Surface Parametrization of Nonsimply Connected Planar Bézier Regions

Orest Shardt

Department of Chemical and Materials Engineering, University of Alberta, Edmonton, Alberta T6G 2V4, Canada

John C. Bowman*

Department of Mathematical and Statistical Sciences, University of Alberta, Edmonton, Alberta T6G 2G1, Canada

Abstract

A technique is described for constructing three-dimensional vector graphics representations of planar regions bounded by cubic Bézier curves, such as smooth glyphs. It relies on a novel algorithm for compactly partitioning planar Bézier regions into nondegenerate Coons patches. New optimizations are also described for Bézier inside–outside tests and the computation of global bounds of directionally monotonic functions over a Bézier surface (such as its axis-aligned bounding box or optimal field-of-view angle). These algorithms underlie the three-dimensional illustration and typography features of the TEX-aware vector graphics language ASYMPTOTE.

Key words: curved triangulation, Bézier surfaces, nondegenerate Coons patches, nonsimply connected domains, inside–outside test, bounding box, field-of-view angle, directionally monotonic functions, vector graphics, PRC, 3D TEX, ASYMPTOTE
2010 MSC: 65D17,68U05,68U15

*Corresponding author
URL: http://www.math.ualberta.ca/∼bowman (John C. Bowman)

To appear in Computer Aided Design

1. Introduction

Recent methods for lifting smooth two-dimensional (2D) font data into three dimensions (3D) have focused on rendering algorithms for the Graphics Processing Unit (GPU) [14]. However, scientific visualization often requires 3D vector graphics descriptions of surfaces constructed from smooth font data. For example, while current CAD formats, such as the PDF-embeddable *Product Representation Compact* (PRC, *précis* in French) [2] format, allow one to embed text annotations, they do not allow text to be manipulated as a 3D entity. Moreover, annotations can only handle simple text; they are not suitable for publication-quality mathematical typesetting.

In this work, we present a method for representing arbitrary planar regions, including text, as 3D surfaces. A significant advantage of this representation is consistency: text can then be rendered like any other 3D object. This gives one complete control over the typesetting process, such as kerning details, and the ability to manipulate text arbitrarily (e.g. by transformation or extrusion) in a compact resolution-independent vector form. In contrast, rendering and mesh-generation approaches destroy the smoothness of the original 2D font data [11].

In focusing on the generation of 3D surfaces from 2D planar data, the emphasis of this work is not on 3D rendering but rather on the underlying procedures for generating vector descriptions of 3D geometrical objects. Vector descriptions are particularly important for online publishing, where no assumption can be made *a priori* about the resolution that will be used to display an image. As explained in Section 2, we focus on surfaces based on polynomial parametrizations rather than nonuniform rational B-splines (NURBS) [7, 18]. In Section 3 we describe a method for splitting an arbitrary planar region bounded by one or more Bézier curves into nondegenerate Bézier patches. This algorithm relies on the optimized Bézier inside–outside test described in Section 4. The implementation of these algorithms in the vector graphics language Asymptote, along with the optimized 3D sizing algorithms presented in Section 5, is discussed in Section 6.

Using a compact vector format instead of a large number of polygons to represent manifolds has the advantage of reduced data representation (essential for the storage and transmission of 3D scenes) and the possibility, using relatively few control points, of exact or nearly exact geometrical descriptions of mathematical surfaces.

2. Bézier vs. NURBS Parametrizations

The atomic graphical objects in PostScript and PDF, Bézier curves and surfaces, are composed of piecewise cubic polynomial *segments* and tensor product *patches*, respectively. A segment $\boldsymbol{\gamma}(t) = \sum_{i=0}^{3} B_i(t)\boldsymbol{P}_i$ has four control points $\boldsymbol{P}_i$, whereas a surface patch is defined by sixteen control points $\boldsymbol{P}_{ij}$:

$$\boldsymbol{P}(u,v) = (x(u,v), y(u,v)) = \sum_{i,j=0}^{3} B_i(u)B_j(v)\boldsymbol{P}_{ij}.$$

Here $B_i(u) = \binom{3}{i}u^i(1-u)^{3-i}$ is the ith cubic Bernstein polynomial. Just as a Bézier segment passes through its two end control points, a Bézier patch necessarily passes through its four corner control points. These special control points are called *nodes.* A *straight* segment is one in which the control points are colinear and the derivative of the Bézier parametrization is never zero (i.e. the control points are arranged in the same order as their indices). A *closed* curve $\{\boldsymbol{\gamma}_i\}_{i=0}^{n-1}$ satisfies $\boldsymbol{\gamma}_0(0) = \boldsymbol{\gamma}_{n-1}(1)$.

It is often desirable to project a 3D scene to a 2D vector graphics format understood by a web browser or high-end printer. Although NURBS are popular in computer-aided design [7] because of the additional degrees of freedom introduced by weights and general knot vectors, these benefits are tempered by both the lack of support for NURBS in popular 2D vector graphics formats (PostScript, PDF, SVG, EMF) and the algorithmic simplifications afforded by specializing to a Bézier parametrization. Bézier curves are also commonly used to describe glyph outlines. For such reasons, we restrict our attention in this work to (polynomial) Bézier curves and surfaces.

Unlike their Bézier counterparts, NURBS are invariant under perspective projection. This is only an issue if projection is done before the rendering stage, as is necessary when a 2D vector representation of a curve or surface is constructed solely from the 2D projection of its control points. It is therefore somewhat ironic that NURBS are much less widely implemented in 2D vector graphics formats than in 3D. In 3D vector graphics applications, projection to 2D is always deferred until rendering time, so that the invariance of NURBS under nonaffine projection is irrelevant. Moreover, while NURBS provide exact parametrizations of familiar conic sections and quadric surfaces, nontrivial manifolds still need to be approximated as piecewise unions of underlying exact primitives.

3. Partitioning Curved 2D Regions

In 3D graphics, text is often displayed with bit-mapped images, textures, or polygonal mesh approximations to smooth font character curves. To allow viewing of smooth text at arbitrary magnifications and locations, a nonpolygonal surface that preserves the curvature of the boundary curves is required. While it is easy to fill the outline of a smooth character in 2D, filling a 3D planar surface requires more sophisticated methods. One approach involves using surface filling algorithms for execution on GPUs [14]. When a vector, rather than a rendered, image is desired, a preferable alternative is to represent the text as a parametrized surface.

Methods based on common surface primitives in 3D modelling and rendering can be used to describe planar regions. One method trims the domain of a planar surface to the desired shape [16]. While that approach is feasible, given adequate software support for trimming, this work describes a different approach, where each symbol is represented as a set of planar Bézier patches. We call this procedure *bezulation* since it involves a process similar to the triangulation of a polygon but uses cubic Bézier patches instead of triangles. To generate a surface representing the region bounded by a set of *simple closed* Bézier curves (intersecting only at the end points), algorithms were developed for (i) expressing a simply connected 2D region as a union of Bézier patches and (ii) breaking up a nonsimply connected region into simply connected regions. (Self-intersecting curves can be handled by splitting at the intersection points.) These algorithms allow one to express text surfaces conveniently as Bézier patches.

Bezulation of a simply connected planar region involves breaking the region up into patches bounded by closed Bézier curves with four or fewer segments. This is performed by the routine `bezulate` (see Algorithm 1) using an adaptation of a naïve triangulation algorithm, modified to handle curved edges, as illustrated in Figure 1.

```
Input: simple closed curve C
Output: array of closed curves A
while C.segments > 4 do
    found ← false;
    for n = 3 to 2 do
        for i = 0 to C.segments-1 do
            L ← line segment between nodes i and i + n of C;
            if countIntersections(C,L) = 2 and midpoint of L is
            inside C then
                p ← subpath of C from node i to i + n;
                q ← subpath of C from node i + n to i + C.segments;
                A.push(p+L);
                C ← L + q;
                found ← true;
                break;
            end
        end
        if found then
            break;
        end
    end
    if not found then
        refine C by inserting an additional node at the parametric
        midpoint of each segment;
    end
end
```

Algorithm 1: `bezulate` partitions a simply connected region.

