

Correcting Texture Mapping Errors Introduced by Graphics Hardware

MANUEL M. OLIVEIRA

Department of Computer Science
State University of New York at Stony Brook
oliveira@cs.sunysb.edu

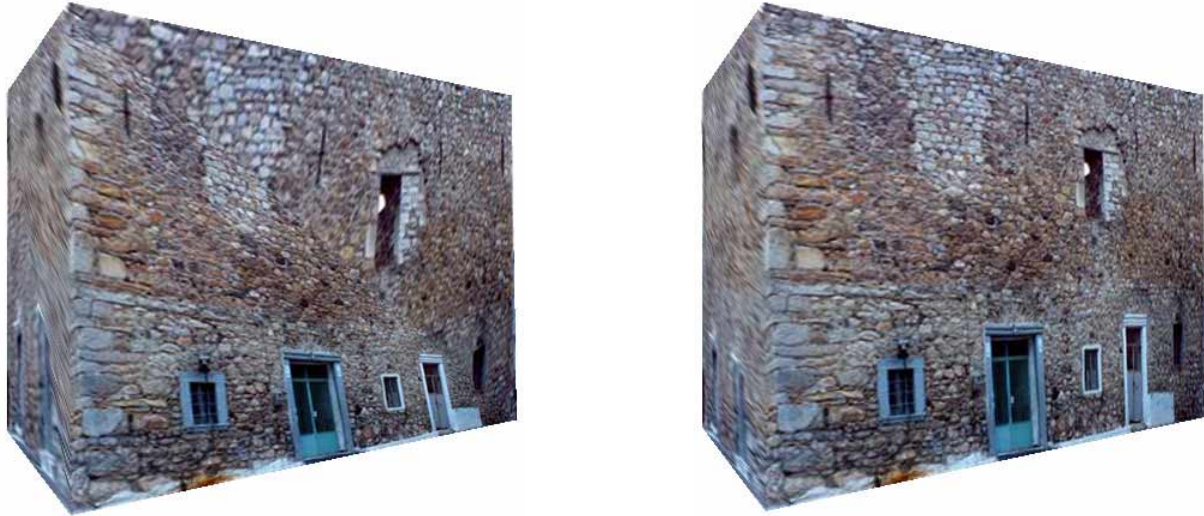


Figure 1. View of an old building produced by texture mapping two rectangles using texture patches from the photograph shown in Figure 2. The image on the left was rendered conventionally using OpenGL. Note the distorted façade. On the right, the rendering produced using our algorithm.

Abstract

Most rendering engines subdivide arbitrary polygons into a collection of triangles, which are then rendered independently of each other. This procedure can introduce severe artifacts when rendering texture-mapped quadrilaterals. These errors result from a fundamental limitation: the impossibility of performing general projective mappings using only the information available at the vertices of a single triangle. Texture mapping involving arbitrary quadrilaterals has practical importance in applications that use real images as textures, such as in image-based modeling and rendering. This paper presents an efficient adaptive-subdivision solution to the problem, which, due to its simplicity, can be easily incorporated into graphics APIs.

1 Introduction

Polygonal meshes are extensively used in computer graphics to approximate arbitrary surfaces using simple planar primitives. While triangles are by far the most used of these primitives, quadrilaterals are also frequently used, especially for the modeling of large textured planar

surfaces. Graphics libraries, such as OpenGL, have recognized the importance of quadrilaterals for geometric modeling and provide special constructs for specifying quadrilaterals directly [21].

To simplify the design of graphics hardware, arbitrary polygons are usually subdivided into triangles, which are then rendered independently of each other. This practice, however, can introduce objectionable artifacts when rendering texture-mapped quadrilaterals. Although texture mapping of quadrilaterals is a well-understood problem [10], these errors persist due to the impossibility of performing arbitrary projective mappings using only the information available at the vertices of a single triangle. This is an artifact of the triangle-oriented graphics pipeline and cannot be fixed with the use of perspective-correct interpolation techniques [2, 11] during rasterization. Thus, all state-of-the-art polygon engines suffer from these problems.

Mappings involving arbitrary quadrilaterals have practical importance for applications that use real images as textures, such as in image-based modeling and rendering [6, 7, 8, 13]. For instance, consider the problem of constructing geometric models for buildings from a set



Figure 2. Photograph of an old building.

of uncalibrated photographs (*i.e.*, without explicit information about the camera pose or the imaged geometry) [8, 13]. Figure 2 shows a photograph of an old building for which a texture-mapped box can be used as a reasonable approximate model. Notice, however, that due to the foreshortening introduced by perspective projection, the image of the façade is not rectangular. When textured onto a rectangle using, for instance, OpenGL, the resulting image presents severe distortions (Figure 3 (bottom left)). The image to its right shows the rendering produced by our algorithm. In this case, although occlusions are still relative to the original viewpoint, the errors introduced by the rasterization of triangles have been removed. Notice, for example, the horizontal and vertical lines of the portals.

