Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

A Quadrilateral Rendering Primitive

Kai Hormann Marco Tarini

Visual Computing Lab, ISTI / CNR, Pisa

Abstract

The only surface primitives that are supported by common graphics hardware are triangles and more complex shapes have to be triangulated before being sent to the rasterizer. Even quadrilaterals, which are frequently used in many applications, are rendered as a pair of triangles after splitting them along either diagonal. This creates an undesirable C^1-discontinuity that is visible in the shading or texture signal. We propose a new method that overcomes this drawback and is designed to be implemented in hardware as a new rasterizer. It processes a potentially non-planar quadrilateral directly without any splitting and interpolates attributes smoothly inside the quadrilateral. This interpolation is based on a recent generalization of barycentric coordinates that we adapted to handle perspective correction and situations in which a quadrilateral is partially behind the point of view.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—*graphics processors*; I.3.3 [Computer Graphics]: Picture/Image Generation—*display algorithms*; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—*color, shading, shadowing, and texture*

1. Introduction

Quadrilateral meshes are a natural and popular choice for *modelling* many classes of 3D surfaces. For example, they are useful for approximating Bézier patches and to describe rotational surfaces and rectangular height fields. In contrast, current GPUs do not support *rendering* of quadrilaterals directly and whenever a quadrilateral (or *quad*) is displayed, one of several options has to be chosen (see Figure 1).

The most frequently used approach is the **diagonal split** along one of the two diagonals which decomposes the quad into two triangles [SA03]. As a consequence, any attribute interpolation inside the quad will be only C^0-continuous at the chosen diagonal. This C^1-discontinuity is undesirable because it is well visible if, for example, shading is computed from interpolated normals or texture mapping is applied.

A less common technique is to perform **two-fold linear interpolation** with a standard scan-line rasterizer. Attributes are linearly interpolated first along the projected edges and then across each horizontal span from edge to edge [SA03]. Unfortunately, this also creates C^1-discontinuities, namely along the horizontal screen lines that pass through the quad's vertices. These discontinuities are more disturbing because they vary with the viewing direction and even if the camera is just rotated around the z-axis [Duf79].

An effective way to avoid discontinuities is to utilize the **projective map** that maps the quad to the planar unit square and to compute the attributes for a point inside the quad by bilinear interpolation over this standard domain [Duf79]. Although this method interpolates smoothly inside the quad it has several drawbacks: it gives a false impression of depth; it does not interpolate attributes linearly along the edges, thus creating C^0-discontinuities between two neighbouring quads; and it requires the projected quad to be convex.

Another option is to refine the quad into $n \times n$ small quads so as to reduce the artefacts caused by the diagonal split. The positions and attributes of the new vertices can be computed by **bilinear interpolation** or any other subdivision scheme either on-the-fly or by preprocessing. For sufficiently large n, this gives visually smooth results, but since many more vertices and triangles must undergo the vertex processing and triangle setup phase, it can cause a severe performance downgrade, especially for geometry-limited applications.

In this paper we present a new way to render quadrilaterals as atomic primitives that gives results which are often visually similar to the ones obtained by bilinear interpolation and in any case superior to the diagonal split. Moreover, it inherits most of the properties that we are used to from rendering triangles and blends well with the design of current graphics systems.

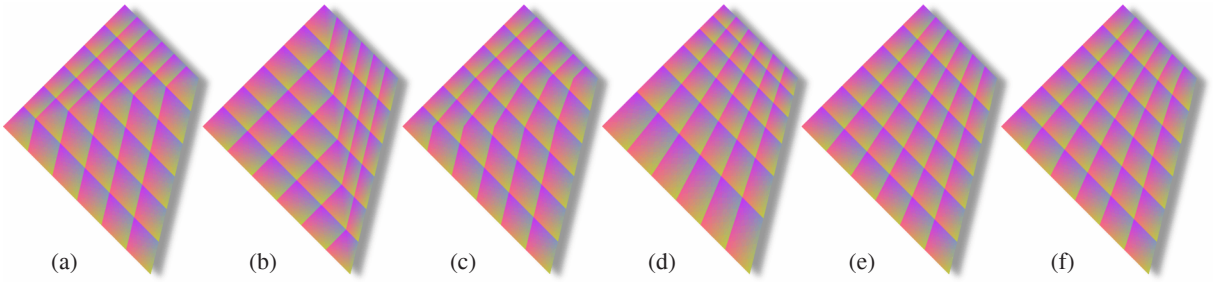


Figure 1: *Texture mapping a regular pattern with different methods onto a flat quad that is seen from above. From left to right: the two diagonal splits (a,b), two-fold linear interpolation (c), projective map (d), bilinear interpolation (e), our method (f).*

Our approach is based on recent theoretical and technological advances. On the one hand, we utilize and extend Floater's *mean value coordinates* [Flo03] which generalize the concept of barycentric coordinates and allow to interpolate attributes smoothly over the interior of a planar quad and linearly along its edges (Section 3). On the other hand, modern GPUs can execute an increased number of complex operations per second. Admittedly, this computational power has mostly been spent on per-vertex and per-fragment processing so far, but it could also be used to implement our method efficiently as a new rasterizer (Section 4). This quad rasterizer would provide a new "tube" for the graphics pipeline between the vertex and the fragment shader that could be used as an alternative to the existing rasterizers for triangles, lines, and points.