A line segment lies within a closed curve when it intersects the curve only at its endpoints and its midpoint lies strictly inside the curve. If after checking all connecting line segments between nodes separated by $n = 3$ or $n = 2$ segments, none of them lie entirely inside the shape, the original curve is refined by dividing each segment of the curve at its parametric midpoint, as illustrated in Fig. 2. The bezulation process then continues with the refined curve. This algorithm can be modified to subdivide more optimally, for example, to avoid elongated patches that sometimes lead to rendering problems.

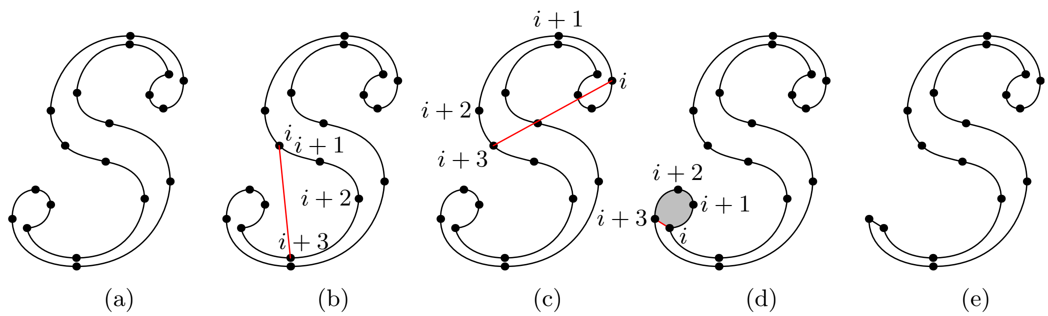

Figure 1: The `bezulate` algorithm. Starting with the original curve (a), several possible connecting line segments (shown in red) between nodes separated by $n = 3$ or $n = 2$ segments are tested. Connecting line segments are rejected if they do not lie entirely inside the original curve. This occurs when the midpoint is not inside the curve (b) or when the connecting line segment intersects the curve more than twice (c). If a connecting line segment passes both tests, the shaded section is separated (d) and the algorithm continues with the remaining curve (e).

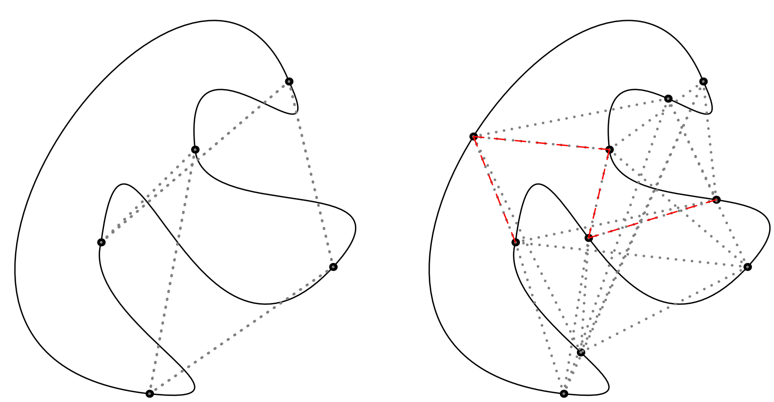

Figure 2: The left-hand figure illustrates a case where all line segments between nodes separated by two or three segments (dotted lines) are rejected in the `bezulate` algorithm. The right-hand curve shows that several connections (dashed red lines) become possible upon subdivision.

If the region is convex, Algorithm 1 is easily seen to terminate: all connecting line segments are admissible, and each patch removal decreases the number of points in the curve. Moreover, from the point of view of Algorithm 1, upon sufficient subdivision a non-convex region eventually becomes indistinguishable from a polygon, in which case the algorithm reduces to a straightforward polygonal triangulation.

3.1. Nonsimply Connected Regions

Since the `bezulate` algorithm requires simply connected regions, nonsimply connected regions must be handled specially. The "holes" in a nonsimply connected domain can be removed by partitioning the domain into a set of simply connected regions, each of which can then be bezulated.

For convenience we define a *top-level curve* to be a curve that is not contained inside any other curve and an *outer (inner) curve* to be the outer (inner) boundary of a filled region. With these definitions, the glyph "%" has two inner curves and two top-level curves that are also outer curves.

The algorithm proceeds as follows. First, to determine the topology of the region, the curves are sorted according to their relative insidedness, as determined by the nonzero winding number rule. Since the curves are assumed to be simple, any point on an inner curve can be used to test whether that curve is inside another curve. The result of this sorting is a collection of top-level curves grouped with the curves they surround. Each of these groups is treated independently.

Figure 4 illustrates the `partition` routine (see Algorithm 2). Each group is examined recursively to identify regions bounded by inner and outer curves. First, the inner curves in the group are sorted topologically to find the inner curves that are top-level curves with respect to the other inner curves. The inner curves that are not top-level curves are processed with a recursive call to `partition`. The nonsimply connected region between the outer (top-level) curve and the inner (top-level) curves is now split into simply connected regions. This is illustrated in Figure 3. The intersections of the inner and outer curves with a line segment from a point on an inner curve to a point on the outer curve are found using a numerically robust explicit cubic root solver. Consecutive intersections of this line segment, at points A and B, on the inner and outer curves, respectively, are selected. Let t_B be the value of the parameter used to parameterize the outer curve at B. Starting with $\Delta = 1$, Δ is halved until the line segment $\overline{AC}$, where C is the point on the outer curve at $t_B + \Delta$, does not intersect the outer curve more than once, does not intersect any inner curve (other than once at A), and the region bounded by $\overline{AB}$, $\overline{AC}$, and $\overset{\frown}{BC}$ does not contain any inner curves. Once Δ and the point C have been found, the outer curve, less the segment between B and C, is merged with $\overline{BA}$, followed by the inner curve and then $\overline{AC}$. The region bounded by $\overline{AB}$, $\overline{AC}$, and $\overset{\frown}{BC}$ is a simply connected region. Additional simply connected regions are found when the outer curve is merged with the other inner curves. Once the merging with all inner curves has been completed, the outer curve becomes the boundary of the final simply connected region.

The recursive algorithm for partitioning nonsimply connected regions into simply connected regions is summarized below. The function `sort` returns groups of top-level curves and the curves they contain. However, it is not recursive; the inner curves are not sorted. The function `merge` returns the simply connected regions formed from the single outer curve and multiple inner curves that are supplied to it.

```
Input: array of simple closed curves C
Output: array of closed curves A
foreach group of nested curves G in sort(C) do
    innerGroups ← sort(G.innerCurves);
    foreach group of nested curves H in innerGroups do
        A.push(partition(H.innerCurves));
    end
    A.push(merge(G.toplevel, top-level curves of all groups in
    innerGroups));
end
return A;
```

Algorithm 2: `partition` splits nonsimply connected regions into simply connected regions. The pseudo-code functions `sort` and `merge` are described in the text.

Although the rendering technique of Ref. [14] could be modified to produce Bézier patches, it appears to generate more patches than `bezulate`. For example, the "e" shown in Fig. 3 of Ref. [14] corresponds to roughly twice as many (4-segment) patches as the ten patches generated by `bezulate` for the "*e*" in Fig. 7. Our interest is in compact 3D vector representations, where the objective is to minimize the number of generated patches. In contrast, in real-time rendering, one aims to minimize overall execution time. An inter-

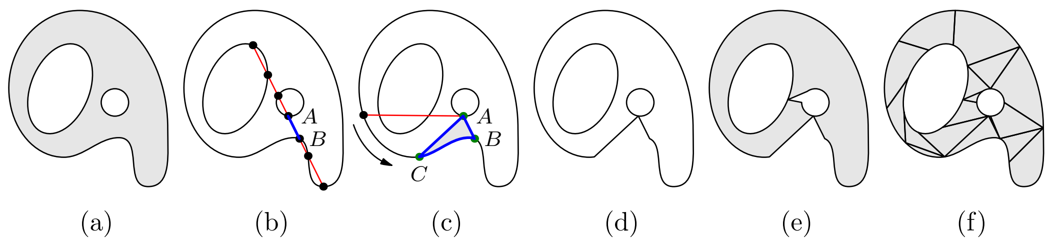

Figure 3: Splitting of non-simply connected regions into simply connected regions. Starting with a non-simply connected region (a), the intersections between each curve and an arbitrary line segment from a point on an inner curve to the outer curve are found (b). Consecutive intersections of this line segment, at points A and B, on the inner and outer curves, respectively, identify a convenient location for extracting a region. One searches along the outer curve for a point C such that the line segment AC intersects the outer curve no more than once, intersects an inner curve only at A, and determines a region ABC between the inner and outer curves that does not contain an inner curve. Once such a region is found (c), it is extracted (d). This extraction merges the inner curve with the outer curve. The process is repeated until all inner curves have been merged with the outer curve, leaving a simply connected region (e) that can be split into Bézier surface patches. The resulting patches and extracted regions are shaded in (f).