While several researchers [12, 14, 3, 19, 18, 17] have devised methods for minimizing distortions resulting from the mapping of textures onto arbitrary surfaces, we focus on a much more restricted problem: eliminating texture distortions introduced by current graphics engines (as opposed to distortions introduced during the mapping itself). Note, however, that the rendering of distortion-free mappings may still contain artifacts introduced during rasterization, as illustrated in Figure 3 (left).

This paper presents an efficient adaptive-subdivision solution to remove these artifacts. The subdivision process is driven by the shapes of the quadrilateral and the texture patch, as well as by the projected area of the quadrilateral in screen space. The algorithm is based on the observation that mappings involving parallelograms are affine and, therefore, can be correctly performed by a triangle-oriented rasterization unit. We show that by recursively subdividing a convex quadrilateral, the resulting polygons converge to parallelograms. Thus, the solution consists of adaptively subdividing both the polygons and the texture patches before rendering.

Section 2 provides a review of some relevant observations for understanding the problem and its solution. Section 3 describes the details of the subdivision

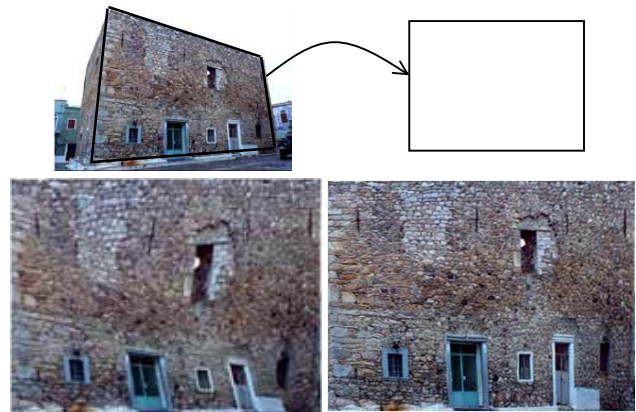


Figure 3. Mapping the building façade onto a rectangle (top). Rendering produced by OpenGL (bottom left). Result produced by our algorithm (bottom right).

algorithm, while Sections 4 and 5 presents results and conclusions.

2 Understanding the Problem

Consider a texture map defined over the parameter space $[0,1] \times [0,1]$ and a trapezoid Q with vertices A , B , C and D , as shown in Figure 4. Also, let texture coordinates $(0,0)$, $(1,0)$, $(1,1)$ and $(0,1)$ be assigned to vertices A , B , C and D , respectively. While the expected image resulting from such a mapping is illustrated in Figure 5, the results produced by current graphics engines are shown in Figure 6. These are not only incorrect, but are also dependent on the order in which the vertices of the quadrilateral are specified. For example, in Figure 5 (left) the vertices were specified as A , B , C and D . In Figure 5 (right), the order was B , C , D and A . As the trapezoid is subdivided along one of its diagonals, half of the texture is compressed in the smaller triangle, while the other half is stretched in the bigger one (Figure 6). The underlying triangles are easily identifiable.

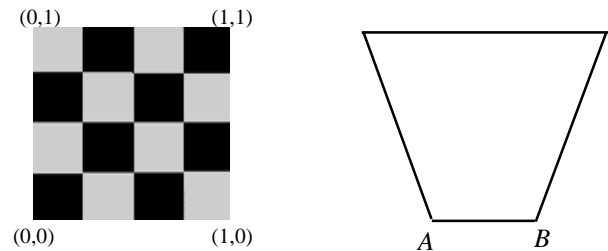


Figure 4. Texture map (left) and trapezoid (right).



Figure 5. Desired texture-mapped polygon.

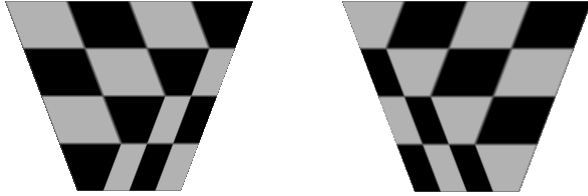


Figure 6. Texture-mapped trapezoid rendered by state-of-the-art graphics hardware. The results are incorrect and dependent of the order in which the vertices are specified.

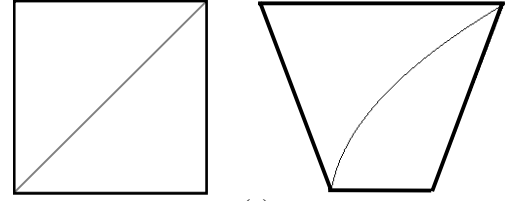
2.1 From Another Perspective

Another way to understand this problem is to consider it from a texture-mapping perspective. Texture mapping can be conceptually understood as a two-step process: a transformation from texture to object space followed by a transformation from object to screen space [10]. The 3D coordinates of a point P inside Q can be expressed as a bilinear interpolation of the coordinates of the vertices of Q , as shown in Equation (1), where (s,t) are the normalized coordinates (*i.e.*, $s, t \in [0,1]$) associated with P in Q .

