY2006/cs6491_fall/Various/p1.htm)
- [5] Arvo, J., Novins, K., "Appearance-preserving manipulation of hand-drawn graphs, Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, November 29-December 02, 2005, Dunedin, New Zealand
- [6] Lutterkort, D., Peters, J., "Smooth paths in a polygonal channel," Proceedings of the fifteenth annual symposium on Computational geometry, p.316-321, June 13-16, 1999, Miami Beach, Florida, United States