2. Main Idea

Our algorithm renders a three-dimensional, potentially non-planar quad $Q = [V_0, V_1, V_2, V_3]$ in the same three steps that are commonly used for triangles:

1. **Vertex shader.** Project the 3D vertices V_i to screen and perform any other per-vertex operation.
2. **Rasterizer.** Create a fragment for every pixel inside the projected 2D quad and use generalized barycentric coordinates to appropriately interpolate the vertex attributes.
3. **Fragment shader.** Process the produced fragments with their interpolated attributes.

The first step is identical to the one that is currently used for other primitives (triangles, lines, and points). A 3D vertex V_i with world coordinates (X_i, Y_i, Z_i) is projected to screen by first multiplying its *homogeneous coordinate vector* with the 4×4 matrix $\mathbf{M}$ that combines the necessary transformations (model-view, perspective, and window-to-viewport),

$$(\tilde{x}_i, \tilde{y}_i, \tilde{z}_i, w_i) = (X_i, Y_i, Z_i, 1)\,\mathbf{M}. \qquad (1)$$

The result is then normalized to obtain the *screen position* $v_i = (x_i, y_i) = (\tilde{x}_i/w_i, \tilde{y}_i/w_i)$ and the *depth value* $z_i = \tilde{z}_i/w_i$. In addition, all the *vertex attributes* that shall be interpolated inside the quad (normals, texture coordinates, etc.) are set up per vertex in this step, possibly by a vertex program.

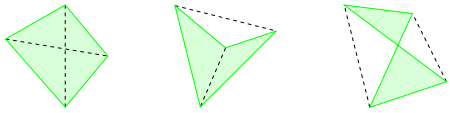


Figure 2: *A quad in front of the camera projects to a convex, concave, or self-intersecting shape in screen space. Dashed lines indicate the quad's diagonals.*

Step two is the core part of our method and although it is similar in spirit, it differs considerably from the triangle analogue. After the four vertices of Q have been projected, we consider the 2D *screen quad* $q = [v_0, v_1, v_2, v_3]$ that can have any of the shapes shown in Figure 2. Finding the interior screen pixels of q and determining their orientation (front- or back-facing) is rather simple and can be done, for example, with a general polygon scan-converter [FvDFH90] or using *edge functions* (Section 4.1). In contrast, interpolating vertex attributes is more delicate and has not been solved satisfactorily, so far.

But why is this interpolation so easy in the case of triangles and yet so hard for quads? The reason is that the underlying concept of barycentric coordinates is trivial for triangles, but it has only recently been generalized to arbitrary convex polygons (see [FHK04] and the references therein). We discovered that the *mean value coordinates* [Flo03], which are one possible generalization, naturally extend to concave and self-intersecting quads (Section 3). We further discuss how they can be adapted to handle perspective correction (Section 3.1) and *exterior quads* (Section 3.2).

The third step proceeds exactly as in current hardware implementations and treats fragments (compute shading, lookup texture, etc.) either with the standard fixed functions or with a user-defined fragment program.

3. Barycentric Coordinates for Quadrilaterals

Given an n-sided planar polygon with vertices $v_0, \ldots, v_{n-1}$, the basic concept of barycentric coordinates is to have for

any point v a set of coordinates $\lambda_i(v)$ with the two properties

$$\sum_{i=0}^{n-1} \lambda_i(v) v_i = v, \tag{2}$$

$$\sum_{i=0}^{n-1} \lambda_i(v) = 1, \tag{3}$$

and to use them for interpolating attributes a_i that are associated with the vertices v_i straightforwardly by

$$a(v) = \sum_{i=0}^{n-1} \lambda_i(v) a_i. \tag{4}$$

It is well-known that the barycentric coordinates for a triangle are uniquely defined by the linear functions

$$\lambda_i(v) = A(v, v_{i+1}, v_{i+2}) / A(v_0, v_1, v_2), \tag{5}$$

where $A(u_0, u_1, u_2)$ denotes the *signed* area of the triangle $[u_0, u_1, u_2]$. Apart from the fact that the attribute interpolation (4) can efficiently be computed for these coordinates, their most important properties are:

- **Lagrange property.** Since $\lambda_i(v_j)$ equals 1 if $i = j$ and 0 otherwise, it follows that vertex attributes are reproduced at the vertices, i.e. $a(v_j) = a_j$.
- **Edge linearity.** The λ_i are linear along the edges of the triangle, so that the attribute interpolation is C^0-continuous across the common edge of two neighbouring triangles.
- **Positivity.** The interpolated attribute $a(v)$ of a point v that lies *inside* the triangle is a *convex combination* of the a_i because in this (and only this) case all $\lambda_i(v)$ are positive.