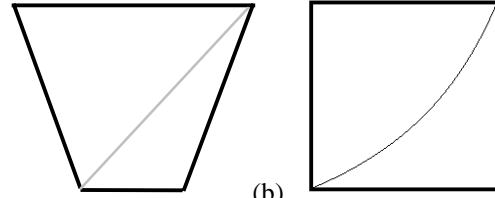
$$P(s,t) = A(1-s)(1-t) + B(s)(1-t) + C(s)(t) + D(1-s)(t) \quad (1)$$

The mapping from texture to object space can be simply stated as $P(s,t).color = Texture(s,t)$. But bilinear mappings only preserve straight lines that are horizontal and vertical in the source quadrilateral. Lines in all other orientations are mapped to quadratic curves [20]. Figure 7 illustrates this situation for the case of the diagonals of the two quadrilaterals shown in Figure 3. Points along the diagonals of the polygons shown on the left are mapped to points with the same normalized coordinates in the polygons on the right. Since, in general, diagonal lines are not mapped to each other (but to quadratic curves), interpolation of texture coordinates associated with the vertices of the subdivided triangles (as done by rasterization units) produces incorrect results. Preserving diagonal lines is equivalent to obtaining an affine mapping, which exists if both quadrilaterals are parallelograms (see Appendix A).

The artifacts in Figures 6 and 3 (left) result from the fact that the mapping between two planar quadrilaterals



(a)



(b)

Figure 7. Diagonal lines are mapped to quadratic curves under bilinear transformations.

usually involves a projective transformation, thus requiring the specification of correspondences among four points in each quadrilateral [10]. Correspondences including only the three vertices of a triangle are not enough in general. However, the mapping between two parallelograms is affine and, in this case, as it suffices to establish correspondences between three pairs of non-collinear points, rasterization units produce correct renderings (Figure 8).

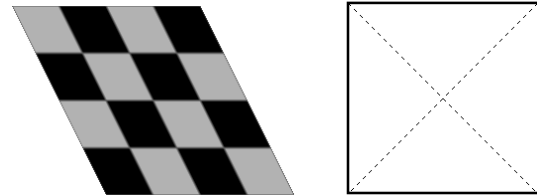


Figure 8. Rendering of a parallelogram as two triangles (left). The diagonals of the two parallelograms map to each other (right).

3 The Adaptive Subdivision Algorithm

Most quadrilaterals and texture patches are not parallelograms. One possible approach to reduce the error is to use subdivision. Catmull's algorithm [4], for instance, consists of subdividing patches until they reach a pixel size. We show that polygons obtained by repeated uniform subdivision of convex quadrilaterals converge to parallelograms (Appendix B). Our algorithm then recursively subdivides the original quadrilateral and the texture patch until each of the sub-quadrilaterals (in both object and texture space) closely approximates a parallelogram. The subdivision process is driven simultaneously by the shapes of the polygon and the texture patch, as well as by the projected area of the

quadrilateral in screen space, as it will be explained shortly. Figure 12 illustrates the case of a mapping involving quadrilaterals of arbitrary shapes.

An important aspect of any adaptive subdivision procedure is a criterion to decide when subdivision is necessary. Usually, this takes the form of an error metric and some threshold. We take advantage of the unique geometry of a parallelogram to define a simple subdivision criterion. Opposite angles (sides) of a parallelogram have equal values (lengths) (Figure 9 (left)). Thus, let $V_1 = A - B$, $V_2 = C - B$, $V_3 = C - D$ and $V_4 = A - D$ be four vectors associated to the edges of a quadrilateral (Figure 9 (right)). For a parallelogram, $V_1 \cdot V_2 = V_3 \cdot V_4$ and $V_1 \cdot V_4 = V_2 \cdot V_3$. Then, for each texture-mapped quadrilateral, we compute the differences $d_1 = \text{abs}(V_1 \cdot V_2 - V_3 \cdot V_4)$ and $d_2 = \text{abs}(V_1 \cdot V_4 - V_2 \cdot V_3)$ and check if they are below a certain threshold. Note that the use of non-normalized vectors in the dot products favors the subdivision of larger polygons, which are most likely to exhibit noticeable artifacts. Given a texture patch T_p to be mapped onto a quadrilateral, the algorithm computes a similar set of four vectors for T_p and performs analogous difference tests, but in this case using a smaller threshold tailored for the normalized texture space. If either the polygon or the texture patch fails its corresponding test, both are subdivided according to their normalized coordinates. The subdivision itself is quite straightforward. It consists of introducing a total of five new vertices: one in the middle of each original edge, plus a vertex shared by all sub-quadrilaterals. Their coordinates are given by $AB=(A+B)*0.5$, $BC=(B+C)*0.5$, $CD=(C+D)*0.5$, $AD=(A+D)*0.5$, $ABCD = (AB+CD)*0.5$, respectively (Figure 11).

If the projection of the polygon covers only a few pixels on the screen, the use of thresholds defined in object and texture spaces might lend to unnecessary refinement. A screen-space criterion should be included to avoid unneeded subdivisions. For this, we use the projected area of the original quadrilateral computed at each frame. Since any step of the subdivision procedure divides a quadrilateral into four smaller polygons, the algorithm assumes that the quadrilateral is split into four

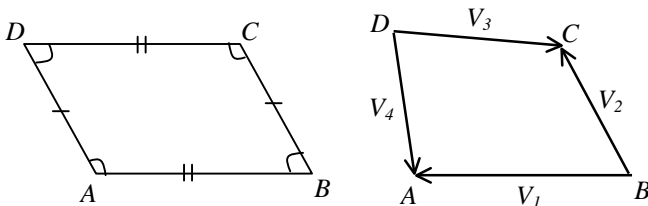


Figure 9. Opposite angles (sides) of a parallelogram are equal (left). Four vectors defined by the vertices of the quadrilateral (right).

regions of equal areas. Although this assumption is often incorrect, it provides a good approximation and is used to avoid computing the projected area for sub-quadrilaterals.

