Barycentric Quad Rasterization

Jules Bloomenthal
Seattle University

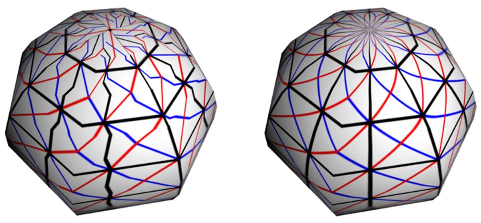


Figure 1. Quad rasterization: error and correction.

Abstract

When a quadrilateral is rendered as two triangles, a C^1 discontinuity can occur along the dividing diagonal. In 2004, Hormann and Tarini used generalized barycentric coordinates to eliminate the discontinuity. The present paper provides an implementation using a geometry shader, unavailable in 2004, and provides additional examples of the barycentric technique.

1. Introduction

A quadrilateral (or *quad*) is a common design element and is scan-line rendered by graphics APIs as two triangles. This *quad split* can introduce a C^1 discontinuity along the dividing diagonal. Although shading continuity has been a concern since the 1970s, the quad split discontinuity was not fully addressed until Hormann and Tarini [2004] employed *generalized barycentric coordinates* to interpolate vertex attributes.

This paper extends the work of Hormann and Tarini in a few ways.

Their paper states, "any attribute interpolation inside the quad will be only C^0 continuous at the chosen diagonal." This is not strictly true, and the present paper describes the conditions that give rise to the discontinuity and affect its severity.

Hormann and Tarini's paper predates the geometry shader, and consequently, their method was partially implemented on the CPU. The present paper provides a GLSL implementation in the form of geometry and pixel shaders.

Lastly, the present paper provides additional, clarifying examples of improved rendering when the C^1 discontinuity is eliminated.

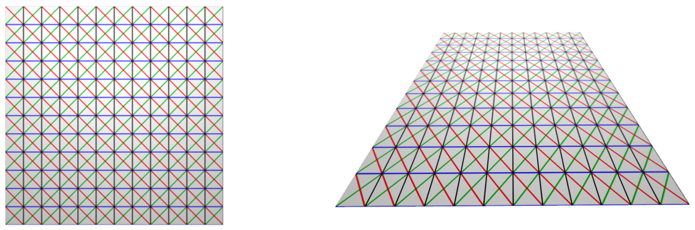


Figure 2. Texture-mapped square rendered as two triangles. Left: Untransformed. Right: Oblique, perspective view.

2. C^1 Discontinuity and Affinity

When a quadrilateral is rasterized as separate triangles, the results may be artifact-free, as with the textured square in Figure 2.

The geometry (the four points that define the square) and the texture (the four corresponding points on the texture map, i.e., *texture coordinates*) have the same, square shape. In perspective, the geometry appears as a trapezoid, but the texture-mapped image remains artifact-free.

If the geometry is a trapezoid, however, the rasterization has a C^1 discontinuity. Why does the trapezoid in Figure 3 have an artifact, but the one in Figure 2 does not?

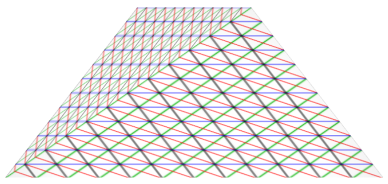


Figure 3. Texture-mapped trapezoid rendered as two triangles.

Consider a triangle defined by three screen points and an associated triangle of three texture points. It is well known that a unique, affine transformation maps the screen triangle to the texture triangle. This guarantees that the straight rows of pixels produced when rasterizing the screen triangle correspond with straight rows of samples in texture space, as shown in Figure 4.

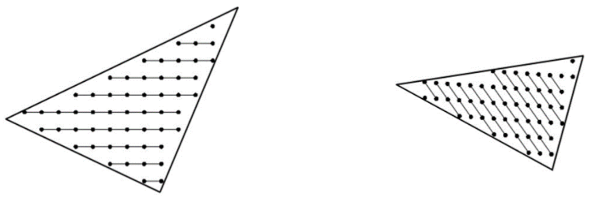


Figure 4. Triangle rasterization. Left: Pixels in screen space. Right: Corresponding samples in texture space.