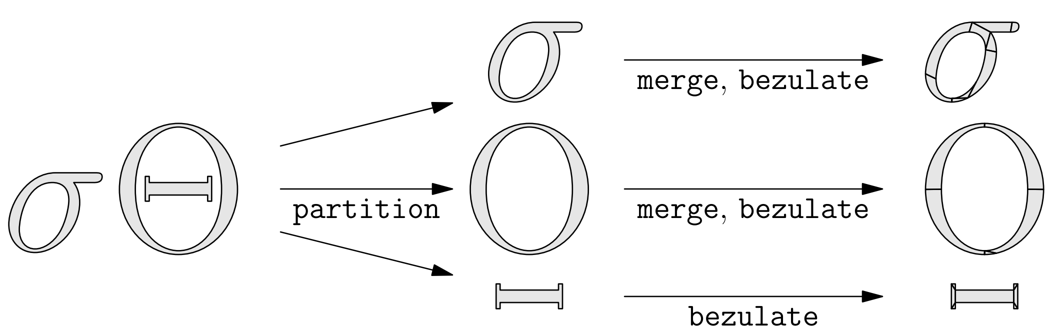

Figure 4: Illustration of the `partition` algorithm. The five curves that define the outlines of the Greek characters σ and Θ are passed in a single array to `partition`.

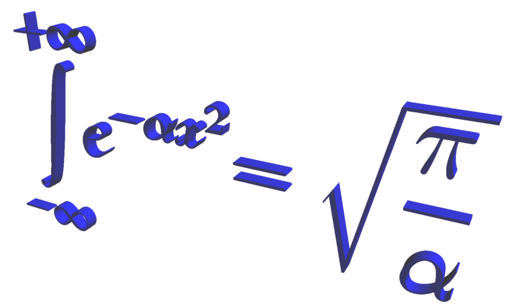

Figure 5: Application of the `bezulate` and `partition` algorithms to lift the Gaussian integral to three dimensions.

Figure 6: Zoomed view of Figure 5 generated from the same vector graphics data. The smooth boundaries of the characters emphasize the advantage of a 3D vector font description.

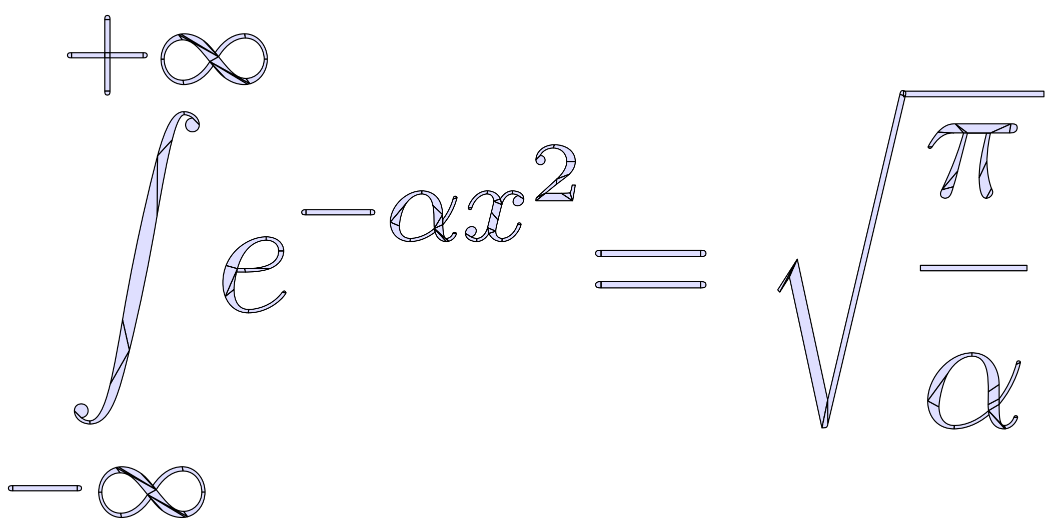

Figure 7: Subpatch boundaries for Figure 5 as determined by the `bezulate` and `partition` algorithms.

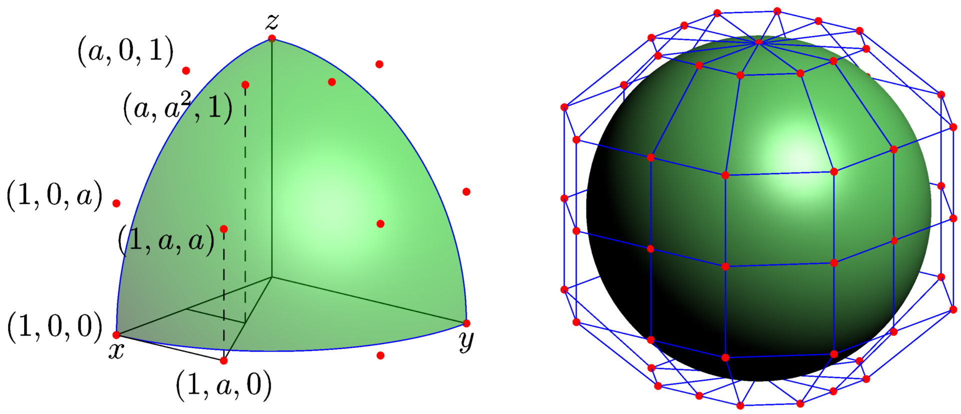

Figure 8: Bézier approximation to a unit sphere. The red dots indicate control points.

esting future research project would be the development of Bézier versions of more advanced triangulation algorithms that efficiently incorporate, for example, feature-based decomposition [11].

The routines `bezulate` and `partition` were used to typeset the TEX equation

$$\int_{-\infty}^{+\infty} e^{-\alpha x^2} = \sqrt{\frac{\pi}{\alpha}}$$

in the interactive 3D diagram shown in Figure 5 and magnified, to emphasize the smooth font boundaries, in Figure 6. The computed subpatch boundaries are indicated in Figure 7.

The 8-patch Bézier approximation to a sphere [17] in Figure 8 (where $a = \frac{4}{3}(\sqrt{2} - 1)$ yields an accuracy of 0.052%), illustrates how `bezulate` can be used to lift TEX to three dimensions. Referring to the interactive 3D PDF version of this article[1] one sees that the labels in Figure 8 have been programmed to rotate interactively so that they always face the camera; this feature, implemented with `Javascript`, is known as *billboard interaction*.

