Vol. 7, No. 2: 9–13

Fast Polygon Area and Newell Normal Computation

Daniel Sunday
John Hopkins University Applied Physics Laboratory

Abstract. The textbook formula for the area of an n-vertex two-dimensional polygon uses $2n + 1$ multiplications and $2n - 1$ additions. We give an improved formula that uses $n + 1$ multiplications and $2n - 1$ additions. A similar formula is derived for a three-dimensional planar polygon where, given the unit normal, the textbook equation cost of $6n + 4$ multiplications and $4n + 1$ additions is reduced to $n + 2$ multiplications and $2n - 1$ additions. Our formula also speeds up Newell's method to compute a robust approximate normal for a nearly planar three-dimensional polygon, using $3n$ fewer additions than the textbook formula. Further, when using this method, one can get the polygon's planar area as equal to the length of Newell's normal for a small additional fixed cost.

1. Introduction

We present a fast method for computing polygon area that was first posted on the author's website (www.geometryalgorithms.com), but has not been previously published. It is significantly faster than the textbook formulas for both two-dimensional and three-dimensional polygons. Additionally, this method can be used to speed up computation of Newell's normal for a three-dimensional polygon. Complete source code is available on the website listed at the end of this paper.

2. The Two-Dimensional Case

For a two-dimensional polygon with n vertices, the standard textbook formula for the polygon's signed area [O'Rourke 98] is:

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i).$$

Each term in the sum involves two multiplications and one subtraction; summing up all the terms take another $n-1$ additions, and the division by two brings the total to $2n+1$ multiplications and $2n-1$ additions. (This ignores the loop's $n-1$ increments of i, as do all subsequent counts.)

With a little algebra, this formula can be simplified. We extend the polygon array by defining $x_n = x_0$ and $x_{n+1} = x_1$, and similarly for y. With these definitions, we get:

$$\begin{aligned} 2A &= \sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) \\ &= \sum_{i=0}^{n-1} x_i y_{i+1} - \sum_{i=0}^{n-1} x_{i+1} y_i \\ &= \sum_{i=1}^{n} x_i y_{i+1} - \sum_{i=1}^{n} x_i y_{i-1} \\ &= \sum_{i=1}^{n} x_i (y_{i+1} - y_{i-1}) \end{aligned}$$

where the third equality comes from recognizing that $x_0 y_1$ is the same as $x_n y_{n+1}$ in the first sum, and index shifting in the second sum. Evaluating the final sum takes n multiplications and $2n-1$ additions, and dividing by two to get the area brings the total to $n+1$ multiplications and $2n-1$ additions.

Using arrays with the first two vertices duplicated at the end, example C code would be:

```
 double findArea(int n, double x[], double y[]) {
    int i;
    double sum = 0.0;
    for (i=1; i <= n; i++) {
        sum += (x[i] * (y[i+1] - y[i-1]));
    }
    return (sum / 2.0);
}
```

By duplicating vertices at the end, we have removed the need to do modular arithmetic calculations on the indices, another source of efficiency. Although a good optimizing compiler will produce fast runtime code from indexed array computations, explicitly efficient C++ code using pointers is given on the website at the end of this paper.

An alternate implementation technique [O'Rourke 98], useful when the vertices of a polygon can be added or deleted dynamically, is to use a circular, doubly-linked list of vertex points beginning at V_{start}. Then, efficient code could be:

```
double findArea(Vertex* Vstart) {
    double sum = 0.0;
    Vertex *cV, *pV, *nV; // vertex pointers
    for (cV = Vstart; ; cV = nV) {
           pV = cV->prev; // previous vertex
           nV = cV->next; // next vertex
           sum += (cV->x * (nV->y - pV->y));
           if (nV == Vstart) break;
    }
    return (sum / 2.0);
}
```

3. The Three-Dimensional Case

In three dimensions, the standard formula for the area of a planar polygon [Goldman 94] is,

$$A = \frac{\mathbf{n}}{2} \cdot \sum_{i=0}^{n-1} V_i \times V_{i+1},$$

where the $V_i = (x_i, y_i, z_i)$ are the polygon's vertices, $\mathbf{n} = (nx, ny, nz)$ is the unit normal vector of its plane, and the signed area A is positive when the polygon is oriented counterclockwise around $\mathbf{n}$. This computation uses $6n + 4$ multiplications and $4n + 1$ additions. We improve this by using the fact that projecting the polygon to two dimensions reduces its area by a constant factor; for example, projection onto the xy-plane reduces the area by a factor of nz. Then we can use the two-dimensional formula above to compute the projected area and divide by nz to get the unprojected area, i.e., we simply compute:

```
Axy = findArea(n, x, y);
Area3D = Axy / nz;
```