In order to define barycentric coordinates for arbitrary polygons it is often easier to first construct a set of *homogeneous coordinates* $\mu_i(v)$ that satisfy

$$\sum_{i=0}^{n-1} \mu_i(v)(v_i - v) = 0 \tag{6}$$

and then derive the barycentric coordinates by normalization

$$\lambda_i(v) = \mu_i(v) \Big/ \sum_{j=0}^{n-1} \mu_j(v). \tag{7}$$

These $\lambda_i(v)$ obviously fulfil Equations (2) and (3) and a general construction of homogeneous coordinates $\mu_i(v)$ has been discussed in [FHK04] in the case of *convex* polygons. The critical part about using their construction for our purposes is the extension to *concave* or *self-intersecting* quads. For most

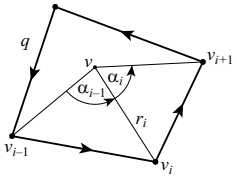


Figure 3: *Notation used for mean value coordinates.*

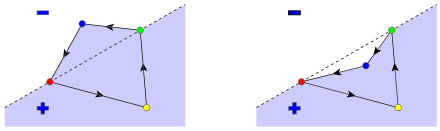


Figure 4: *For a quad with positive orientation, the homogeneous coordinate with respect to the blue vertex is positive in the shaded region and likewise for the other vertices.*

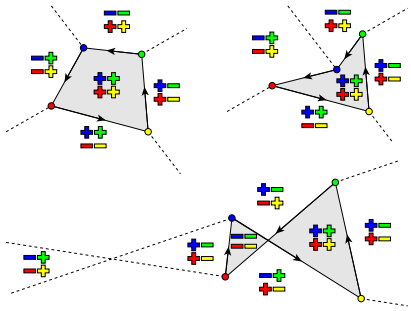


Figure 5: *The homogeneous coordinates all have the same sign inside the quad and the sign depends on the local orientation. Outside the quad, the signs are mixed.*

choices of functions μ_i there exist points v in the interior of such quads for which the denominator in (7) is zero, so that the $\lambda_i(v)$ are not well-defined. This happens, for example, in the case of *Wachspress coordinates* [Wac75, MLBD02].

One choice that does not suffer from this drawback, and has the additional advantage of a modest computational cost (Section 4.2), is the set of *mean value coordinates* [Flo03]. They are defined by

$$\mu_i(v) = \frac{\tan(\alpha_{i-1}(v)/2) + \tan(\alpha_i(v)/2)}{r_i(v)}, \tag{8}$$

where $\alpha_i(v)$ is the *signed* angle in the triangle $[v, v_i, v_{i+1}]$ at v and $r_i(v) = \|v - v_i\|$ is the distance between v and the i-th vertex (see Figure 3).

It follows directly from the definition (8) that if the screen quad q is positively oriented (i.e. counterclockwise), then μ_i is positive inside q and on the left side of the (oriented) line through v_{i-1} and v_{i+1} outside the quad (see Figure 4). If q is negatively oriented, then all signs are reversed, but in any case the key property is that the μ_i all have the same sign inside q and mixed signs outside, even in the case of self-intersecting quads (see Figure 5).

Therefore, the normalization in (7) is always well-defined for a point v inside the quad and its barycentric coordinates $\lambda_i(v)$ are all positive while they have mixed signs if v lies outside. Moreover, the λ_i have the Lagrange property and are linear along the edges, except at the intersection vertex of a self-intersecting quad.

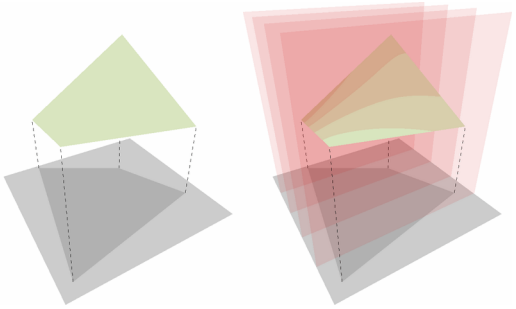


Figure 6: *A flat shaded quad (left) and its intersection with a sequence of parallel semi-transparent planes (right).*

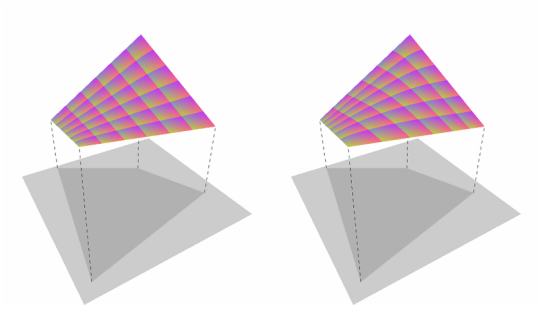


Figure 7: *Texture mapping with standard (left) and perspectively correct interpolation (right).*

3.1. Perspective correction

The simplest rendering method for a triangle is *flat shading* and using the same idea for a quad gives the expected result: a four-sided region that is filled with a constant colour. But the situation gets more exciting if we interpolate the depth values z_i of the four vertices with Equation (4) and assign the resulting depth $z(v)$ to each of the generated fragments. By intersecting the quad with parallel planes we see that this turns the quad into a smooth surface whose intersections with planes are generally curved (see Figure 6).