3.2. Nondegenerate Planar Bézier Patches

The `bezulate` algorithm described above decomposes regions bounded by closed curves (according to the nonzero winding number rule) into subregions

[1] See `http://asymptote.sourceforge.net/articles/`.

bounded by closed curves with four or fewer segments. Further steps are required to turn these subregions into nondegenerate Bézier patches. First, if the interior angle between the incoming and outgoing tangent directions at a node is greater than 180°, the boundary curve is split at this node by following the interior angle bisector to the first intersection with the path. This is done to guarantee that the patch normal vectors at the nodes all point in the same direction. Next, curves with less than four segments are supplemented with null segments (four identical control points) to bring their total number of segments up to four. A closed curve with four segments defines the twelve boundary control points of a Bézier patch in the x–y plane. The remaining four interior control points $\{\boldsymbol{P}_{11}, \boldsymbol{P}_{12}, \boldsymbol{P}_{21}, \boldsymbol{P}_{22}\}$ are then chosen to satisfy the Coons interpolation [6, 8, 1]

$$\begin{aligned}\boldsymbol{P}(u,v) = \sum_{i=0}^{3} &[(1-v)B_i(u)\boldsymbol{P}_{i,0} + vB_i(u)\boldsymbol{P}_{i,3} + (1-u)B_i(v)\boldsymbol{P}_{0,i} + uB_i(v)\boldsymbol{P}_{3,i}] \\ &-(1-u)(1-v)\boldsymbol{P}_{0,0} - (1-u)v\boldsymbol{P}_{0,3} - u(1-v)\boldsymbol{P}_{3,0} - uv\boldsymbol{P}_{3,3}.\end{aligned}$$

The resulting mapping $\boldsymbol{P}(u,v)$ need not be bijective [19, 21, 22, 13], even if the corner control points form a convex quadrilateral (despite the fact that a Coons patch for a convex polygon is always nondegenerate). In terms of the 2D scalar cross product $\boldsymbol{p}\times\boldsymbol{q} = p_x q_y - p_y q_x$, the Coons patch is seen to be a diffeomorphism of the unit square $D = [0,1] \times [0,1]$ if and only if the Jacobian

$$J(u,v) = \frac{\partial(x,y)}{\partial(u,v)} = \boldsymbol{\nabla}_u x \times \boldsymbol{\nabla}_v y = \sum_{i,j,k,\ell=0}^{3} B_i'(u)B_j(v)B_k(u)B_\ell'(v)\boldsymbol{P}_{ij}\times\boldsymbol{P}_{k\ell}$$

(the z component of the corresponding 3D normal vector) is sign definite [19]. Since $J(u,v)$ is a continuous function of its arguments, this means that J must not vanish anywhere on D. A sign reversal of the Jacobian can manifest itself as an outright overlap of the region bounded by the curve or as an internal multivalued wrinkle, as illustrated in Figure 9. Rendering problems, such as the black smudges visible in Figures 9(b) and (e), can occur where isolines collide.

Randrianarivony and Brunnett [19] (and later H. Lin *et al.* [13]) describe sufficient conditions for $J(u,v)$ to be nonzero throughout D. In the case of a cubic Bézier patch, the 36 quantities

$$T_{pq} = \sum_{i+k=p} \sum_{j+\ell=q} \boldsymbol{U}_{i,j}\times\boldsymbol{V}_{k,\ell}\binom{2}{i}\binom{3}{k}\binom{3}{j}\binom{2}{\ell} \qquad p,q = 0,1,\ldots,5,$$

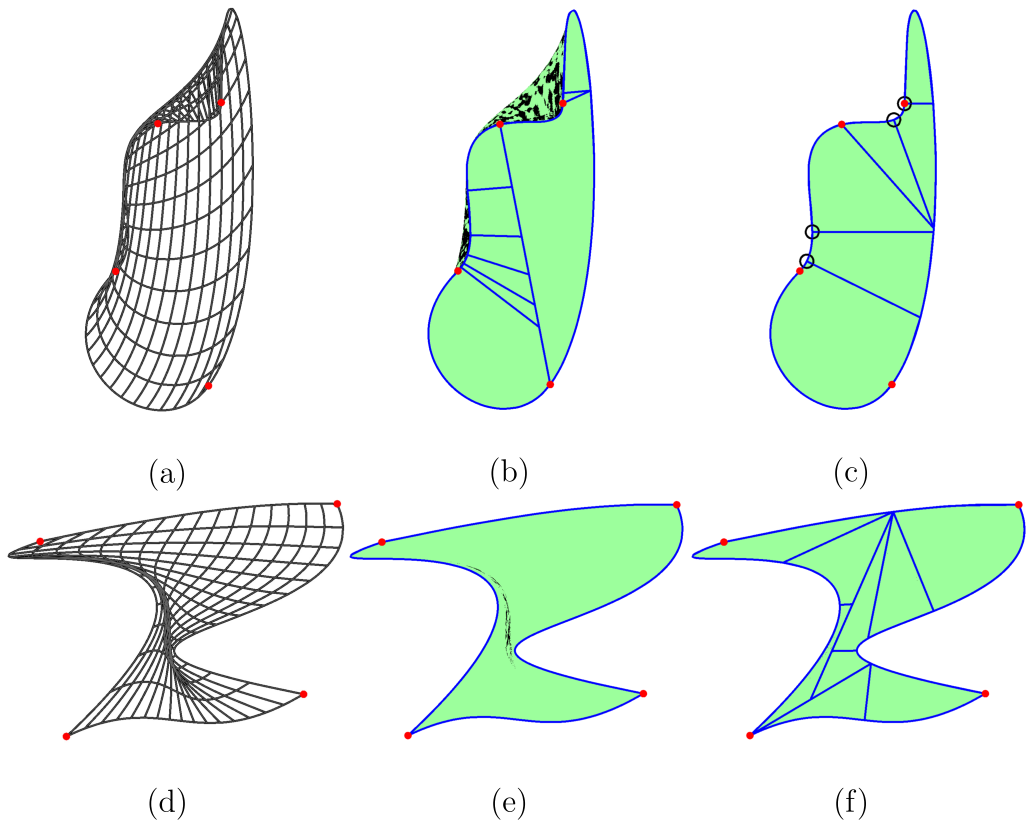

Figure 9: Degeneracy in a Coons patch. The dots indicate corner control points (nodes) and the open circles indicate the points of greatest degeneracy on the boundary, as determined by the quartic root solver: (a) overlapping isoline mesh; (b) overlapping patch; (c) nonoverlapping subpatches; (d) internally degenerate isoline mesh; (e) internally degenerate patch; (f) nondegenerate subpatches.

where $\boldsymbol{U}_{i,j} = \boldsymbol{P}_{i+1,j} - \boldsymbol{P}_{i,j}$ and $\boldsymbol{V}_{i,j} = \boldsymbol{P}_{i,j+1} - \boldsymbol{P}_{i,j}$, are required to be of the same sign. This follows from the fact that $J(u,v) = \sum_{p,q=0}^{5} T_{pq} u^p v^q (1-u)^{5-p}(1-v)^{5-q}$.

Randrianarivony *et al.* show further that every degenerate Coons patch can be decomposed into a finite union of nondegenerate subpatches (some with reversed orientation). However, the adaptive subdivision algorithm they propose to exploit this fact does not prescribe an optimal boundary point at which to do the splitting. A better algorithm is based on the following elementary theorem, which provides a practical means of detecting Coons patches with degenerate boundaries.

Theorem 1 (Nondegenerate Boundary)**.** *Consider a closed counter-clockwise oriented four-segment curve p in the x–y plane such that the interior angles formed by the incoming and outgoing tangent vectors at each node are less than or equal to $180°$. Let $J(u,v)$ be the Jacobian of the corresponding Coons patch constructed from p, with control points $\boldsymbol{P}_{ij}$, and define the fifth-degree polynomial*

$$f(u) = \sum_{i,j=0}^{3} B_i'(u) B_j(u) \boldsymbol{P}_{i,0} \times (\boldsymbol{P}_{j,1} - \boldsymbol{P}_{j,0}).$$

If $f(u) \geq 0$ whenever $f'(u) = 0$ on $u \in (0,1)$, then $J(u,0) \geq 0$ on $[0,1]$. Otherwise, the minimum value of $J(u,0)$ occurs at a point where $f'(u) = 0$.

