Interactive Display of Large Scale Trimmed NURBS Models*

Subodh Kumar
Department of Computer Science
University of North Carolina
Chapel Hill NC 27599

Dinesh Manocha
Department of Computer Science
University of North Carolina
Chapel Hill NC 27599

Abstract: We present an algorithm for interactive display of trimmed NURBS surfaces. The algorithm converts the NURBS surfaces to Bézier surfaces, tessellates each trimmed Bézier surface into triangles and renders them using the triangle rendering capabilities common in current graphics systems. It makes use of tight bounds for uniform tessellation of Bézier surfaces into cells and traces the trimming curves to compute the trimmed regions of each cell. This is based on trim curve tracing, intersection computation with the cells, and triangulation of the cells. The resulting technique also makes efficient use of spatial and temporal coherence between successive frames for cell computation and triangulation. Polygonization anomalies like cracks and angularities are avoided as well. The algorithm works well in practice and, on the high end graphics systems, is able to display trimmed models described using thousands of Bézier surfaces at interactive frame rates.

1 Introduction

Current graphics systems have reached the capability of rendering millions of transformed, shaded and z-buffered triangles per second [Ake93, Fea89]. However in many applications involving CAD/CAM, virtual reality, animation and visualization the object models are described in terms of non-uniform

*Supported by DARPA ISTO Order A410, NSF Grant MIP-9306208, Junior Faculty Award, University Research Award, NSF Grant CCR-9319957, ARPA Contract DABT63-93-C-0048 and NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract 8920219 and ONR Contract N00014-94-1-0738

rational B-spline (NURBS) surfaces. This class includes Bézier surfaces and other rational parametric surfaces like tensor product and triangular patches. Large scale models consisting of thousands of such surfaces are commonly used to represent shapes of automobiles, submarines, airplanes, building architectures, sculptured models, mechanical parts and in applications involving surface fitting over scattered data or surface reconstruction. Current renderers of sculptured models on commercial graphics systems, while faster than ever before, are not able to render them in real time for applications involving virtual worlds and other immersive technologies.

Main Result: We present an algorithm for interactive display of large scale NURBS models on current graphics systems. Given NURBS surface representations, the algorithm decomposes them into a series of Bézier surfaces and computes *tight bounds* for on-line tessellation. For trimmed models the algorithm computes intersections between the trimming curves and domain tessellation to determine the visible portions of each surface. We perform *back-patch culling* to determine visible surfaces on solid models and make use of *temporal and spatial coherence* between adjacent frames. The resulting algorithm is portable and its actual performance is a function of the hardware resources available on a system (memory, CPUs, and special purpose chips). Our current implementation on the SGI-VGX can display about *four hundred surfaces* and on Pixel-planes 5, [Fea89] about *six thousand surfaces* at interactive frame rates (about $12 - 15$ frames a second).

Previous Work: Curved surface rendering has seen active research in the last two decades and boasts of rich literature. The main techniques are based on pixel level surface subdivision, ray tracing, scan-line display and polygonization [Cat74, Cla79, LR81, For79, Kaj82, NSK90, LCWB80]. Techniques based on ray tracing, scan-line display and pixel level display do not make efficient use of the hardware capabilities available on current architectures. As a result, algorithms based on polygonization are, in general, faster. Different methods have been proposed for polygonization [Baj90, AES93, Bea91, Dea89, LR81, Luk93, LC93, Che93, SC88, SL87, FK90, RHD89, Roc87, AES91, FMM86]. These are based on adaptive or uniform subdivision of NURBS surfaces. [RHD89] have proposed a real time algorithm for trimmed surfaces. However the bounds used for tessellating the Bézier surfaces are loose for rational surfaces and in some cases even undersample the surface. Furthermore, the operations used on trimming curves are relatively expensive and affects the performance of the overall algorithm. Some techniques to improve the tessellation and their computations are presented in [FMM86, AES91, AES93]. The algorithm presented in this paper has considerable improvements over these algorithms (e.g. see Figs. 1 and 2).