A nice observation is that we can explicitly describe this surface. If we take any point v inside the quad, then what we really see on screen at this position is the 3D point

$$S(v) = \lambda_0'(v)V_0 + \lambda_1'(v)V_1 + \lambda_2'(v)V_2 + \lambda_3'(v)V_3, \quad (9)$$

where the *perspective coordinates* $\lambda_i'(v)$ are defined as

$$\lambda_i'(v) = \frac{\lambda_i(v)}{w_i} \Bigg/ \sum_{j=0}^{3} \frac{\lambda_j(v)}{w_j} = \frac{\mu_i(v)}{w_i} \Bigg/ \sum_{j=0}^{3} \frac{\mu_j(v)}{w_j}. \quad (10)$$

with w_i from (1). In fact, if we write $S(v)$ in its homogeneous form and project it to screen space by first multiplying the transformation matrix $\mathbf{M}$ as in (1) and then dividing by the last component, then it is easy to show that this point maps exactly back to v with depth $z(v)$.

Apart from its theoretical value, this observation also has a practical one as it can be used for a *perspectively correct* interpolation of attributes inside the screen quad. Consider, for example, the edge midpoint $v = (v_0 + v_1)/2$ in screen coordinates. For this point we have $\lambda_0(v) = \lambda_1(v) = 1/2$ and $\lambda_2(v) = \lambda_3(v) = 0$ and it follows from (9) that the corresponding 3D surface point is $S(v) = (1-\sigma)V_0 + \sigma V_1$ with $\sigma = w_0/(w_0 + w_1)$ usually not equal to $1/2$. Since attribute interpolation should be linear along the edges of the 3D quad Q, the correct value at v is $(1-\sigma)a_0 + \sigma a_1$, in contrast to the attribute $(a_0 + a_1)/2$ that we get with the interpolation in (4). However, we can still use this equation for perspectively correct interpolation if we interpolate the attributes a_i/w_i and divide the result by the interpolation of the values $1/w_i$,

$$a(v) = \sum_{i=0}^{3} \lambda_i(v)\frac{a_i}{w_i} \Bigg/ \sum_{i=0}^{3} \lambda_i(v)\frac{1}{w_i}. \quad (11)$$

Note that this is just a restatement of Equation (9) with the V_i replaced by a_i and that this "trick" is well-known for triangles. An example that illustrates the effect of perspectively correct interpolation for a quad is shown in Figure 7.

3.2. Points behind the eye

So far, we have tacitly assumed that all vertices lie *in front* of the point of view (or *eye*), in other words, that all w_i from Equation (1) are positive. We will now show how to adapt our technique to work even in the case in which some vertices lie *behind* the eye.

For a better understanding, consider what happens if we move one vertex of a quad towards the eye and beyond while the other vertices remain fixed (see Figure 8). As V_1 approaches the eye-plane, the projected vertex v_1 leaves the screen at the bottom because w_1 converges to zero and therefore y_1 diverges to $-\infty$. The moment we push V_1 beyond this plane, w_1 becomes negative, y_1 jumps from $-\infty$ to $+\infty$, and if we keep moving, v_1 will eventually re-appear at the top of the screen.

In analogy to triangles, we have to render the *external* screen quad (see Figure 9) in this situation and in general whenever the vertices V_i lie on different sides of the eye-plane so that the w_i have different signs. This implies that we must consider the external angles and distances for the computation of the homogeneous coordinates in Equation (8).

For example, in the situation at the bottom of Figure 8, α_0' and r_1' should be used instead of α_0 and r_1 (cf. Figure 3). While it is clear that $\alpha_0' = \pi + \alpha_0$, it is less obvious how to compute the external distance, but it can be shown that $r_1' = -r_1$ gives the correct result, i.e. the barycentric coordinates that we compute with these values still have all the desired properties (Lagrange property at the vertices, linearity along the edges, and positivity inside the quad).

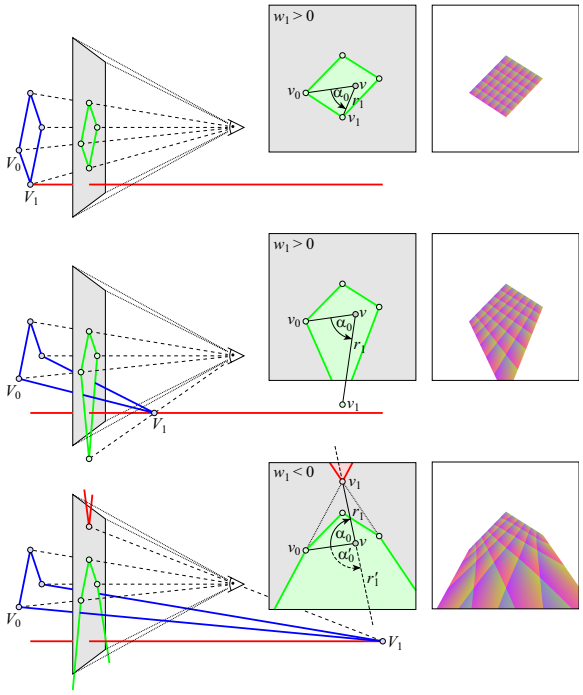


Figure 8: *If a vertex of the 3D quad moves behind the eye (left), then the screen quad becomes an external quad (middle), but the barycentric interpolation still works (right) if we use the appropriate external angles and distances.*