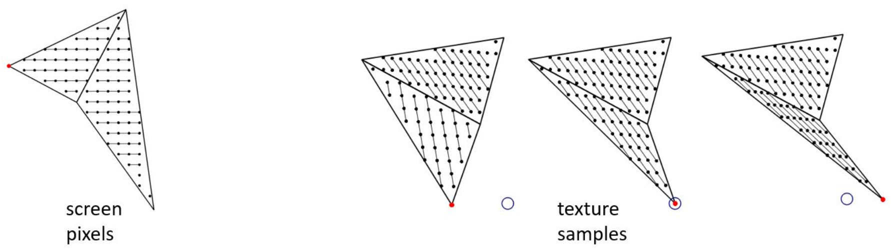


Figure 5. Effect on texture gradient of placement of fourth texture point (ideal location shown as circle).

If a fourth screen point and a fourth texture point are added, a second screen triangle and second texture triangle are formed. The rows of pixels in the two screen triangles obviously align with each other. But, for the rows of samples to align between the two texture triangles, the affine transformation that relates the first screen triangle to the first texture triangle must be applied to the new screen location to determine the new texture location. Placing the new texture point elsewhere produces a discontinuity in the texture gradient, as shown in Figure 5.

2.1. Severity of Discontinuity

An affine relationship between texture and geometry is not always possible. For example, the longitude/latitude parameterization and Mercator map typical of a textured sphere imply that a quad becomes increasingly trapezoidal as it approaches a pole, whereas its texture domain remains rectangular. (See Figure 6.) As the foreshortening increases, so does the severity of the C^1 texture discontinuity.

At the pole, the upper edge of quad *Q2* is degenerate. When the quad is divided, the triangle along the upper edge covers no pixels and, thus, its corresponding texture (which should contribute 50% of the quad texture) is ignored. (See Figure 7.)

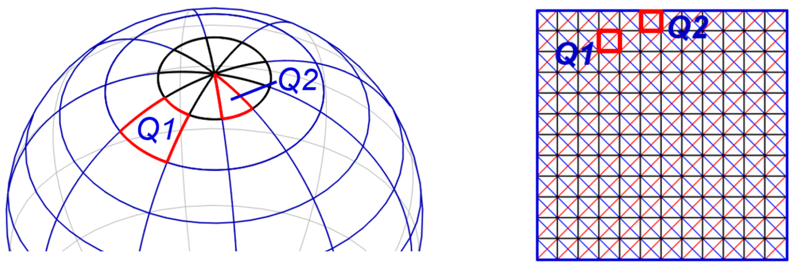


Figure 6. Quads on a sphere. Left: Geometric space. Right: Texture space.

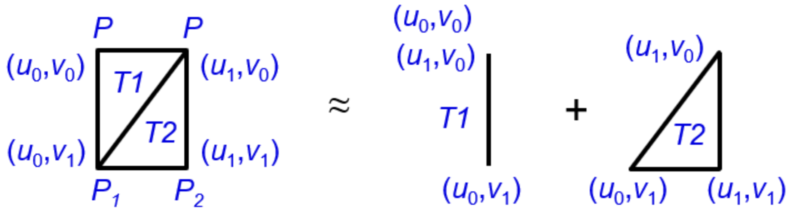


Figure 7. Split, degenerate quad. Left: Geometric space. Right: Texture space.

3. Generalized Barycentric Coordinates

To eliminate quad split C^1 discontinuity, Hormann and Tarini computed, for each pixel within the quad, the generalized barycentric coordinates of the pixel with respect to the screen positions of the four quad corners. These four-dimensional coordinates serve as weighting coefficients to interpolate vertex attributes (uv-coordinates or other shading attributes).

The generalized barycentric coordinates are *mean value coordinates*, developed by Floater [2003]. Their calculation is shown in Figure 8.

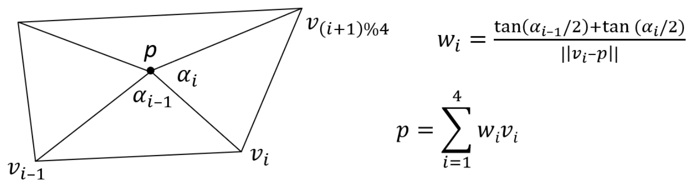

$v_{(i+1)\%4}$
p
α_i
α_{i-1}
v_{i-1}
v_i
$w_i = \frac{\tan(\alpha_{i-1}/2)+\tan(\alpha_i/2)}{||v_i-p||}$
$p = \sum_{i=1}^{4} w_i v_i$

Figure 8. Generalized barycentric coordinates (unnormalized mean value coordinates).