The pseudocode for the adaptive subdivision algorithm is shown in Figures 10 and 11. In Figure 10, we calculate the projected area (in pixels) of the original quadrilateral as the area of a 2D polygon [16]. The pixel coordinates are obtained using the ModelView and Projection matrices, as well as the dimensions of the window readily available from OpenGL [21]. The computed area is then passed as a parameter to the adaptive subdivision algorithm shown in Figure 11.

The adaptive subdivision procedure (Figure 11) computes the difference values d_1 and d_2 , for both the quadrilateral and the texture patch. The differences are then compared to threshold values defined for object space ($threshold_q$) and texture space ($threshold_t$), respectively. If all difference values are below their corresponding thresholds or if the area of the quadrilateral is less than two hundred pixels, the quadrilateral is

```

void ComputeAreaAndDraw(Quad Q, Text_param ST,
                        int window_width, int window_height)
{
    GLfloat modelViewMatrix[16], projMatrix[16], vertices[16];
    Vec2f vertex_pixel[4];
    float polygon_area = 0.0;
    //
    glGetFloatv(GL_MODELVIEW_MATRIX, modelViewMatrix);
    glGetFloatv(GL_PROJECTION_MATRIX, projMatrix);
    Initialize_vertices_matrix(Q, vertices);
    // multiply Projection*ModelView*vertices
    glPushMatrix();
    glLoadIdentity();
    glMultMatrixf(projMatrix);
    glMultMatrixf(modelViewMatrix);
    glMultMatrixf(vertices);
    glGetFloatv(GL_MODELVIEW_MATRIX, vertices);
    glPopMatrix();
    // perspective division
    for (int i=0; i<13; i+=4)
        if (vertices[i+3] != 0) {
            vertices[i] /= vertices[i+3]; vertices[i+1] /= vertices[i+3];
        }
    // map vertices to pixel coordinates
    for (i=0; i<4; i++) {
        vertex_pixel[i].x = 0.5*(vertices[4*i] + 1.0f)*window_width;
        vertex_pixel[i].y = 0.5*(vertices[4*i+1] + 1.0f)*window_height;
    }
    // compute polygon area in pixels
    for (i=0; i<4; i++)
        polygon_area += (vertex_pixel[i].x*vertex_pixel[(i+1)%4].y -
                        vertex_pixel[i].y*vertex_pixel[(i+1)%4].x);
    polygon_area = 0.5*fabs(polygon_area);
    // call subdivide and draw
    SubdivideAndDrawQuad(Q, ST, polygon_area);
}

```

Figure 10. Computation of the projected area of the quadrilateral in screen space.

rendered. Otherwise, it is subdivided and the procedure is recursively called for the four sub-quadrilaterals. The resulting mesh is an unrestricted quad-tree [5].

```

void SubdivideAndDrawQuad(Quad Q, Text_param ST,
                          float q_area)
{
    Quad Q_prime;
    Text_param ST_prime;
    // compute the dot products for the polygon
    float v1_dot_v2_quad = (Q.A - Q.B) dot (Q.C - Q.B);
    float v3_dot_v4_quad = (Q.C - Q.D) dot (Q.A - Q.D);
    float v1_dot_v4_quad = (Q.A - Q.B) dot (Q.A - Q.D);
    float v2_dot_v3_quad = (Q.C - Q.B) dot (Q.C - Q.D);
    // compute the dot products for the texture
    float v1_dot_v2_text = (ST.A - ST.B) dot (ST.C - ST.B);
    float v3_dot_v4_text = (ST.C - ST.D) dot (ST.A - ST.D);
    float v1_dot_v4_text = (ST.A - ST.B) dot (ST.A - ST.D);
    float v2_dot_v3_text = (ST.C - ST.B) dot (ST.C - ST.D);
    // check if the quad covers less than 200 pixels or if
    // differences are less than the threshold
    if (q_area < 200 || // area of the quad in pixels
        (fabs(v1_dot_v2_quad - v3_dot_v4_quad) < threshold_q &&
         fabs(v1_dot_v4_quad - v2_dot_v3_quad) < threshold_q &&
         fabs(v1_dot_v2_text - v3_dot_v4_text) < threshold_t &&
         fabs(v1_dot_v4_text - v2_dot_v3_text) < threshold_t)) {
    // no need to subdivide: draw the quadrilateral
    glBegin(GL_QUADS);
        glVertex2fv(&(ST.A.s)); glVertex3fv(&(Q.A.x));
        glVertex2fv(&(ST.B.s)); glVertex3fv(&(Q.B.x));
        glVertex2fv(&(ST.C.s)); glVertex3fv(&(Q.C.x));
        glVertex2fv(&(ST.D.s)); glVertex3fv(&(Q.D.x));
    glEnd();
    }
    else { // subdivide the quadrilateral
        vec3f AB = (Q.A + Q.B)*0.5f;
        vec3f BC = (Q.B + Q.C)*0.5f;
        vec3f CD = (Q.C + Q.D)*0.5f;
        vec3f AD = (Q.A + Q.D)*0.5f;
        vec3f ABCD = (AB + CD)*0.5f;
    // subdivide texture parameter space
        vec2f AB_st = (ST.A + ST.B)*0.5f;
        vec2f BC_st = (ST.B + ST.C)*0.5f;
        vec2f CD_st = (ST.C + ST.D)*0.5f;
        vec2f AD_st = (ST.A + ST.D)*0.5f;
        vec2f ABCD_st = (AB_st + CD_st)*0.5f;
    // Initialize parameters and recursive call
        Q_prime(A, AB, ABCD, AD);
        ST_prime(AB_st, AB_st, ABCD_st, AD_st);
        SubdivideAndDrawQuad(Q_prime, ST_prime, 0.25*q_area);
        Q_prime(AB, B, BC, ABCD);
        ST_prime(AB_st, B_st, BC_st, ABCD_st);
        SubdivideAndDrawQuad(Q_prime, ST_prime, 0.25*q_area);
        Q_prime(ABCD, BC, C, CD);
        ST_prime(ABCD_st, BC_st, C_st, CD_st);
        SubdivideAndDrawQuad(Q_prime, ST_prime, 0.25*q_area);
        Q_prime(AD, ABCD, CD, D);
        ST_prime(AD_st, ABCD_st, CD_st, D_st);
        SubdivideAndDrawQuad(Q_prime, ST_prime, 0.25*q_area);
    }
}