Proof. First we note, since $B_1'(0) = -B_0'(0) = 3$ and $B_2'(0) = B_3'(0) = 0$, that $J(u,0) = 3f(u)$ and

$$J(0,0) = 3f(0) = 9(\boldsymbol{P}_{1,0} - \boldsymbol{P}_{0,0}) \times (\boldsymbol{P}_{0,1} - \boldsymbol{P}_{0,0}) \geq 0$$

since this is the cross product of the outgoing tangent vectors at $\boldsymbol{P}_{0,0}$. Likewise, $J(1,0) = 3f(1) \geq 0$. We know that the continuous function f must achieve its minimum value on $[0,1]$ at some $u \in [0,1]$. If f were negative somewhere in $(0,1)$ we could conclude that $f(u) < 0$, so that $u \in (0,1)$, and hence f would have an interior local minimum at u, with $f'(u) = 0$. But this is a contradiction, given that $f(u) \geq 0$ whenever $f'(u) = 0$. □

The significance of Theorem 1 is that it affords a means of detecting a point u on the boundary where the Jacobian is most negative. This requires finding roots of the quartic polynomial

$$f'(u) = \sum_{i,j=0}^{3} [B_i''(u) B_j(u) + B_i'(u) B_j'(u)] \boldsymbol{P}_{i,0} \times (\boldsymbol{P}_{j,1} - \boldsymbol{P}_{j,0}).$$

$$\begin{pmatrix} 5-20u+30u^2-20u^3+5u^4 & -3+24u-54u^2+48u^3-15u^4 & -6u+27u^2-36u^3+15u^4 & -3u^2+8u^3-5u^4 \\ -7+36u-66u^2+52u^3-15u^4 & 3-36u+108u^2-120u^3+45u^4 & 6u-45u^2+84u^3-45u^4 & 3u^2-16u^3+15u^4 \\ 2-18u+45u^2-44u^3+15u^4 & 12u-63u^2+96u^3-45u^4 & 18u^2-60u^3+45u^4 & 8u^3-15u^4 \\ 2u-9u^2+12u^3-5u^4 & 9u^2-24u^3+15u^4 & 12u^3-15u^4 & 5u^4 \end{pmatrix}$$

Table 1: Coefficients of the polynomials $M_{ij} = (B_i'' B_j + B_i' B_j')/3$.

The coefficients of this quartic polynomial can be computed using the polynomials $M_{ij} = (B_i'' B_j + B_i' B_j')/3$ tabulated in Table 1. The method of Neumark [15], which relies on numerically robust cubic and quadratic root solvers, is then used to find algebraically all real roots of the quartic equation $f'(u) = 0$ that lie in $(0, 1)$. The Jacobian is computed at each of these points; if it is negative anywhere, the point where it is most negative is determined. The patch is then split along an interior line segment perpendicular to the tangent vector at this point. The next intersection point of the patch boundary with this line is used to split the patch into two pieces. Each of these pieces is then treated recursively (beginning with an additional call to `bezulate`, should the new boundary curve happen to have five segments).

If a patch possesses only internal degeneracies, like the one in Figure 9(d), the patch boundary is arbitrarily split into two closed curves, say along the perpendicular to the midpoint of some nonstraight side. The blue lines in Figures 9(b) and (f) illustrate such a midpoint splitting. The arguments of Randrianarivony *et al.* [19] establish that only a finite number of such subdivisions will be required to obtain a nondegenerate patch. Nondegenerate subpatches oriented in the direction opposite to the normal vector corresponding to the original oriented curve should be discarded to avoid rendering interference with correctly aligned overlying subpatches.

The blue lines in Figure 9(c) show that our quartic algorithm generates six subpatches, a substantial improvement over the nine subpatches produced by adaptive midpoint subdivision [19] in Figure 9(b). Figure 9(c) also emphasizes the ability of the quartic root algorithm to detect the optimal (most degenerate) points (circled) for splitting the boundary curve. As mentioned earlier, in both cases, it is possible that splitting can lead to curves with five segments. Such curves are split further by the `bezulate` algorithm so that any degeneracy of the resulting subpatches can be addressed.

Since our algebraic quartic root solver is explicit, optimal subdivision, which splits at the point of greatest degeneracy, can be implemented more efficiently than naïve midpoint subdivision. In our high-level ASYMPTOTE implementation, the costs of adaptive midpoint subdivision for Figures 9(b) and Figure 9(f) were approximately the same. Using optimal subdivision in Figure 9(c) was 34% faster than adaptive midpoint splitting, whereas there was only 2% additional overhead in checking for boundary degeneracy in Figure 9(f) (which possesses only internal degeneracy). Patches having only internal degeneracy arise relatively rarely in practice, but when they do, the subpatches obtained by adaptive midpoint subdivision also tend to exhibit internal degeneracy. Once internal degeneracy has been detected in a patch, we find that it is typically more efficient not to check its degenerate subpatches for boundary degeneracy (otherwise the overhead in checking for boundary degeneracy in Figure 9(f) would grow to 50%). Of course, since our interest is not in real-time rendering but in surface generation, the real advantage of optimal subdivision is that it can significantly reduce the number of generated patches (e.g. Figure 9(c) has one-third fewer patches than Figure 9(b)).

4. An Optimized Bézier Inside–Outside Test

Although PostScript has an `infill` function for testing whether a particular point would be painted by the PostScript `fill` command, this is only an approximate digitized test corresponding to the resolution of the output device. Our `bezulate` routine requires a vector graphics algorithm, one that yields the winding number of an arbitrary closed piecewise Bézier curve about a given point. It is convenient to define the winding number contribution of a smooth (not necessarily closed) curve γ about z *via* the complex line integral

$$\nu(\gamma, z) = \frac{1}{2\pi i} \int_\gamma \frac{1}{\zeta - z}\, d\zeta,$$

allowing the usual winding number of a closed curve $\cup_i \gamma_i$ to be expressed as $\sum_i \nu(\gamma_i, z)$.

A straightforward generalization of the standard ray-to-infinity method for computing winding numbers of a polygon about a point requires the solution of a cubic equation. As is well known, the latter problem can become numerically unstable as two or three roots begin to coalesce. While a conventional ray-curve (or ray-patch) intersection algorithm based on recursive subdivision [16] could be employed to count intersections, this typically entails excessive subdivision.

A more efficient but still robust subdivision method for computing the winding number of a closed Bézier curve arises from the topological observation that if a point z lies outside the convex hull of a Bézier segment γ, then γ can be deformed to a straight line segment L between its endpoints by a continuous function $F : \gamma \times [0,1] \mapsto L$ such that $F(\gamma(t), s) \neq z$ for all $(s,t) \in [0,1] \times [0,1]$ (i.e. without crossing z). Cauchy's theorem guarantees that the winding number contribution $\nu(\gamma, z)$ is unchanged by this deformation.

A given point will typically lie outside the convex hull of most segments of a Bézier curve. The orientation of these segments relative to the given point can be quickly and robustly determined, just as in the usual ray-to-infinity method for polygons (see e.g. [9]), to determine the contribution, if any, to the winding number. For this purpose, Jonathan Shewchuk's public-domain adaptive precision predicates for computational geometry [20] are highly recommended.

In the infrequent case where z lies on or inside the convex hull of a segment, de Casteljau subdivision is used to split the Bézier segment about its parametric midpoint. Typically the convex hulls of the resulting subsegments will overlap only at their common control point, so that z can lie strictly inside at most one of these hulls. This observation is responsible for the efficiency of the algorithm: as illustrated in Figure 10, one continues subdividing until the point is outside the convex hull of both segments or until machine precision is reached.

The orientation of segments whose convex hulls do not contain z can be handled by using the topological deformation property together with adaptive precision predicates. Denoting by `straightContribution(P,Q,z)` the usual ray-to-infinity method for determining the winding number contribution of a line segment $\overline{\mathsf{PQ}}$ relative to a point `z`, the contribution from a Bézier

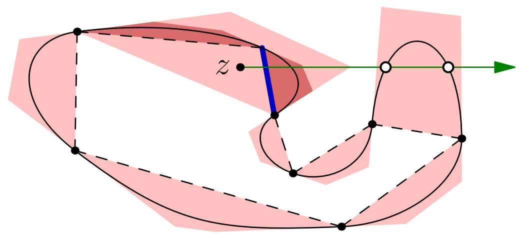

Figure 10: The `BézierWindingNumber` algorithm. Since z lies inside the convex hull of one Bézier segment, indicated by a light shaded region, that segment must be subdivided. On subdivision, z now lies outside the convex hulls of the subsegments, indicated by the dark shaded regions; each subsegment γ may be continuously deformed to a straight line segment between its endpoints without crossing z. The usual polygon inside–outside test may then be applied: the green ray establishes a winding number contribution of $+1$ due to the orientation of z with respect to the thick blue line. The intersections indicated by open circles can be ignored because the relevant Bézier segment can be deformed to a line segment that does not intersect the ray.

segment S can be computed as `curvedContribution(S,z)` (Algorithm 3).

```
Input: segment S, pair z
Output: winding number contribution of S about z
W ← 0;
if z lies within or on the convex hull of S then
    foreach subsegment s of S do
        W ← W + curvedContribution(s,z);
    end
else
    W ← W + straightContribution(S.beginpoint,S.endpoint,z);
end
return W;
```

Algorithm 3: `curvedContribution(S,z)` determines the winding number contribution from a Bézier segment S about z.