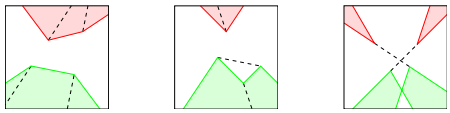


Figure 9: *Examples of the exterior shapes that the configurations in Figure 2 can assume under perspective projection. Red parts lie behind the eye and will be clipped.*

4. Implementation

Traditional hardware implementations for the rasterization of triangles are based on a simplified version of the general algorithm for scan-converting arbitrary polygons. This algorithm intersects successive scan-lines with the polygon and fills the resulting *spans* of adjacent interior pixels from left to right [FvDFH90]. This procedure simplifies a lot in the case of triangles because the number of spans is always one and it further allows to compute the attribute interpolation (4) or (11) with incremental updates on-the-fly. The general algorithm can also be simplified considerably for quadrilaterals as the maximal number of spans is two and the book-keeping of what is called *active edges* is kept at a minimum. We further explain in Section 4.2 how incremental updates can be used to optimize the computation of the attribute interpolation with mean value coordinates.

4.1. Membership test

The basic scan-line algorithm visits pixels sequentially and a common variant to speed it up is to process several pixels in parallel per clock cycle at the cost that not all of them necessarily lie inside the primitive [Pin88]. The *PixelPlanes* architecture [FIP*89] takes this idea to the extreme and completely abandons the scan-line idea by processing all pixels of a 128×128 patch simultaneously using a SIMD array. Another alternative to the scan-line algorithm is to visit pixels along a space-filling Hilbert curve in a hierarchical fashion [MWM01].

All these approaches have in common that they require an efficient test to determine whether a pixel lies inside or outside the primitive. As noticed in Section 3, this test can be implemented by checking the signs of the barycentric coordinates $\lambda_i(v)$, but especially for quads a more efficient solution is desirable.

For triangles the membership is usually tested with the help of *edge functions* [FIP*89, Pin88]. An edge function $E_{ij}(v)$ is defined to be positive (true) if and only if v lies on the left side of the line through v_i and v_j and the simplest function that fulfils this requirement is the signed area of the triangle $[v, v_i, v_j]$. It follows that a pixel at position v is inside the triangle $[v_0, v_1, v_2]$ if and only if the three edge functions E_{01}, E_{12}, and E_{20} have a common sign and the sign itself further indicates whether the pixel is front- or back-facing.

For quads there is a similar strategy that requires to evaluate the four edge functions E_{01}, E_{12}, E_{23}, and E_{30} that correspond to the quad's edges plus a fifth edge function E_{20} that tests against one of the diagonals. A detailed analysis of all the different cases that can occur (see Figure 2) reveals that a pixel is inside the quad and front-facing if and only if either of the boolean expressions

$$\begin{aligned} &E_{20} \wedge E_{01} \wedge E_{12} \wedge (E_{23} \vee E_{30}), \\ &(\neg E_{20}) \wedge E_{23} \wedge E_{30} \wedge (E_{01} \vee E_{12}) \end{aligned} \tag{12}$$

is true and that it is inside and back-facing if the same holds after negating all edge functions. An elegant way to handle exterior quads (see Figure 9) is to evaluate the homogeneous variants of the edge functions as suggested by Olano and Greer [OG97] in the case of triangles.

4.2. Barycentric coordinates

For every pixel that has passed the membership test we now want to interpolate attributes either in the standard way (4) for the depth value or with perspective correction (11) for any other attribute. Both interpolations require to compute and normalize the mean value coordinates (8) which can be simplified using the identity

$$\tan\left(\frac{\alpha_i}{2}\right) = \frac{r_i r_{i+1} - D_i}{A_i}$$

where r_i is the length of the vector $s_i = v_i - v$, the value D_i denotes the dot-product between s_i and s_{i+1}, and A_i is twice

the signed area of the triangle $[v, v_i, v_{i+1}]$. A similar identity holds for exterior angles $\alpha_i' = \pi + \alpha_i$,

$$\tan\left(\frac{\alpha_i'}{2}\right) = -\cot\left(\frac{\alpha_i}{2}\right) = \frac{-r_i r_{i+1} - D_i}{A_i},$$

and substituting these formulas as well as any occurrence of exterior distances $r_i' = -r_i$ into Equation (8) leads to the following pseudo-code that handles all cases correctly:

barycentric coordinates (point2D v)