```

Figure 11. Adaptive subdivision algorithm.

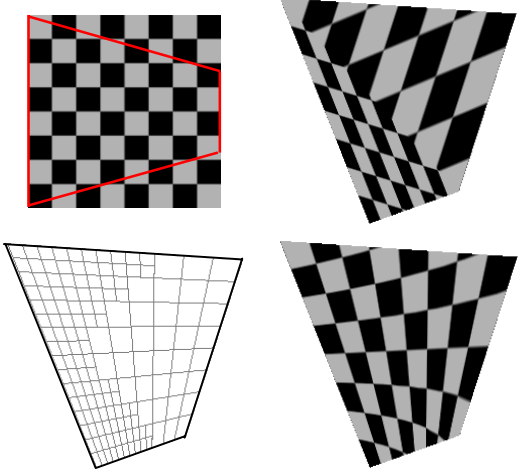


Figure 12. Mapping involving quadrilaterals of arbitrary shapes. The selected portion of the checkerboard on top left was texture-mapped onto the quadrilateral shown to its right. Top right: result produced by OpenGL. Bottom left: adaptively subdivided mesh. Bottom right: rendering produced with our algorithm.

4 Results

We have implemented the adaptive subdivision algorithm described in Figures 10 and 11 in C++. According to our experience, an object-space threshold (*threshold_q* in Figure 11) of 10^{-3} and a texture-space threshold (*threshold_t*) of 10^{-4} seems to be appropriate to detect the need for polygon and texture subdivisions, respectively. For the screen-space criterion, we used a polygon area of two hundred pixels, which was defined empirically. The use of an area threshold of four hundred pixels produces virtually the same results, except when the entire original quadrilateral covers a little less than four hundred pixels and, in this case, the artifacts introduced by conventionally rendering the quadrilateral as two triangles are still noticeable.

Figure 12 illustrates the case where both the polygon and the texture patch are not parallelograms and both drive the subdivision process. The polygon was oriented parallel to the image plane to avoid foreshortening introduced by perspective projection, thus allowing us to fully concentrate on the artifacts to be corrected for. On the top left image, a polygonal line highlights the limits of the texture patch. The image on the top right shows the conventional rendering produced by OpenGL. The bottom left image shows the adaptively subdivided mesh and, to its right, the artifact-free rendering produced by our algorithm.

Figure 13 illustrates the effect of the screen-space criterion for stopping the subdivision process. Here, the same texture patch used in Figure 12 was mapped to the

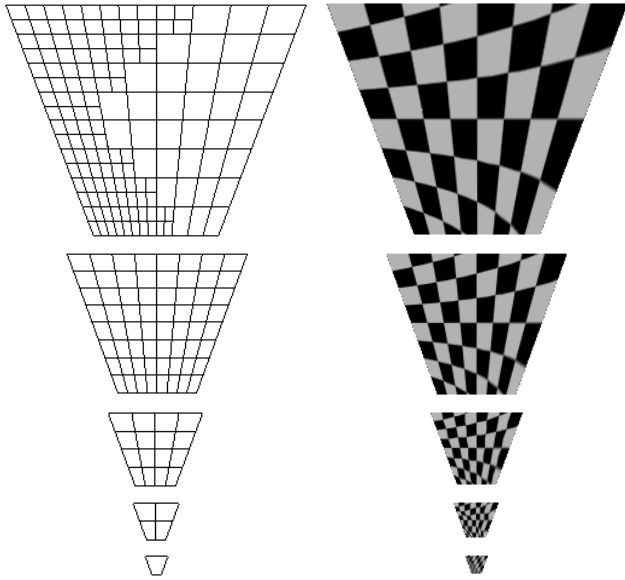


Figure 13. As the projected area of the quadrilateral in screen space becomes smaller, the resulting mesh becomes coarser.