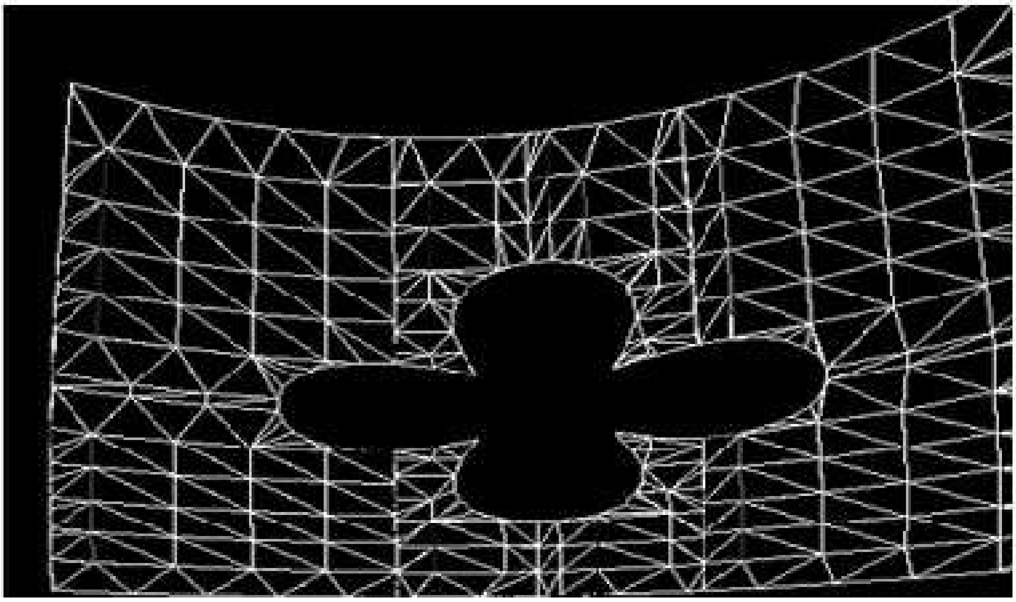


Figure 1: [RHD89]'s Trimming Algorithm

Organization: The rest of the paper is organized in the following manner. In Section 2 we analyze the problem of computing polygonal approximations to surface models and give an overview of our approach. In Section 3, we consider visibility processing and introduce *back-patch culling*. The algorithms for dynamic tessellation of Bézier surfaces based on tight bounds are presented in Section 4. Section 5 deals with handling of trimming curves and computation of visible portions of each surface model. The use of coherence is demonstrated in section 6. We discuss implementation in Section 7 and compare its performance with that of earlier algorithms. In this paper we have demonstrated these techniques on tensor-product surface models only. However, they are directly applicable to models composed of triangular patches as well.

2 Polygonal Approximation of Surfaces

Any surface rendering algorithm based on polygonization involves two phases of computation for each frame:

- *Polygon Generation Phase:* Approximate the surface by polygons. The number of polygons generated should result in a smooth image after Gouraud or Phong shading.

- *Polygon Rendering Phase:* Render the polygons through the graphics pipeline using transformation, smooth shading and z-buffering for hidden surface elimination.

Each of these contributes to the running time of the overall algorithm. The performance of polygon rendering algorithms is system dependent and typically is a function of the number of polygons and the size and distribution of these polygons on the screen. Our emphasis is on developing efficient algorithms for polygon generation (we are really interested in triangle generation, since triangles can be rendered significantly faster than general polygons on most architectures) and in the process we want to minimize:

1. The number of polygons generated.

2. The time spent in generating the polygons.

These two goals are conflicting. On one hand we can compute a very fine polygonization à priori and can render all the polygons at each frame. In this case, almost no time is spent in polygon generation and all the time is spent in rendering. However the number of polygons needed for close-up (zoomed) views of some surfaces can be extremely high (a few thousand) and for models consisting of thousands of surfaces, this requires hundreds of megabytes of storage, and the capability to render hundreds of millions of polygons per second. We can reduce the demand on polygon rendering capability by computing different *multiresolution approximations* of each surface and at each phase choosing one of the approximations as a function of the viewing parameters. But the memory requirements only get worse. On the other hand, we can compute on-line, the minimum number of polygons required for a smooth image as a function of the viewing parameters (for each frame). The resulting algorithm is based on adaptive subdivision and takes considerable time in the polygon generation for each frame. As a result, it is too slow for interactive performance on large scale models.

Overview: Any good algorithm has to balance the conflicting goals highlighted above. Our approach to interactive display of large scale models has the following facets.

1. *Visibility Processing :* Perform simple on-line computations to isolate patches not visible from the current viewpoint. This includes use of viewing frustum as well as a new technique, *back-patch culling.*

2. *Dynamic Tessellation :* Given the viewing parameters, we dynamically compute the tessellation appropriate for smooth shading. As a result, we need only a few megabytes of memory to

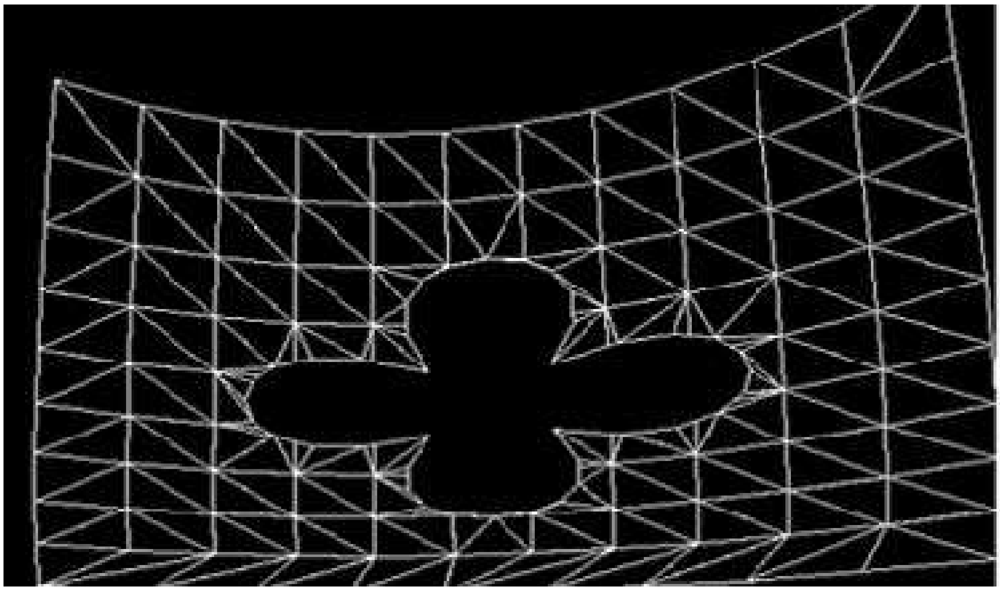


Figure 2: Our Trimming Algorithm

store the polygonization for large scale models. We use tight bounds to optimize the number of polygons generated.

3. *Trimmed Surfaces :* We use a simple algorithm to determine the regions of a patch visible due to trimming curves. This is based on intersection determination, coving and tiling and visible cell determination.

4. *Spatial and Temporal Coherence :* We make use of coherence between successive frames to minimize the overall computations for polygon generation. In particular, we perform incremental computations.