These coordinates yield a C^∞-continuous interpolation within the quad and linear interpolation along its edges. This holds for concave, nonplanar, or self-intersecting quads, and it extends to n-sided polygons. As shown in Figure 9, C^1 discontinuity is eliminated and quad edges match neighboring quads.

To compute the generalized barycentric coordinates, the pixel shader must access all four vertex screen locations as well as the attributes of the four vertices. This can be arranged with a geometry shader, provided in the code listing in the appendix.

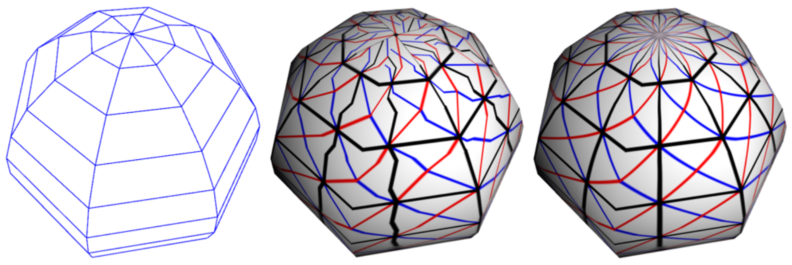


Figure 9. Textured sphere. Middle: Quad split. Right: Barycentric rasterization, with longitudinal and latitudinal resolutions of 8 and 16.

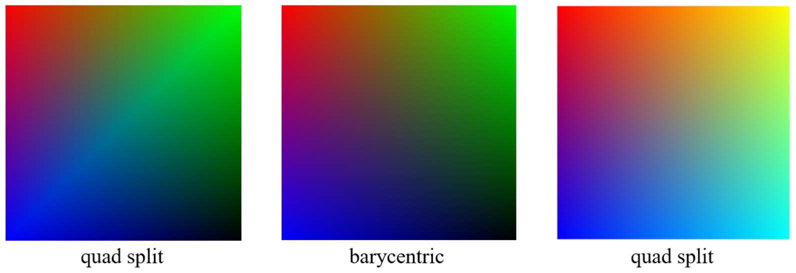


Figure 10. Effect on colors. Left, middle: Blue, red, green, and black. Right: Blue, red, yellow, and cyan.

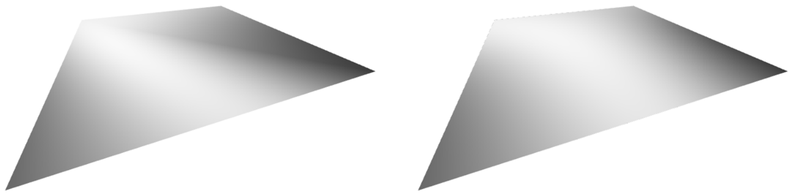


Figure 11. Effect on normals. Left: Quad split. Right: Barycentric.

3.1. Et Tu, Shading?

Although the quad split discontinuity is readily observed with texture mapping, it is also visible as a consequence of color or lighting.

In Figure 10, the left and middle squares interpolate blue, red, green, and black; these colors do not form a square in color space. When rendered as a quad split, a C^1 discontinuity results. The square on the right interpolates blue, red, yellow, and cyan, which do form a square in color space; when rendered as a quad split, no discontinuity results.

If normals at quad corners differ significantly, the quad split rendering can appear to introduce a bend along the diagonal, as in Figure 11.

3.2. Resolution

For the quad split, the smaller the screen-space projection, the less severe is the discontinuity. The examples in Figure 12 compare quad split with the barycentric method at longitudinal/latitudinal resolutions of 30/20 (top) and 100/40 (bottom).

The artifacts worsen toward the pole; they can be reduced with increased density of latitude toward the poles, while maintaining the same quad count. The closeup in Figure 13 reveals that, even with high quad count and increased density at the pole, the barycentric rasterization is discernibly better.