trapezoid of Figure 4. As the projected area of the polygon gets smaller, the screen-space threshold prevents the subdivision process from proceeding through unnecessary steps (Figure 13). The mesh then becomes progressively coarser going from several polygons down to a single quadrilateral (left column) without affecting the quality of the rendering (right column). From top to bottom, the projected areas of the quadrilaterals shown in Figure 13 cover 33,600, 11,879, 3,051, 795 and 197 pixels, respectively.

The subdivision criterion is evaluated on the fly and, therefore, vertex and texture coordinates can be changed at any time. This is illustrated in Figure 14, where the top row shows a checkerboard viewed in perspective. Note that, since both the polygon and the texture patch are parallelograms (squares), the algorithm performs no subdivisions. As vertex V (the bottom right vertex in Figure 14) is dragged in 3D, the quadrilateral is automatically subdivided to avoid rendering artifacts. This is shown in the middle and bottom rows of Figure 14. In these examples, polygon planarity was enforced. In the middle row, V was pulled back and to the left. On the bottom row, V was pushed closer to the other vertices. The corresponding meshes are shown on the left and illustrate the dynamics of the subdivision process.

If the vertices of the quadrilateral are not constrained to be coplanar, the adaptive subdivision algorithm can still be used (Figure 15). In this case, however, one should not perform the area size test shown in Figure 11 ($q_area < 200$). Projections of non-planar quadrilaterals

may fold over depending on the viewing conditions, causing the fold silhouettes to “pop”. This happens because a slight change in viewpoint may cause the estimated projected area of the associated sub-quadrilaterals to oscillate above and below the area-subdivision threshold, thus causing different numbers of polygons to be rendered at the silhouettes.

Figure 1 shows the rendering of the building in Figure 2 from a novel viewpoint. These images were produced by texture mapping two rectangles with two (quadrilateral) texture patches corresponding to the façade and the visible sidewall of the building. The image on the left was produced by conventionally texture mapping the rectangles using OpenGL. Note the severe distortions on the façade. The image on the right was rendered using our adaptive subdivision algorithm and presents no such distortions.

The presented algorithm removes artifacts at the expense of rendering extra polygons. Although the rendering time is expected to increase, most scenes contain a relatively small number of quadrilaterals. Moreover, current PC graphics accelerators are already capable of rendering about thirty million triangles and about two billion filtered textured pixels per second [1,15]. As a result, the renderings produced with the proposed algorithm show a great improvement in image quality while no perceived frame-rate degradation has been observed.

Adaptive subdivision of quadrilateral meshes introduces T-vertices, which can be eliminated by splitting the corresponding quadrilaterals into triangles [5]. Although T-vertices are a major concern when the illumination function is evaluated only at the vertices of the mesh, such as in Gouraud shading [9] and in radiosity

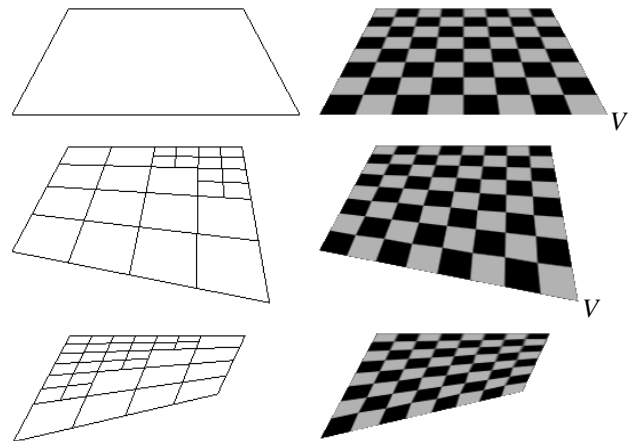


Figure 14. Top: a checkerboard seen in perspective. Middle and bottom: as vertex V is dragged in 3D, the quadrilateral is automatically re-tessellated. In all cases the corresponding meshes are shown to the left.

methods [5], the problem is less concerning in the presence of flat textured surfaces, especially if the resulting polygons are small.

The presented subdivision algorithm was designed for convex quadrilaterals. When used with concave quadrilaterals, the shared vertex may fall outside of the original polygon and the resulting rendering will exhibit a fold similar to the ones produced in the case of non-planar quadrilaterals (Figure 15). The conventional rendering of concave texture-mapped quadrilaterals (as two triangles) may also present a fold depending on the order in which the vertices are specified. Selecting the shared vertex so that all edges of the sub-quadrilaterals fall inside the original quadrilateral eliminates the folding problem.