$s_i = v_i - v$ $\quad i = 0, \ldots, 3$
$A_i = s_i \times s_{i+1}$ $\quad i = 0, \ldots, 3$
$D_i = s_i \cdot s_{i+1}$ $\quad i = 0, \ldots, 3$
$r_i = \|s_i\| \cdot \operatorname{sign}(w_i)$ $\quad i = 0, \ldots, 3$
$t_i = (r_i r_{i+1} - D_i)/A_i$ $\quad i = 0, \ldots, 3$
$\mu_i = (t_{i-1} + t_i)/r_i$ $\quad i = 0, \ldots, 3$
$\Sigma = \mu_0 + \mu_1 + \mu_2 + \mu_3$
$\lambda_i = \mu_i/\Sigma$ $\quad i = 0, \ldots, 3$
return $(\lambda_0, \lambda_1, \lambda_2, \lambda_3)$

This code can be further simplified in a hardware implementation because A_i is linear and the values D_i and r_i^2 vary quadratically in v. Thus, all of them can be computed efficiently with incremental updates from pixel to pixel and the vectors s_i are no longer needed then. Analogously to triangles, the initial values are generated in a *quad setup* phase that precedes the actual rasterization.

4.3. Back-face culling and clipping

For a triangle, back-face culling can be decided for the entire primitive and the same holds for quads except if the screen quad self-intersects (see Figure 2). In that situation, the shape decomposes into two triangles, one of which is front-facing while the other is back-facing and should be culled. Note that span coherence allows discarding entire spans at a time in a scan-line implementation. However, the best implementation of back-face culling depends heavily on the chosen rasterization technique.

Similar considerations hold for clipping against the z_{near}- and z_{far}-planes. An ad hoc approach is to discard fragments with a depth value outside the interval $[z_{\text{near}}, z_{\text{far}}]$, but depending on the specific architecture, there may be ways to clip them earlier in the pipeline. Note that clipping against the sides of the viewing frustum usually comes for free by producing only fragments that lie inside the viewport.

5. Results

Although our method is intended to be implemented as a hardware rasterizer, in order to test it and have a visual insight to the results obtained with it, we resorted to an hybrid software/hardware implementation that runs on current triangle rasterizing GPUs. All rendering examples shown in this paper have been produced with this implementation.

5.1. Hybrid implementation

In our test implementation, all vertex processing (including the projection) has been lifted from the vertex shader to the CPU, while the computation of the barycentric coordinates moved down from the hypothetical rasterizer to the fragment shader. The result is inefficient compared to a real hardware rasterizer, but it is good enough to render simple quad meshes in real-time.

To be specific, we first project all vertices according to Equation (1), record the values v_i, z_i, and w_i, and store them into the current rendering state so that we can access them in the fragment shader later. The original vertex attributes a_i (e.g. per-vertex normals) are similarly stored in this state.

We then render the quad by simply splitting it into two triangles. Each fragment that is produced by the triangle rasterizer is processed by a fragment program that computes the barycentric coordinates λ_i from the fragment's screen position v and from the points v_i as shown in the pseudo-code above and carries out the necessary attribute interpolations as mentioned in Section 4.2. Interpolated attributes are then processed as usual to compute the final colour value.

By rendering the two triangles we are sure to generate all fragments inside the quad plus some additional ones in some cases (for concave and self-intersecting shapes) that lie outside. The latter ones are detected and killed in the fragment program by checking the signs of the barycentric coordinates as mentioned in Section 3.

5.2. Properties and discussion

The improvement of our method over the currently used diagonal split is remarkable and the examples in Figures 1, 10, and 11 emphasize that the quality matches more or less the one obtained by an expensive refinement of the quad with bilinear interpolation in 3D. The quality mainly stems from the smoothness of the mean value coordinates that guarantees a C^∞-continuous interpolation inside the quad.

Another important characteristic of our rendering technique is that it is highly compatible with triangle rendering. The linearity of the attribute interpolation along edges ensures C^0-continuity across adjacent quads or between neighbouring quads and triangles; crease angles can still be obtained as usual by assigning different normals at the vertices shared by two adjacent quads (or triangles); and depth values are computed consistently with other primitives. Furthermore, the only part of the graphics pipeline that is affected is the rasterizer and any per-vertex and per-fragment processing remains unchanged.

Another advantage is that our quads are compliant with the *OpenGL* specification [SA03] and only the API *implementation* of the primitives created in GL_QUAD or GL_QUAD_STRIP mode would have to be replaced. Direct3D syntax, however, would require an extension to include the "quad" as a new "primitive type".