3.3. Inverse Bilinear Interpolation

Vertex attribute coefficients may also be computed with *inverse-bilinear interpolation* [Quilez 2010]. Although generally yielding good results, the inversion can fail (the

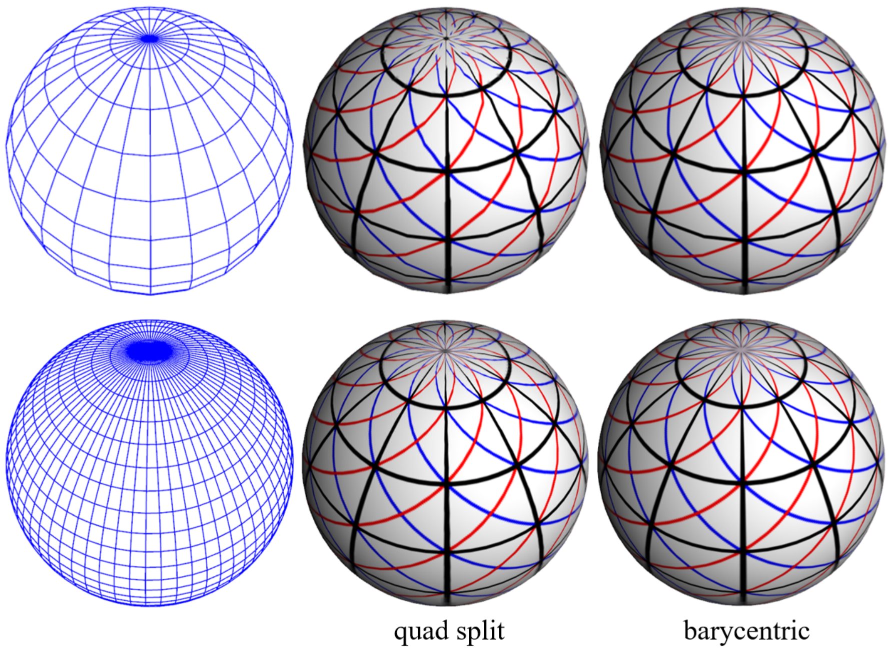


Figure 12. Resolution comparison. Top: 600 quads. Bottom: 4000 quads.

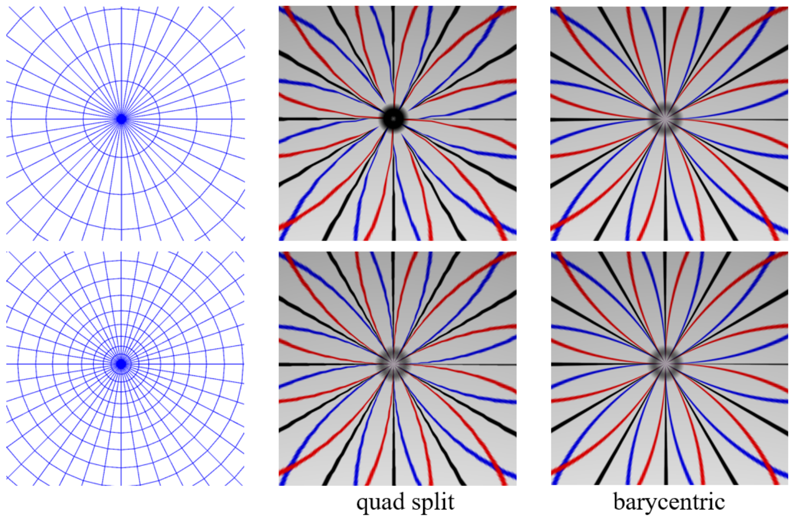


Figure 13. Pole closeup. Top: Uniform latitude spacing. Bottom: Denser at pole.

quadratic discriminant becoming negative) if the quad intersects the camera plane. The undefined region, shown in Figure 14 in tan, grows in size as the quad moves to the rear.

Support for quads with points behind the eye was an explicit objective of the paper by Hormann and Tarini [2004].