Background: Given a NURBS model, we use knot insertion to decompose them into a series of Bézier patches [Far90]. In the process, we insert the minimum number of knots as a function of the knot sequence of the original surface and its order. Closely spaced knots, with tolerance less than 2×10^{-5} are coerced to the same value before knot insertion. *Trimmed* NURBS surfaces are decomposed into a series of trimmed Bézier surfaces. This involves knot insertion algorithm as well as the breaking up of the trimming curves at the patch boundaries. Trimming curves are typically represented as piecewise linear curves or NURBS. Piecewise linear curves are split at the patch boundaries and new points inserted at the boundary. NURBS curves are converted into Bézier curves and split across the surface boundaries as well by computing their intersections with the

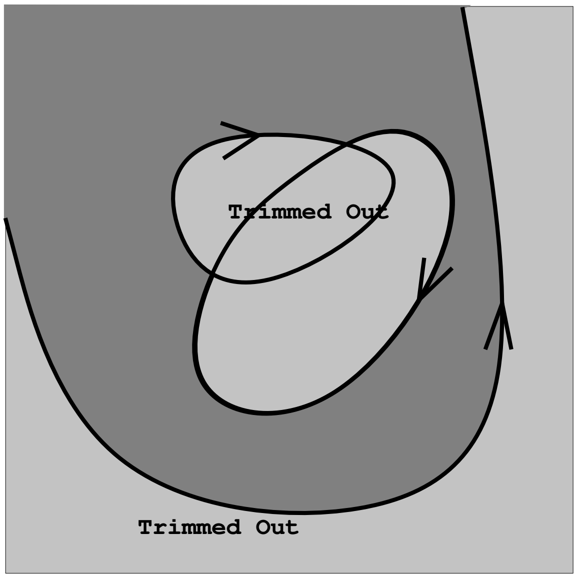


Figure 3: Trimming Rule

patch boundaries.

The 3D coordinate system in which the NURBS model is defined is referred to as the *object space*. Viewing transformations, like rotation, translation and perspective, map it onto a viewing plane known as the *image space*. Associated with this transformation are the viewpoint, viewing cone and clipping planes. Finally, *screen space* refers to the 2D coordinate system defined by projecting the image space onto the plane of the screen.

Trimming rule: A trimmed surface is represented by the set of its control points and a set of trimming curves (or trim curves). The *exterior* of a trimming curve is defined as the right side of the curve when traced starting at $\mathbf{p}_0$. (In that sense the trim curves are oriented curves.) A trim curve *trims out* its exterior. This is called the *handedness rule* of trimming. It does not allows self intersecting trim curves but two different trim curves can intersect. Sometimes a *winding rule* is used to defined the exterior of a curve. According to this rule, the region of a surface that is enclosed by an even number of loops is trimmed out. This means that trimming curves must explicitly be loops. Very often the source of these trimming curves are intersection of surfaces, and the semantics of these curves are exactly akin to the handedness rule. For example Fig. 3 shows a trimmed region that could require redefinition of curves if we used the odd-winding rule.

3 Visibility Computations

Given a large model consisting of Bézier patches, not all patches are typically visible from a viewpoint. A good part of the model may be clipped by the viewing volume. The rest of the model is tessellated and the triangles are sent down the rendering pipeline. On the other hand if we a priori determine that a Bézier patch is invisible from a given viewpoint, we don't need to even generate the triangles for that patch. In general, the exact computation of the visible portions of a NURBS model is a non-trivial problem requiring silhouette computation. We show that it is relatively simple to perform a visibility check to find most of the patches that are completely non-visible. Combined with z-buffering, this can calculate exact visibility very fast.

A Bézier surface, defined in homogeneous coordinates as $\mathbf{F}(u, v) = (X(u, v), Y(u, v), Z(u, v), W(u, v))$, is specified by a mesh of control points. Furthermore, the entire surface is contained in the convex polytope of the control points [Far90]. Let us denote this convex polytope as P_F. We also compute an axis-aligned bounding box, B_F, defined by eight vertices as the smallest volume bounding box enclosing P_F.