The winding number for a closed curve p about z may then be evaluated with the algorithm `bézierWindingNumber(C,z)` (Algorithm 4).

```
Input: curve C, pair z
Output: winding number of C about z
W ← 0;
foreach segment S of C do
    if S is straight then
        W ← W + straightContribution(S.beginpoint,S.endpoint,z);
    else
        W ← W + curvedContribution(S,z);
    end
end
return W;
```

Algorithm 4: `bézierWindingNumber(C,z)` computes the winding number of a closed Bézier curve C about z.

A practical simplification of the above algorithm is the widely used optimization of testing whether a point is inside the 2D (axis-aligned) bounding box of the control points rather than their convex hull. Since the convex hull of a Bézier segment is contained within the bounding box of its control points, one can replace "convex hull" by "control point bounding box" in the above algorithm without modifying its correctness. One can easily check numerically that the cost of the additional spurious subdivisons is well offset by the computational savings in testing against the control point bounding box.

5. Global Bounds of Directionally Monotonic Functions

We now present efficient algorithms for computing global bounds of real-valued directionally monotonic functions $f : \mathbb{R}^3 \to \mathbb{R}$ defined over a Bézier patch $\boldsymbol{P}$ parametrized by $(u, v) \in [0, 1] \times [0, 1]$. By *directionally monotonic* we mean that f is a monotonic function of each of the three Cartesian directions while holding the other two fixed; if f is differentiable this means that f has sign-semidefinite partial derivatives. These algorithms can be used to compute the 3D (axis-aligned) bounding box of a Bézier surface, the bounding box of its 2D projection, or the optimal field-of-view angle for sizing a 3D scene (see Fig. 11). The key observation is that the convex hull property of a Bézier patch holds independently in each direction and even under inversions like $z \to 1/z$.

A naïve approach (subdivision without regard to monotonicity) to computing the bounding box of a Bézier patch requires subdivision whenever the subpatch bounding boxes overlap in any of the three Cartesian directions. However, the number of required subdivisons can be greatly reduced by decoupling the three directions: in Algorithm 5, the problem is split into finding the maximum and minimum of the three *Cartesian axis projections* $f(x, y, z) = x$, $f(x, y, z) = y$, and $f(x, y, z) = z$ evaluated over the patch. This requires a total of six applications of Algorithm 5. The extrema of these special choices for f over a polyhedron $\mathcal{C}$ occur at vertices of $\mathcal{C}$.

More general choices of directionally monotonic functions f are also of interest. For example, to determine the bounding box of the 2D perspective projection (based on similar triangles) of a surface, one can apply Algorithm 6 in eye coordinates to the functions $f(x, y, z) = x/z$ and $f(x, y, z) = y/z$. This is useful for sizing a 3D object in terms of its 2D projection. For example, these functions were used to calculate the optimal field-of-view angle 13.4° for the Klein bottle shown in Figure 11.

For an arbitrary directionally monotonic function f and polyhedron $\mathcal{C}$, we observe that

$$\boldsymbol{P} \subset \mathcal{C} \Rightarrow f(\boldsymbol{P}) \subset f(\mathcal{C}) \subset [\min_{\partial\mathcal{C}} f, \max_{\partial\mathcal{C}} f], \tag{1}$$

noting that the function value at each point $\boldsymbol{v} \in \boldsymbol{P}$ is bounded by the function values at the nearest two intersection points of $\partial\mathcal{C}$ with a line through $\boldsymbol{v}$ in the x, y, or z direction.

Our algorithms exploit Eq. (1) together with de Casteljau's subdivision algorithm and the fact that a Bézier patch is confined to the convex hull of its control points. However, a patch is only guaranteed to intersect its convex hull at the four corner nodes.

For the special case where f is a projection onto the Cartesian axes, the function $\texttt{CartesianMax}(f, \boldsymbol{P}, f(\boldsymbol{P}_{00}), d)$ given in Algorithm 5 computes the global maximum M of a Cartesian axis projection $f : \mathbb{R}^3 \to \mathbb{R}$ over a Bézier patch $\boldsymbol{P}$ to recursion depth d. Here, the value $f(\boldsymbol{P}_{00})$ provides a convenient starting value (lower bound) for M; if the maximum of a surface consisting of several patches is desired, the value of M from previous patches is used to seed the calculation for the subsequent one. The algorithm exploits the fact that the extrema of each coordinate over the convex hull $\mathcal{C}$ of $\{\boldsymbol{P}_{ij}\}$ occur at vertices of $\mathcal{C}$. First, one replaces M by the maximum of f evaluated at the four corner nodes and the previous value of M. If the maximum of the

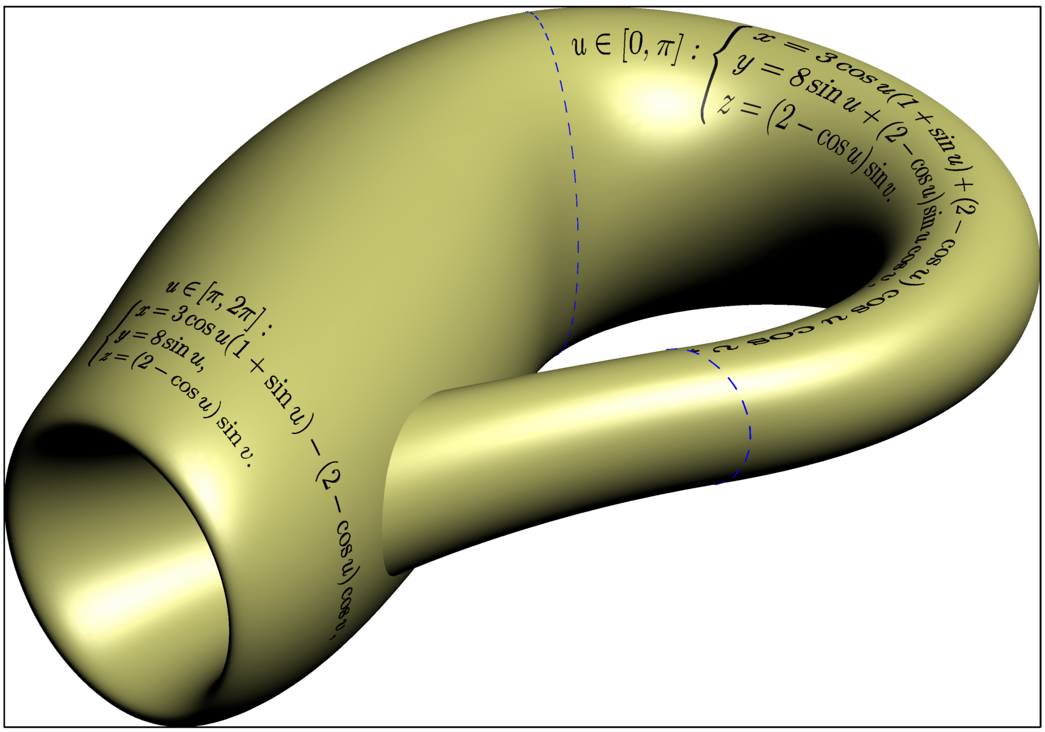

Figure 11: A Bézier approximation to a projection of a four-dimensional Klein bottle to three dimensions. The `FunctionMax` algorithm was used to determine the optimal field of view for this symmetric perspective projection of the scene from the camera location $(25.09, -30.33, 19.37)$ looking at $(-0.59, 0.69, -0.63)$. The extruded 3D TEX equations embedded onto the surface provide a parametrization for the surface over the domain $u \times v \in [0, 2\pi] \times [0, 2\pi]$.