If nz is nearly zero, it is better to project onto one of the other planes. If we project onto the zx- or yz-plane, we have to use the formulas:

```
Area3D = Azx / ny; // where Azx = findArea(n, z, x), or
Area3D = Ayz / nx; // where Ayz = findArea(n, y, z)
```

respectively. Thus, one should find the largest of $|nx|$, $|ny|$, and $|nz|$ and use the computation excluding that coordinate. Since the smallest possible value for this is $1/\sqrt{3}$, the largest factor by which one multiplies is $\sqrt{3}$; hence we lose, at most, that factor in precision of the answer and possible overflow is limited. Given a unit normal, this method uses only $n + 2$ multiplications, $2n - 1$ additions, and a small overhead choosing the component to ignore.

As a side benefit, we can use exactly the same trick to compute the values in Newell's robust formula [Tampieri 92] for the approximate normal of a nearly planar three-dimensional polygon:

```
double nwx = Ayz;
double nwy = Azx;
double nwz = Axy;
```

We call this normal vector, $\mathbf{nw} = (nwx, nwy, nwz)$, with the projected areas as its components, the *Newell normal* of the polygon. In the most efficient possible version, the computations of Ayz, Azx, and Axy could all be done in a single loop, amortizing the cost of the loop indexing. The resulting computation uses $3n + 3$ multiplications and $6n - 3$ additions, which is $3n$ fewer additions than the textbook Newell formula.

For a further side benefit, while computing the Newell normal, one can quickly recover the polygon's planar area at the same time. In fact, this area is exactly equal to the length $nlen$ of the Newell normal, since $Area3D = Axy/(nwz/nlen) = (Axy * nlen)/Axy = nlen$. Thus, the polygon area is gotten for the small additional cost of three multiplies, two adds, and one square root needed to compute $nlen$. Finally, just three more divides by $nlen$ converts the Newell normal to a unit normal.

Acknowledgments. I'd like to thank both the reviewer and editor for feedback that greatly simplified this paper's presentation, and also for suggesting the application to Newell's method.

References

[Goldman 94] Ronald Goldman. "Area of Planar Polygons and Volume of Polyhedra." In *Graphics Gems II*, pp. 170–171 edited by J. Arvo, Morgan Kaufmann, 1994.

[O'Rourke 98] Joseph O'Rourke. *Computational Geometry in C, Second Edition.* New York: Cambridge University Press, 1998.

[Tampieri 92] Filippo Tampieri. "Newell's Method for Computing the Plane Equation." In *Graphics Gems III*, pp. 231–232, Academic Press, 1992.

Web Information:

C++ source code for all three algorithms is available at:
http://www.acm.org/jgt/papers/Sunday02

Daniel Sunday, Johns Hopkins University, Applied Physics Laboratory, 11100 Johns Hopkins Road, Laurel, MD 20723 (dan.sunday@jhuapl.edu)

Received March 19, 2002; accepted August 9, 2002.

New Since Original Publication

For some applications, such as when modifying legacy code with fixed array sizes, it is not always easy to go back and make two redundant data points in the array by repeating the first two points. But, since only the first and last points of the array require computing array indices modulo n and since the sum must be initialized anyway, a slight modification breaks those two special case triangles out of the loop. The same number of calculations is required. This eliminates any data point redundancy while retaining the performance of the method. The revised code for a two-dimensional polygon is as follows. (Note: I'd like to thank Robert D. Miller, East Lansing, Michigan for pointing this out to me.)

```
// return the signed area of a 2D polygon (x[],y[])
inline double
findArea(int n, double *x, double *y)       // 2D n-vertex polygon
{
    // Assume that the 2D polygon's vertex coordinates are
    // stored in (x[0], y[0]), ..., (x[n-1], y[n-1]),
    // with no assumed vertex replication at the end.

    // Initialize sum with the boundary vertex terms
    double sum = x[0] * (y[1] - y[n-1]) + x[n-1] * (y[0] - y[n-2]);

    for (int i=1; i < n-1; i++) {
        sum += x[i] * ( y[i+1] - y[i-1] );
    }
    return (sum / 2.0);
}
```

Current Contact Information:

Daniel Sunday, Johns Hopkins University, Applied Physics Laboratory, 11100 Johns Hopkins Road, Laurel, MD 20723 (dan.sunday@jhuapl.edu)