5 Conclusions

Graphics engines usually subdivide arbitrary polygons into triangles, making it impossible, in general, to correctly render texture-mapped quadrilaterals from the resulting triangulation. This paper introduced an adaptive subdivision algorithm for correcting these artifacts. The subdivision is guided by the shapes of the quadrilateral and texture patch, and by the projected area of the quadrilateral in screen space. The algorithm is simple, efficient, and can be easily incorporated into graphics APIs to hide the limitations of the underlying hardware.

The discussed artifacts result from non-linear mappings involving the quadrilateral and the texture map (Figure 7). If texture coordinates are carefully assigned so that the mapping is linear, no distortions happen. Figure 16 illustrates this situation, which is the typical case for polygons with more than 4 sides. Although this produces images without artifacts, it limits the results that can be achieved.

The presented subdivision approach can be used with any texture-resampling method. Although the most immediate application of this algorithm is to correct artifacts in conventional texture mapping, it can also be used with other kinds of mappings (*e.g.*, bump mapping and normal mapping) involving quadrilateral patches.

Acknowledgements

The author would like to thank Hong Qin for an enlightening discussion during an earlier stage of this work and the anonymous reviewers for their insightful suggestions. Figure 1 shows an old building in Chios and is a courtesy of the Chian Federation. ATI generously donated a Radeon™ graphics card for this research.

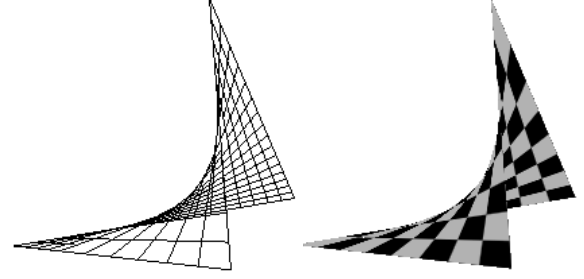


Figure 15. Non-planar quadrilateral.

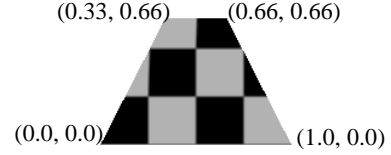


Figure 16. Trapezoid mapped with the checkerboard in Figure 4 (left) with the assigned texture coordinates.

Appendix A

Preserving diagonal lines between two quadrilaterals is equivalent to obtaining an affine mapping. Rewriting Equation (1) in terms of the x and y (or z) components of the vertices of Q , one gets:

$$P_x(s, t) = A_x(1-s)(1-t) + B_x(s)(1-t) + C_x(s)(t) + D_x(1-s)(t) \quad (2)$$

$$P_y(s, t) = A_y(1-s)(1-t) + B_y(s)(1-t) + C_y(s)(t) + D_y(1-s)(t) \quad (3)$$

Solving Equations (2) and (3) for s and t gives¹

$$as^2 + bs + c = 0, \quad s \in [0, 1] \quad (4)$$

$$t = \frac{(P_y - A_y) + sk_4}{k_5 + sk_6} \quad (5)$$

where

$$a = (k_2k_6 + k_3k_4)$$

$$b = (k_2k_5 - k_1k_4 + k_3(P_y - A_y) - k_6(P_x - A_x))$$

$$c = (A_x - P_x)k_5 + (A_y - P_y)k_1$$

and

$$k_1 = (A_x - D_x)$$

$$k_4 = (A_y - B_y)$$

$$k_2 = (B_x - A_x)$$

$$k_5 = (D_y - A_y)$$

$$k_3 = (A_x - B_x + C_x - D_x) \quad k_6 = (A_y - B_y + C_y - D_y)$$

In order for the diagonals of two quadrilaterals to map to each other, Equations (4) and (5) should be linear in s and t , respectively, for both quadrilaterals. This can be achieved, for instance, if $k_3 = k_6 = 0$, in which case they become parallelograms and the mapping becomes affine.

¹ The (s, t) indices associated with A, B, C, D and P were omitted to improve readability.

Appendix B

Polygons obtained by repeated uniform subdivision of convex quadrilaterals converges to parallelograms.

Proof: Let (x_1, y_1) , (x_2, y_2) , (x_3, y_3) and (x_4, y_4) be the coordinates of the vertices of a quadrilateral in 2D (the extension to 3D is straightforward) shown in Figure 17. Also, let s_1 and s_2 be the slopes of the line segments l_1 and l_2 , respectively. V_i and V_j are two new vertices introduced by the subdivision process and whose coordinates are given by $0.5*(x_1+x_2, y_1+y_2)$ and $0.5*(x_3+x_4, y_3+y_4)$, respectively. In order to show that as the subdivision progresses the sub-quadrilaterals converge to parallelograms, it suffices to show that the value of the slope s_3 of line l_3 defined by V_i and V_j is between the values s_1 and s_2 (i.e., the sides of the sub-quadrilaterals converge to parallel segments). The subdivision along the other dimension is similar.