function evaluated at the remaining 12 control points is less than or equal to M, the subpatch can be discarded (by Eq. 1, noting that the maximum of $f(\mathcal{C})$ occurs at a control point and hence cannot exceed M). Otherwise, the patch is subdivided along the $u = v = 1/2$ isolines and the process is repeated using the new value of M. The method quickly converges to the global maximum of f over the entire patch.

```
Input: real function f(triple), patch P, real M, integer depth
Output: real M
M ← max(M, f(P_00), f(P_03), f(P_30), f(P_33));
if depth = 0 then
 | return M;
end
V ← max(f(P_01), f(P_02), f(P_10), f(P_11), f(P_12), f(P_13),
        f(P_20), f(P_21), f(P_22), f(P_23), f(P_31), f(P_32));
if V ≤ M then
 | return M;
end
foreach subpatch S of P do
 | M ← max(M, FunctionMax(f, S, M, depth − 1));
end
return M;
```

Algorithm 5: `CartesianMax(f,`$\boldsymbol{P}$`,M,depth)` returns the maximum of M and the global bound of a Cartesian component f of a Bézier patch $\boldsymbol{P}$ evaluated to recursion level depth.

For a general directionally monotonic function f (consider $f(x, y, z) = xy$ over $\mathcal{C} = \{(x, y, 0) : 0 \leq x \leq 1,\ 0 \leq y \leq x\}$), the maximum of $f(\mathcal{C})$ need not occur at vertices of $\mathcal{C}$: one instead needs to examine the function value at the appropriate vertex of the bounding box of $\mathcal{C}$. For example, if f is a monotonic increasing function on $\mathcal{C}$ in each of the three Cartesian directions,

$$\boldsymbol{P} \subset \mathcal{C} \subset \mathrm{box}(\boldsymbol{a}, \boldsymbol{b}) \Rightarrow f(\boldsymbol{P}) \subset f(\mathcal{C}) \subset [f(\boldsymbol{a}), f(\boldsymbol{b})], \tag{2}$$

where $\mathrm{box}(\boldsymbol{a}, \boldsymbol{b}) = \{\boldsymbol{v} : a_i \leq v_i \leq b_i,\ i = 1, 2, 3\}$. This follows by successively comparing the function values at each point $\boldsymbol{v} \in \boldsymbol{P}$ with $f(a_1, v_2, v_3)$, $f(a_1, a_2, v_3)$, and $f(a_1, a_2, a_3)$, along with $f(b_1, v_2, v_3)$, $f(b_1, b_2, v_3)$, and $f(b_1, b_2, b_3)$.

The global maximum M of a directionally monotonic increasing function $f : \mathbb{R}^3 \to \mathbb{R}$ over a Bézier patch $\boldsymbol{P}$ can then be efficiently computed to recursion depth d by calling the function `FunctionMax`$(f, \boldsymbol{P}, f(\boldsymbol{P}_{00}), d)$ given in Algorithm 6. First, one replaces M by the maximum of f evaluated at the four corner nodes and the previous value of M. One then computes the vertex $\boldsymbol{b}$ of the bounding box$(\boldsymbol{a}, \boldsymbol{b})$ for the convex hull $\mathcal{C}$ of $\{\boldsymbol{P}_{ij}\}$. If the maximum $f(\boldsymbol{b})$ of the function on box$(\boldsymbol{a}, \boldsymbol{b})$ is less than or equal to M, the subpatch can be discarded. Otherwise, the patch is subdivided along the $u = v = 1/2$ isolines and the process is repeated using the new value of M.

Input: real function $\mathsf{f(triple)}$, patch $\boldsymbol{P}$, real M, integer depth
Output: real M
$\mathsf{M} \leftarrow \max(\mathsf{M}, \mathsf{f}(\boldsymbol{P}_{00}), \mathsf{f}(\boldsymbol{P}_{03}), \mathsf{f}(\boldsymbol{P}_{30}), \mathsf{f}(\boldsymbol{P}_{33}))$;
if $\mathsf{depth} = 0$ **then**
  return M;
end
$\mathsf{x} \leftarrow \max(\hat{\boldsymbol{x}} \cdot \boldsymbol{P}_{ij} : 0 \leq i, j \leq 3)$;
$\mathsf{y} \leftarrow \max(\hat{\boldsymbol{y}} \cdot \boldsymbol{P}_{ij} : 0 \leq i, j \leq 3)$;
$\mathsf{z} \leftarrow \max(\hat{\boldsymbol{z}} \cdot \boldsymbol{P}_{ij} : 0 \leq i, j \leq 3)$;
if $f((\mathsf{x}, \mathsf{y}, \mathsf{z})) \leq \mathsf{M}$ **then**
  return M;
end
foreach *subpatch* S *of* $\boldsymbol{P}$ **do**
  $\mathsf{M} \leftarrow \max(\mathsf{M}, \texttt{FunctionMax}(\mathsf{f}, \mathsf{S}, \mathsf{M}, \mathsf{depth} - 1))$;
end
return M;

Algorithm 6: `FunctionMax`(f,$\boldsymbol{P}$,M,depth) returns the maximum of M and the global bound of a real-valued directionally monotonic increasing function f over a Bézier patch $\boldsymbol{P}$ evaluated to recursion level depth. Here $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$, $\hat{\boldsymbol{z}}$ are the Cartesian unit vectors.

6. 3D Vector Typography

Donald Knuth's TEX system [12], the de-facto standard for typesetting mathematics, uses Bézier curves to represent 2D characters. TEX provides a portable interface that yields consistent, publication-quality typesetting of equations, using subtle spacing rules derived from centuries of professional mathematical typographical experience. However, while it is often desirable to illustrate abstract mathematical concepts in TeX documents, no compatible descriptive standard for technical mathematical drawing has yet emerged.

The recently developed Asymptote language[2] aims to fill this gap by providing a portable TeX-aware tool for producing 2D and 3D vector graphics [4]. In mathematical applications, it is important to typeset labels and equations with TeX for overall consistency between the text and graphical elements of a document. In addition to providing access to the TeX typesetting system in a 3D context, Asymptote also fills in a gap for nonmathematical applications. While open source 3D bit-mapped text fonts are widely available,[3] resources currently available for scalable (vector) fonts appear to be quite limited in three dimensions.

Asymptote was inspired by John Hobby's METAPOST (a modified version of METAFONT, the program that Knuth wrote to generate the TeX fonts), but is more powerful, has a cleaner syntax, and uses IEEE floating point numerics. An important feature of Asymptote is its use of the simplex linear programming method to solve overall size constraint inequalities between fixed-sized objects (labels, dots, and arrowheads) and scalable objects (curves and surfaces). This means that the user does not have to scale manually the various components of a figure by trial-and-error. The 3D versions of Asymptote's deferred drawing routines rely on the efficient algorithms for computing the bounding box of a Bézier surface, along with the bounding box of its 2D projection, described in Sec. 5. Asymptote natively generates PostScript, PDF, SVG, and PRC [2] vector graphics output. The latter is a highly compressed 3D format that is typically embedded within a PDF file and viewed with the widely available Adobe Reader software.

The biggest obstacle that was encountered in generalizing Asymptote to produce 3D interactive output was the fact that TeX is fundamentally a 2D program. In this work, we have developed a technique for embedding 2D vector descriptions, like TeX fonts, as 3D surfaces (2D vector graphics representations of TeX output can be extracted with a technique like that described in Ref. [5]). While the general problem of filling an arbitrary 3D

[2]available from `http://asymptote.sourceforge.net` under the GNU Lesser General Public License.

[3]For example, see `http://www.opengl.org/resources/features/fontsurvey/`.

closed curve is ill-posed, there is no ambiguity in the important special case of filling a planar curve with a planar surface.

Together with the 3D generalization of the METAFONT curve operators described by [3, 4], these algorithms provide the 3D foundation for Asymptote. Since our procedure transforms text into Bézier patches, which are the surface primitives used in Asymptote, all of the existing 3D Asymptote algorithms can be used without modification.