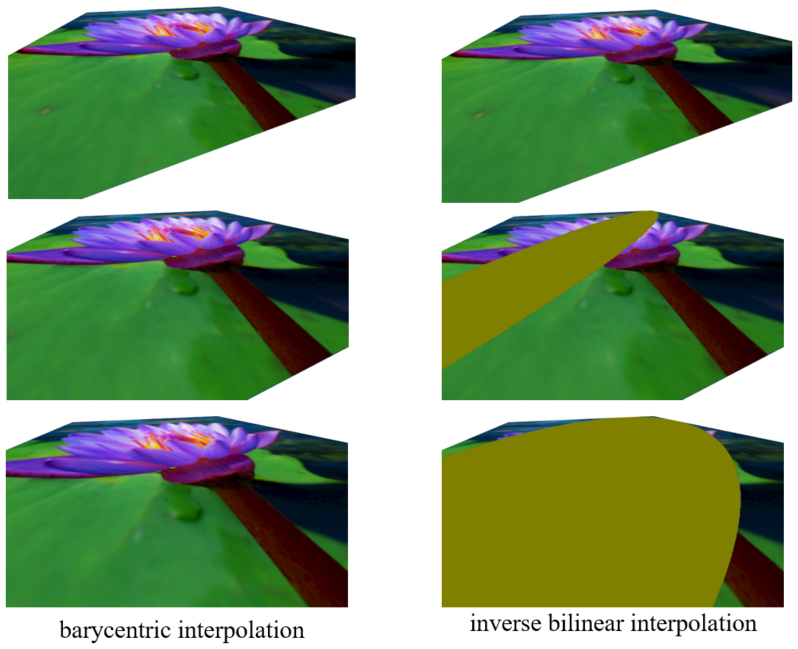


Figure 14. Effect of points behind the eye.

3.4. Nonplanar Quads

For nonplanar quadrilaterals, including “bow-tie” quads (see Figure 15) that self-intersect on the screen, barycentric rasterization is free of degeneracies, except at the point of self-intersection.

The rasterization is, however, view dependent. In their paper, Hormann and Tarini [2004] noted that this is more visible for textured, nonplanar quads that are large on the screen, but much less observable for small quad projections and, in any case, superior to the quad split. Inverse bilinear interpolation is subject to discontinuity and degeneracy.

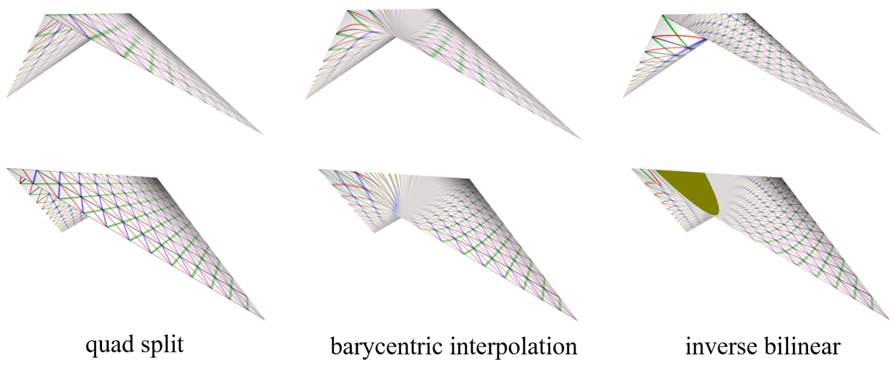


Figure 15. Bow-tie quad. Top and bottom rows show the same quad, but differently oriented.

4. Conclusion

Compared with the quad split, the use of barycentric coordinates yields improved quality for geometry/texture disaffinity, but requires more than 100 arithmetic operations per pixel.

One practical strategy is to invoke barycentric quad rasterization only for quads with geometric/texture affinity below some subjective threshold.

A measure of affinity is to compute, for each quad corner, the (triangle) barycentric coordinates with respect to the other three corners. These should be compared against the barycentric coordinates of the corresponding texture location with respect to the other three texture locations. If there is affinity, these two sets of barycentric coordinates will match. This test is described at the end of the appendix.

Acknowledgements

The author is grateful for discussions with Ken Shoemake and for comments by the JCGT reviewers.

A. Source Code

To marshal needed inputs for the pixel shader, four vertices are sent (as an array, for each quad) to a GLSL geometry shader, in Listing 1. This is initiated with an OpenGL call to `glDrawArrays` or `glDrawElements` with `GL_LINES_ADJACENCY` as the mode (these subroutines do not accept `GL_QUADS` as a mode, but `GL_LINES_ADJACENCY` achieves the same result).

The perspective-space coordinates of the vertices, `gpos0–3`, and the texture coordinates, `guv0–3`, are then sent to the rasterizer (as individual variables, not arrays) and subsequently to the pixel shader for use in computing uv-coordinates.