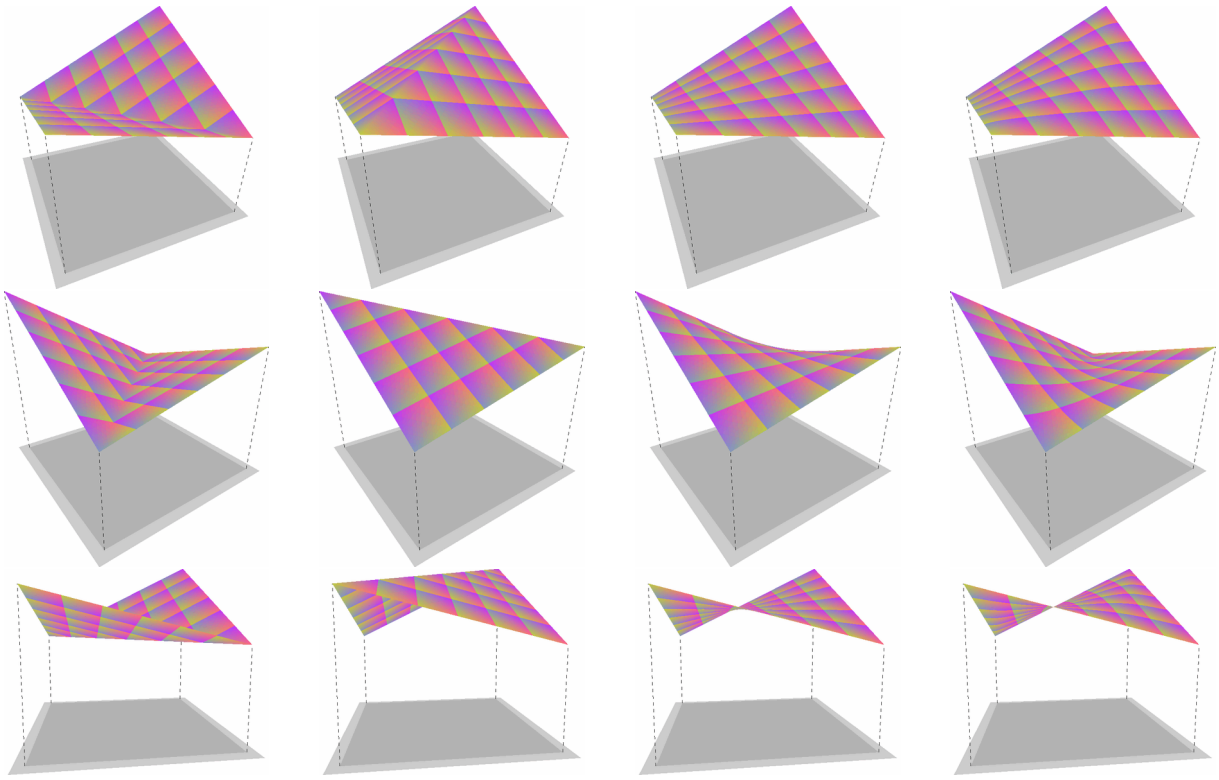


Figure 10: *Visual comparison of different methods to render a single textured quad. We added a shadow and dotted vertical lines to suggest the actual 3D geometry of the quad. From left to right: the two diagonal spits, a* 20×20 *refinement of the quad using bilinear interpolation, and our proposed method that renders the quad as an atomic primitive. Each column shows the same quad seen from different points of view, resulting in the three possible projected shapes (cf. Figure 2).*

The cost of the proposed technique is an increased complexity of the rasterizer. The part of the ARB fragment program that computes the barycentric coordinates in our hybrid test implementation (see Section 5.1) is only 19 instructions long and this suggests that a hard-wiring in a rasterizer would not be too demanding.

A potential drawback of our method is the view dependency of the result as the underlying 3D shape of the rendered surface (see Section 3.1) varies according to the current viewing direction (except for planar quads). This is unavoidable because we always fill the interior of a 2D screen quad and a non-planar 3D surface that projects to a quadrilateral shape from all viewing directions does not exist. On the other hand, this guarantees that every pixel corresponds to exactly one surface point and therefore enables a straightforward rendering of two-sided quads. The effect of view dependency can be undesirable at times and becomes especially visible for textured non-planar quads that cover big portions of the screen. But it is almost invisible for quad meshes with higher resolution for which the diagonal split still creates ugly discontinuities (see Figure 11).