6.1. 3D Arrowheads

Arrows are frequently used in illustrations to draw attention to important features. We designed curved 3D arrowheads that can be viewed from a wide range of angles. For example, the default 3D arrowhead was formed by bending the control points of a cone around the tip of a Bézier curve. Planar arrowheads derived from 2D arrowhead styles are also implemented; they are oriented by default on a plane perpendicular to the initial viewing direction. Examples of these arrows are displayed in Figures 12 and 13. The `bezulate` algorithm was used to construct the upper and lower faces of the filled (red) planar arrowhead in Fig. 13.

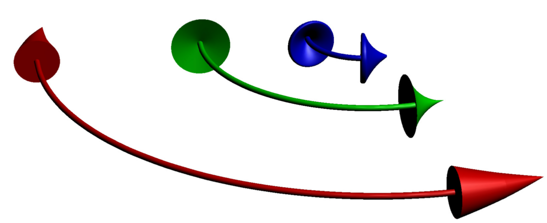

Figure 12: Three-dimensional revolved arrowheads in Asymptote.

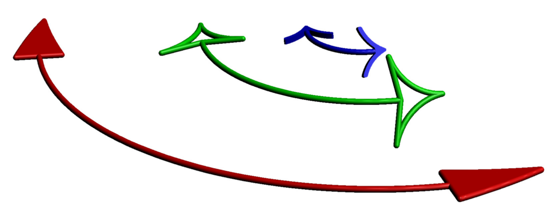

Figure 13: Planar arrowheads in Asymptote.

6.2. *Double Deferred Drawing*

Journal size constraints typically dictate the final width and height, in PostScript coordinates, of a 2D or projected 3D figure. However, it is often convenient for users to work in more physically meaningful coordinates. This requires *deferred drawing:* a graphical object cannot be drawn until the actual scaling of the user coordinates (in terms of PostScript coordinates) is known [4]. One therefore needs to queue a function that can draw the scaled object later, when this scaling is known. ASYMPTOTE's high-order functions provide a flexible mechanism that allows the user to specify either or both of the 3D model dimensions and the final projected 2D size. This requires two levels of deferred drawing, one that first sizes the 3D model and one that scales the resulting picture to fit the requested 2D size [5]. The 3D bounding box of a Bézier surface, along with the bounding box of its 2D projection, can be efficiently computed with the method described in Section 5.

6.3. *Efficient Rendering*

Efficient algorithms for determining the bounding box of a Bézier patch also have an important application in rendering. Knowing the bounding box of a Bézier patch allows one to determine, at a high level, whether it is in the field of view: offscreen Bézier patches can be dropped before mesh generation occurs [10]. This is particularly important for a spatially adaptive algorithm as used in ASYMPTOTE's OPENGL-based renderer, which resolves the patch to one pixel precision at all zoom levels. Moreover, to avoid subdivision cracks, renderers typically resolve visible surfaces to a uniform resolution. It is therefore important that offscreen patches do not force an overly fine mesh within the viewport. As a result of these optimizations, the native ASYMPTOTE adaptive renderer is typically comparable in speed with the fixed-mesh PRC renderer in ADOBE READER, even though the former yields higher quality, true vector graphics output.

7. Conclusions

In this work we have developed methods that can be used to lift smooth fonts, such as those produced by TeX, into 3D. Treating 3D fonts as surfaces allows for arbitrary 3D text manipulation, as illustrated in Figures 6 and 11. The `bezulate` algorithm allows one to construct planar Bézier surface patches by decomposing (possibly nonsimply connected) regions bounded by simple closed curves into subregions bounded by closed curves with four or fewer segments. The method relies on an optimized subdivision algorithm for testing whether a point lies inside a closed Bézier curve, based on the topological deformation of the curve to a polygon. We have also shown how degenerate Coons patches can be efficiently detected and split into nondegenerate subpatches. This is required to avoid both patch overlap at the boundaries of the underlying curve and rendering artifacts (patchiness, smudges, or wrinkles) due to normal reversal.

We have illustrated applications of these techniques in the open source vector graphics programming language ASYMPTOTE, which we believe is the first software to lift TEX into 3D. This represents an important milestone for publication-quality scientific graphing.

Acknowledgments

We thank Philippe Ivaldi, Radoslav Marinov, Malcolm Roberts, Jens Schwaiger, and Olivier Guibé for discussions and assistance in implementing the algorithms described in this work. Special thanks goes to Andy Hammerlindl, who designed and implemented much of the underlying ASYMPTOTE language, and to Michail Vidiassov, who helped write the PRC driver. Financial support for this work was generously provided by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Adobe, July 2008. Document management—Portable Document Format—Part 1: PDF 1.7. ISO 32000-1:2008.

[2] Adobe, 2008. PRC format specification. http://livedocs.adobe.com/acrobat_sdk/9/Acrobat9_HTMLHelp/API_References/PRCReference/PRC_Format_Specification/.

[3] Bowman, J. C., 2007. The 3D Asymptote generalization of MetaPost Bézier interpolation. Proceedings in Applied Mathematics and Mechanics 7 (1), 2010021–2010022.

[4] Bowman, J. C., Hammerlindl, A., 2008. Asymptote: A vector graphics language. TUGboat: The Communications of the TEX Users Group 29 (2), 288–294.

[5] Bowman, J. C., Shardt, O., 2009. Asymptote: Lifting TEX to three dimensions. TUGboat: The Communications of the TEX Users Group 30 (1), 58–63.

[6] Coons, S., 1964. Surfaces for computer aided design. Tech. rep., MIT, Cambridge, Massachusetts, available as AD 663504 from the National Technical Information service, Springfield, VA 22161.

[7] Farin, G., 1992. From conics to NURBS: A tutorial and survey. IEEE Computer Graphics and Applications 12 (5), 78–86.

[8] Farin, G., 2002. Curves and surfaces for CAGD: a practical guide. Morgan Kaufmann, San Francisco, CA.

[9] Haines, E., 1994. Point in polygon strategies. Academic Press Professional, Inc., San Diego, CA, USA, pp. 24–46.

[10] Hasselgren, J., Munkberg, J., Akenine-Möller, T., 2009. Automatic pre-tessellation culling. ACM Trans. Graph. 28 (2), 1–10.

[11] Hui, K., Wu, Y., 2005. Feature-based decomposition of trimmed surface. Computer-Aided Design 37 (8), 859–867.

[12] Knuth, D. E., 1986. The TEXbook. Addison-Wesley, Reading, Massachusetts.

[13] Lin, H., Tang, K., Joneja, A., Bao, H., 2007. Generating strictly non-self-overlapping structured quadrilateral grids. Comput. Aided Des. 39 (9), 709–718.

[14] Loop, C., Blinn, J., 2005. Resolution independent curve rendering using programmable graphics hardware. In: SIGGRAPH '05: ACM SIGGRAPH 2005 Papers. ACM, New York, NY, USA, pp. 1000–1009.

[15] Neumark, S., 1965. Solution of cubic and quartic equations. Pergamon Press, Oxford.

[16] Nishita, T., Sederberg, T. W., Kakimoto, M., 1990. Ray tracing trimmed rational surface patches. SIGGRAPH Comput. Graph. 24 (4), 337–345.

[17] Ostby, E., 1988. Defining a sphere with Bezier patches, `http://www.gamedev.net/reference/articles/article407.asp`.

[18] Piegl, L., Tiller, W., 1997. The NURBS book. Springer, Berlin.

[19] Randrianarivony, M., Brunnett, G., May 2004. Necessary and sufficient conditions for the regularity of planar Coons map. Tech. Rep. SFB393/04-07, Technische Universität Chemnitz, Chemnitz, Germany.

[20] Shewchuk, J. R., Oct. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. Discrete & Computational Geometry 18 (3), 305–363.

[21] Wang, C., Tang, K., 2004. Algebraic grid generation on trimmed parametric surface using non-self-overlapping planar Coons patch. International Journal for Numerical Methods in Engineering 60 (7), 1259–1286.

[22] Wang, C. C. L., Tang, K., 2005. Non-self-overlapping Hermite interpolation mapping: a practical solution for structured quadrilateral meshing. Computer-Aided Design 37 (2), 271–283.