Thus, without loss of generality, let $s_1 \geq s_2$. If $s_1 \geq s_3 \geq s_2$, according to Figure 17

$$\frac{(y_4 - y_1)}{(x_4 - x_1)} \leq \frac{(y_4 - y_1) + (y_3 - y_2)}{(x_4 - x_1) + (x_3 - x_2)} \leq \frac{(y_3 - y_2)}{(x_3 - x_2)} \quad (6)$$

Verifying Inequality (6) is equivalent to show that (7) is true.

$$\frac{a}{b} \leq \frac{c}{d} \Rightarrow \frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d} \quad (7)$$

From the left hand side of (7), $a \leq b(c/d)$ and $c \geq d(a/b)$.

From the right hand side of (7),

$$\frac{a+c}{b+d} \leq \frac{c}{d} \Leftrightarrow a+c \leq \frac{c}{d}(b+d) \Leftrightarrow a \leq b \frac{c}{d}$$

$$\frac{a}{b} \leq \frac{a+c}{b+d} \Leftrightarrow a+c \geq \frac{a}{b}(b+d) \Leftrightarrow c \geq d \frac{a}{b}$$

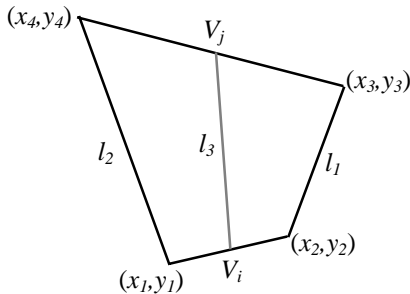


Figure 17. A quadrilateral with arbitrary shape.

References

- [1] ATI Radeon: http://www.ati.com/na/pages/products/pc/radeon64_ddr/index.html.
- [2] J. Blinn. Hyperbolic Interpolation. IEEE Computer Graphics & Applications, July 1992, pages 89-94.

- [3] E. Bier and K. Sloan. Two-Part Texture Mapping. IEEE Computer Graphics and Application, Sept. 1986, pp. 40-53.
- [4] E. Catmull. A Subdivision Algorithm for Computer Display of Curved Surfaces. Ph.D. Dissertation, Department of Computer Science, University of Utah, December 1974.
- [5] M. Cohen and J. Wallace. Radiosity and Realistic Image Synthesis. Academic Press, 1993.
- [6] P. Debevec, C. Taylor, J. Malik. Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach. Proceedings of SIGGRAPH 96, pages 11-20, 1996.
- [7] P. Debevec, Y. Yu and G. Borshukov. Efficient View-Dependent Image-Based Rendering with Projective Texture Mapping. In George Drettakis and Nelson Max, Editors. Rendering Techniques'98, Springer-Verlag/Wien, 1998, pp. 105-116.
- [8] A. Criminisi, I. Reid, A. Zisserman. Single View Metrology. Proceedings ICCV'99, pages 434-442.
- [9] H. Gouraud. Continuous Shading of Curved Surfaces. IEEE Transactions on Computers, Vol. C-20, No. 6, pages 623-629, June 1971.
- [10] P. Heckbert. Fundamentals of Texture Mapping. Master's Thesis, Report No. UCB/CSD 89/516, UC Berkeley.
- [11] P. Heckbert and H. Moreton. Interpolation for Polygon Texture Mapping and Shading. In State of the Art in Computer Graphics. David Rogers and Rae Earnshaw, editors. Springer-Verlag, 1991, pages 101-111.
- [12] B. Lévy and J.L. Mallet. Non-Distorted Texture Mapping for Sheared Triangulated Meshes. Proceedings of SIGGRAPH 98 (Orlando Florida, July 19-24, 1998). In Computer Graphics Proceedings, Annual Conference Series, 1998, ACM SIGGRAPH, pp. 343-352.
- [13] D. Liebowitz, A. Criminisi and A. Zisserman. Creating Architectural Models from Images. Proceedings of EUROGRAPHIC'99 (Milan, Italy, Sept. 7-11, 1999). Computer Graphics Forum, Vol. 18, No. 3, pp. 39-50.
- [14] S. Ma and H. Lin. Optimal Texture Mapping. Proceeding of EUROGRAPH'88 (September 12-16, 1988). David Duce and Pierre Jancene, Editors. North-Holland, Amsterdam, pp. 421-428.
- [15] nVidia GeForce2 Ultra: <http://www.nvidia.com/Products/GeForce2ultra.nsf>.
- [16] J. O'Rourke. Computational Geometry in C. Cambridge University Press, Reprint 1995.
- [17] D. Peachey. Solid Texturing of Complex Surfaces. Proceedings of SIGGRAPH 85 (July 22-26, 1985), Vol. 19, No. 3, (SIGGRAPH'85), pp. 279-286.
- [18] K. Perlin. An Image Synthesizer. Proceedings of SIGGRAPH 85 (July 22-26, 1985), Vol. 19, No. 3, (SIGGRAPH'85), pp. 287-296.
- [19] G. Turk. Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion. Proceedings of SIGGRAPH 91 (July 28 - August 2, 1991), Vol. 25, No. 4, (SIGGRAPH'91), pp. 289-298.
- [20] G. Wolberg. Digital Image Warping. IEEE Computer Society Press, 1990.
- [21] M. Woo et al. OpenGL Programming Guide, third edition. Addison-Wesley, 1999.