Computing the Barycentric Coordinates of a Projected Point (Preprint of an Article in the Journal of Graphics Tools)

Wolfgang Heidrich

The University of British Columbia

Abstract

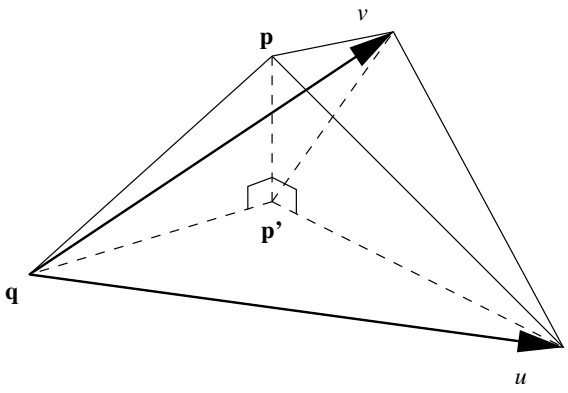
An efficient algorithm is described for computing the barycentric coordinates of the projection of a point into the plane of a triangle. The method requires no square roots or conditionals, and only one floating point division, making it suitable for both CPU and GPU implementations.

1 Introduction

A number of applications in computer graphics and related fields require the computation of the barycentric coordinates for the projection $\mathbf{p}'$ of a point $\mathbf{p}$ into the plane of a triangle T. Examples include interference and/or collision detection, as well as the computation of distance fields. Typically, the projection operation and the computation of the barycentric coordinates are handled separately. It turns out, however, that the two tasks can be combined into a single efficient and elegant algorithm.

2 Method

In the following, we assume that the triangle T is given as a point $\mathbf{q}$ and two edge vectors $\vec{u}$ and $\vec{v}$ (see figure). The method proposed here is based on the familiar area ratios, i.e. $b_0 = A_0/A$, $b_1 = A_1/A$, and $b_2 = A_2/A$, where $A := \text{area}(T)$, and A_0, A_1, and A_2 are the *signed* areas of the triangles formed by $\mathbf{p}'$ and one of the edges of T: $A_0 := \text{area}(\Delta(\mathbf{p}', \mathbf{q}+\vec{u}, \mathbf{q}+\vec{v}))$, $A_1 := \text{area}(\Delta(\mathbf{q}, \mathbf{p}', \mathbf{q}+\vec{v}))$, and $A_2 = \text{area}(\Delta(\mathbf{q}, \mathbf{q}+\vec{u}, \mathbf{p}'))$.



The normal $\vec{n}$ of T is given by $\vec{n} := \vec{u} \times \vec{v}$. The area A of T is one half the length of this normal, or

$$4A^2 = (\vec{n} \cdot \vec{n}). \tag{1}$$

For computing the barycentric coordinates of $\mathbf{p}$, let us initially assume that it already lies in the plane of T, i.e. $\mathbf{p}' = \mathbf{p}$.

The normal $\vec{n}_2$ of triangle $\Delta(\mathbf{q}, \mathbf{q}+\vec{u}, \mathbf{p})$ is $\vec{n}_2 := \vec{u} \times \vec{w}$, where $\vec{w} := \mathbf{p} - \mathbf{q}$. The signed area A_2 is then half the length of n_2, and it is positive if n_2 points in the same direction as n (i.e. if $\mathbf{p}$ is inside with respect to the edge), and negative if it points in the opposite direction (i.e. if $\mathbf{p}$ is outside with respect to the edge). A_2 can thus be computed as

$$2A_2 = \frac{(\vec{n}_2 \cdot \vec{n})}{||\vec{n}||} = \frac{(\vec{n}_2 \cdot \vec{n})}{2A}. \tag{2}$$

For the barycentric coordinate b_2, we thus get

$$b_2 = \frac{A_2}{A} = \frac{2A_2}{2A} = \frac{(\vec{n}_2 \cdot \vec{n})}{4A^2} = \frac{(\vec{n}_2 \cdot \vec{n})}{(\vec{n} \cdot \vec{n})}, \tag{3}$$

and likewise $b_1 = (\vec{n}_1 \cdot \vec{n})/(\vec{n} \cdot \vec{n})$ and $b_0 = 1 - b_1 - b_2$, where $\vec{n}_1 := \vec{w} \times \vec{v}$.