Other attributes may be similarly computed. For example, barycentric rasterization for color can be performed by replacing `guv0–3` in the geometry and pixel shaders with `gcolor0–3`.

In their paper, Hormann and Tarini [2004] provide pseudocode to compute generalized barycentric coordinates. An implementation is given in the pixel shader in Listing 2.

In `UV()`, the built-in `gl_FragCoord` input to the pixel shader is converted to perspective-space coordinates, which are used to compute the barycentric weights with respect to the quad corners, `gpos0–3`.

The weights are corrected for perspective by the division by `gpos0–4.w` when computing `v[]` in `BarycentricWeights()` and `f[]` in `UV()`. This ensures that geometrically straight lines remain straight, as they should. The perspective-corrected weights are then applied to `guv0–3` to determine the uv-coordinates for the pixel.

```
#version 330
layout (lines_adjacency) in;                      // 1 quad in
layout (triangle_strip, max_vertices = 6) out;   // 2 triangles out
in vec3 vpoint[4], vnormal[4];
in vec2 vuv[4];
out vec3 gpoint, gnormal;
// Specify flat to disable interpolation by rasterizer.
flat out vec4 gpos0, gpos1, gpos2, gpos3;
flat out vec2 guv0, guv1, guv2, guv3;

void EmitPoint(int i, bool provokeFlatOutput) {
    gpoint = vpoint[i];
    gnormal = vnormal[i];
    gl_Position = gl_in[i].gl_Position;
    if (provokeFlatOutput) {                      // Once per tri
        gpos0 = gl_in[0].gl_Position;
        gpos1 = gl_in[1].gl_Position;
        gpos2 = gl_in[2].gl_Position;
        gpos3 = gl_in[3].gl_Position;
        guv0 = vuv[0];
        guv1 = vuv[1];
        guv2 = vuv[2];
        guv3 = vuv[3];
    }
    EmitVertex();                                 // Send vertex.
}

void main() {
    EmitPoint(0, false); EmitPoint(1, false); EmitPoint(3, true);
    EndPrimitive();                               // Triangle 1
    EmitPoint(1, false); EmitPoint(2, false); EmitPoint(3, true);
    EndPrimitive();                               // Triangle 2
}
```

Listing 1. Geometry shader.

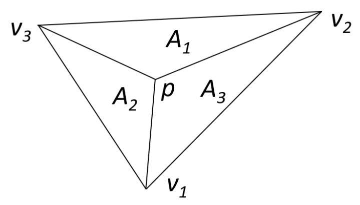
Double precision is used in `BarycentricWeights()` and `UV()` to minimize quantization artifacts. Hormann and Tarini noted that the terms `A`, `D`, and `r` are computable with forward differencing.

A test for affinity between quad geometry and texture relies on the standard definition of barycentric coordinates for a triangle, shown in Figure 16.

Listing 3 computes the barycentric coordinates of each texture point of a quad with respect to the other three texture points, comparing them to the barycentric coordinates of each geometric-space point with respect to the other three geometric-space points.

```
#version 430
in vec3 gpoint, gnormal;
flat in vec4 gpos0, gpos1, gpos2, gpos3;     // Quad points
flat in vec2 guv0, guv1, guv2, guv3;         // Quad uvs
uniform sampler2D image;
uniform vec4 vp;                             // Determine ndc.
uniform vec3 light;

double Area(dvec2 a, dvec2 b) { return(a.x*b.y)-(a.y*b.x); }

dvec4 BarycentricWeights(dvec2 p) {
    // From pseudocode in [Hormann and Tarini 2004]
    double w[4] = { gpos0.w, gpos1.w, gpos2.w, gpos3.w };
    double r[4], t[4], u[4];
    dvec2 v[4] = { dvec2(gpos0)/w[0], dvec2(gpos1)/w[1],
                   dvec2(gpos2)/w[2], dvec2(gpos3)/w[3] }, s[4];
    for (int i = 0; i < 4; i++) {
        s[i] = v[i]-p;
        r[i] = length(s[i])*sign(w[i]);
    }
    for (int i = 0; i < 4; i++) {
        double A = Area(s[i], s[(i+1)%4]);
        double D = dot(s[i],s[(i+1)%4]);
        t[i] = (r[i]*r[(i+1)%4]-D)/A;
    }
    for (int i = 0; i < 4; i++)
        u[i] = (t[(i+3)%4]+t[i])/r[i];
    return dvec4(u[0],u[1],u[2],u[3])/(u[0]+u[1]+u[2]+u[3]);
}

vec2 UV() {
    // Set normalized dev coords; compute barycentric weights.
    dvec2 ndc = dvec2(2*(gl_FragCoord.x-vp[0])/vp[2]-1,
                      2*(gl_FragCoord.y-vp[1])/vp[3]-1);
    vec4 wt = BarycentricWeights(ndc);
    // Do persp div; return uvs.
    float f0 = wt[0]/gpos0.w, f1 = wt[1]/gpos1.w,
          f2 = wt[2]/gpos2.w, f3 = wt[3]/gpos3.w;
    return (f0*guv0+f1*guv1+f2*guv2+f3*guv3)/(f0+f1+f2+f3);
}

void main() {
    // Compute shading intensity.
    vec3 N = normalize(gnormal), L = normalize(light-gpoint);
    float d = max(0, dot(N, L)), intensity = clamp(.05+d, 0, 1);
    // Compute uv-coordinates and texture.
    vec2 uv = UV();
    gl_FragColor = vec4(intensity*texture(image, uv).rgb, 1);
}
```