Acknowledgements

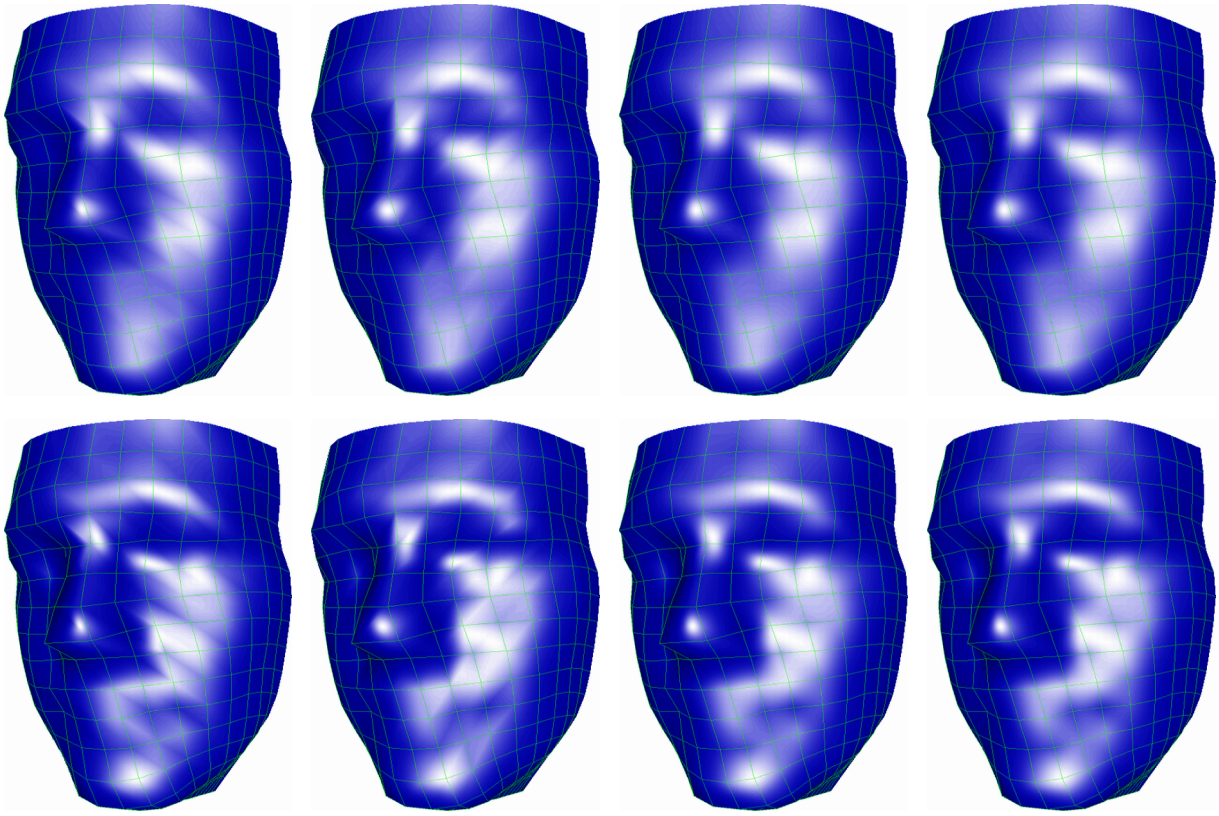
This work was supported in part by the Deutsche Forschungsgesellschaft (DFG HO-2457/1-1) and the projects *ViHAP3D* (EU IST-2001-32641) and *MACROGeo* (FIRB-MIUR RBAU01MZJ5). Thanks to Michael Floater, Paolo Cignoni, and our reviewers for their helpful suggestions.

References

[Duf79] DUFF T.: Smoothly shaded renderings of polyhedral objects on raster displays. *ACM SIGGRAPH Computer Graphics 13*, 3 (1979), 270–275. Proceedings of SIGGRAPH '79. 1

[FHK04] FLOATER M. S., HORMANN K., KÓS G.: A general construction of barycentric coordinates over convex polygons. *Advances in Computational Mathematics* (2004). 2, 3

[FIP*89] FUCHS H., ISRAEL L., POULTON J., EYLES J., GREER T., GOLDFEATHER J., ELLSWORTH D., MOLNAR S., TURK G.,



diagonal split 1 *diagonal split 2* *bilinear interpolation* *our method*

Figure 11: *Visual comparison of different methods to render a simple Phong-shaded quad mesh. The over-imposed wireframe shows the connectivity. From left to right: the two diagonal spits, a* 20×20 *refinement of the quads using bilinear interpolation, and our proposed method that renders the quads as atomic primitives. The per-vertex normals that were in the examples on top were computed from the 3D geometry. In the bottom row, we used the normals that were originally defined for the model which yield a more accurate description of the actual shape, but also make the artefacts of the diagonal split more evident.*

TEBBS B.: Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. *ACM SIGGRAPH Computer Graphics 23*, 3 (1989), 79–88. Proceedings of SIGGRAPH '89. 5

[Flo03] FLOATER M. S.: Mean value coordinates. *Computer Aided Geometric Design 20*, 1 (2003), 19–27. 2, 3

[FvDFH90] FOLEY J. D., VAN DAM A., FEINER S., HUGHES J.: *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990. 2, 5

[MLBD02] MEYER M., LEE H., BARR A. H., DESBRUN M.: Generalized barycentric coordinates on irregular polygons. *Journal of Graphics Tools 7*, 1 (2002), 13–22. 3

[MWM01] MCCOOL M. D., WALES C., MOULE K.: Incremental and hierarchical Hilbert order edge equation polygon rasterization. In *Proceedings of the 2001 Workshop on Graphics Hardware* (2001), pp. 65–72. 5

[OG97] OLANO M., GREER T.: Triangle scan conversion using 2D homogeneous coordinates. In *Proceedings of the 1997 Workshop on Graphics Hardware* (1997), pp. 89–95. 5

[Pin88] PINEDA J.: A parallel algorithm for polygon rasterization. *ACM SIGGRAPH Computer Graphics 22*, 4 (1988), 17–20. Proceedings of SIGGRAPH '88. 5

[SA03] SEGAL M., AKELEY K.: The OpenGL graphics system: A specification (version 1.5), Oct. 2003. 1, 6

[Wac75] WACHSPRESS E. L.: *A Rational Finite Element Basis*. Academic Press, 1975. 3