The key observation in computing the barycentric coordinates for a *projection* $\mathbf{p}'$ in case $\mathbf{p}' \neq \mathbf{p}$ is that Equation 3 still holds: we the get

$$b_2 = \frac{A_2}{A} = \frac{\text{area}(\Delta(\mathbf{q}, \mathbf{q}+\vec{u}, \mathbf{p}'))}{A} = \frac{\text{area}(\Delta(\mathbf{q}, \mathbf{q}+\vec{u}, \mathbf{p}))}{A} \cdot \cos(\vec{n}, \vec{n}_2) = \frac{\frac{1}{2}||\vec{n}_2||}{\frac{1}{2}||\vec{n}||} \cdot \frac{(\vec{n} \cdot \vec{n}_2)}{||\vec{n}|| \cdot ||\vec{n}_2||} = \frac{(\vec{n} \cdot \vec{n}_2)}{(\vec{n} \cdot \vec{n})},$$

where $\vec{n}_2$ is the normal of $\Delta(\mathbf{q}, \mathbf{q}+\vec{u}, \mathbf{p})$ as above.

It turns out that this result is a specialization of an observation proven by Warren, which relates barycentric coordinates in convex polyhedron the barycentric coordinates on its faces [2]. In our case, the polyhedron is the tetrahedron formed by the four points $\mathbf{q}$, $\mathbf{q}+\vec{u}$, $\mathbf{q}+\vec{v}$, and $\mathbf{p}$, which simplifies the situation as just described. The complete algorithm is therefore as follows:

```
projectedBarycentricCoord( Point p, Point q, Vector u, Vector v, double *b )
{
  Vector n= cross( u, v );
  double oneOver4ASquared= 1.0 / dot( n, n );
  Vector w= p - q;
  b[2]= dot( cross( u, w ), n ) * oneOver4ASquared;
  b[1]= dot( cross( w, v ), n ) * oneOver4ASquared;
  b[0]= 1.0 - b[1] - b[2];
}
```

3 Discussion

The alternative to our approach is to first compute the projection $\mathbf{p}'$ as $\vec{w} - (\vec{w} \cdot \vec{n})/(\vec{n} \cdot \vec{n}) \cdot \vec{n}$. Including the computation of $\vec{n}$ and $\vec{w}$ as above, this requires 28 floating point operations (multiplies and adds) plus one division. By far the fastest known method to subsequently compute the barycentric coordinates is then an orthogonal projection of T and $\mathbf{p}'$ along one of the major axes, followed by solving the 2D problem (see, for example, [1]). The 2D problem requires a mere 13 multiplies and adds, but also a second division. This method suffers from two sources of numerical imprecisions that can accumulate. The first source is the computation of $\mathbf{p}'$ itself, which may not produce a point that truly lies in the plane of T. The second source is the 2D projection along a major axis. In order to minimize numerical problems in that part, the projection direction has to correspond to the largest component of $\vec{n}$. This requires additional comparisons and absolute value computations, as well as data-dependent branching. The source code for this approach is about 3-4 times as long as our algorithm due to the handling of the different cases.

By comparison, the algorithm presented here requires 49 multiplies and adds, plus one floating point division. No other operations (integer operations, comparisons, etc.) are required, nor are branches or conditional jumps. This is important in the light of the increasing cost of data-dependent branching on modern CPUs. Also, this feature makes the method suitable for GPUs, which have limited branching capabilities to begin with. In particular, the algorithm can be implemented in both vertex and pixel shaders on all reasonably modern GPUs.

References

[1] S. C. Dollins. Handy mathematics facts for graphics. http://www.cs.brown.edu/people/scd/facts.html.

[2] J. Warren. Barycentric coordinates for convex polytopes. *Advances in Computational Mathematics*, 6(2):97–108, 1996.