Listing 2. Pixel shader.



$$w_i = \frac{A_i}{A_1+A_2+A_3}$$

$$p = \sum_{i=1}^{3} w_i v_i$$

Figure 16. Barycentric coordinates for a triangle.

```
vec3 Cross(vec3 a, vec3 b) {
   return vec3(a.y*b.z-a.z*b.y, a.z*b.x-a.x*b.z, a.x*b.y-a.y*b.x);
}

float Cross(vec2 a, vec2 b) { return vec2(a.x*b.y)-(a.y*b.x); }

float Area(vec3 a, vec3 b, vec3 c) {
    vec3 v = Cross(b-a, c-b);
    return v.z > 0? Length(v) : -Length(v);
}

float Area(vec2 a, vec2 b, vec2 c) { return Cross(b-a, c-b); }

template <Type T> bool Bary(T p, T p1, T p2, T p3, vec3 &bary) {
    float a1 = Area(p, p2, p3),
          a2 = Area(p, p3, p1),
          a3 = Area(p, p1, p2), sum = a1+a2+a3;
    if (abs(sum) < FLT_EPSILON) return false;
    bary = vec3(a1, a2, a3)/sum;
    return true;
}

bool AffineMapped(vec3 *pt, vec2 *uv, float tolerance = .01) {
    // Is uv[] an affine image of pt[]?
    for (int i = 0; i < 4; i++) {
        int i1 = mod(i+1, 4), i2 = mod(i+2, 4), i3 = mod(i+3, 4);
        vec3 bpt, buv;
        bool ptOk = Bary(pt[i], pt[i1], pt[i2], pt[i3], bpt);
        bool uvOk = Bary(uv[i], uv[i1], uv[i2], uv[i3], buv);
        if (!ptOk || !uvOk || Length(bpt-buv) > tolerance)
            return false;
    }
    return true;
}
```

Listing 3. Test for affinity.

References

FLOATER, M. S. 2003. Mean value coordinates. *Computer Aided Geometric Design 20*, 1, 19–27. URL: `https://cgvr.informatik.uni-bremen.de/teaching/cg_literatur/barycentric_floater.pdf`. 68

HORMANN, K., AND TARINI, M. 2004. A quadrilateral rendering primitive. In *Graphics Hardware 2004: Eurographics Symposium Proceedings*, The Eurographics Association, 7–14. URL: `https://diglib.eg.org/handle/10.2312/EGGH.EGGH04.007-014`. 65, 70, 71, 72

QUILEZ, I., 2010. Inverse bilinear interpolation. URL: `https://iquilezles.org/www/articles/ibilinear/ibilinear.htm`. 69

Author Contact Information

Jules Bloomenthal
jules@bloomenthal.com

Jules Bloomenthal, Barycentric Quad Rasterization, *Journal of Computer Graphics Techniques (JCGT)*, vol. 11, no. 3, 65–76, 2022
http://jcgt.org/published/0011/03/04/

Received:	2022-01-24		
Recommended:	2022-04-20	Corresponding Editor:	Matt Pharr
Published:	2022-08-24	Editor-in-Chief:	Marc Olano