3.1 Patch Clipping

The first phase of visibility processing involves checking whether a patch lies in the viewing volume at all. As a gross approximation, we transform the eight corners of B_F to screen space and check if any part of it lies inside the viewing volume. We do not transform all the points of B_F or P_F using a 4×4 matrix multiplication corresponding to the viewing transformation. Corresponding to each frame, we transform the viewing pyramid in an inverse manner. As opposed to computing the inverse of the viewing matrix, we decompose the translation and rotations in the object space and apply them to the eye point in the opposite direction. If the transformed viewing pyramid encompasses the control polytope we *cull* the patch. This is a well known technique for doing simple visibility.

3.2 Back-patch Culling

Large scale models generally consist of a large number of small patches. Given a solid model, whose boundary is composed of Bézier patches, many of them are occluded from the view. Culling out back facing polygons is commonly used to improve rendering performance. Similarly for a Bézier patch, if all the surface normals point away from the eye point we refer it as a *back patch* (Fig. 4).

The normal vectors on the surface are defined as $\mathbf{N}(u,v) = F_u \times F_v$. (We assume that all normals point "outwards" from the model.)

The exact computation of back-patches involves computation of the *Gauss maps* of the surface. They correspond to a mapping of the normals onto the unit-sphere. However, the exact computation is relatively expensive and our algorithm represents the pseudo-normal surface, $\mathbf{N}(u,v)$, as a Bézier surface. This is obtained by taking the cross product of the derivative vectors (Fig. 5) and performing degree elevation to compute the control points of the pseudo-normal surface. The convex hull of the control points of $\mathbf{N}(u,v)$, say P_N, is used to obtain a bounding polytope of $\mathbf{N}(u,v)$. Furthermore, we compute a minimum volume eight vertex axis-aligned box B_N bounding it. Each point on $\mathbf{N}(u,v)$ corresponds to a direction on $\mathbf{F}(u,v)$ and P_N and B_N define multiple sided polytopes in which all these directions lie. Corresponding to the viewing direction vector, say $\mathbf{V}$, we define a half-plane $\mathbf{H}$ perpendicular to $\mathbf{V}$ such that it partitions the unit sphere into two hemispheres ($S1$ and $S2$, see Fig. 5). If the Gauss map of the surface lies entirely in $S1$, it is not visible to the eye. Given $\mathbf{V}$, we compute $\mathbf{H}$ and check whether all the vertices of $\mathbf{P}_N$ lie in the halfspace containing $S1$.

If $\mathbf{F}(u,v)$ has a polynomial representation, the pseudo-normal surface consists of $2m \times 2n$ control points. For rational patches, the degree bound obtained after taking the cross-products is $(4m-1)$ in u and $(4n-1)$ in v. However it can be improved in the following way. Let $\mathbf{f}(u,v) = (X(u,v), Y(u,v), Z(u,v)$.

$$\mathbf{F}_u = \frac{\mathbf{f}_u W - \mathbf{f} W_u}{W^2}, \quad \mathbf{F}_v = \frac{\mathbf{f}_v W - \mathbf{f} W_v}{W^2}.$$

Therefore, the pseudo-normal surface can be written as:

$$\mathbf{N} = (\mathbf{f}_u W - \mathbf{f} W_u) \times (\mathbf{f}_v W - \mathbf{f} W_v).$$

After expanding this expression, simplifying, and dividing by W we get

$$\mathbf{N} = \mathbf{f}_u \times \mathbf{f}_v W - \mathbf{f}_u \times \mathbf{f} W_v - \mathbf{f} W_u \times \mathbf{f}_v.$$

Thus, the pseudo normal surface can be represented by $3m \times 3n$ control mesh. Testing for visibility reduces to checking whether each of these control points, or just the bounding box $\mathbf{B}_N$, is in the appropriate half-space.

As the objects are transformed using translation, rotation or zoom, we do not transform each control point of Bézier surface and recompute $\mathbf{N}(u,v)$. Rather, we transform the viewing volume and the corresponding viewing vector $\mathbf{V}$ and half-plane $\mathbf{H}$. As a result, at each frame we are only

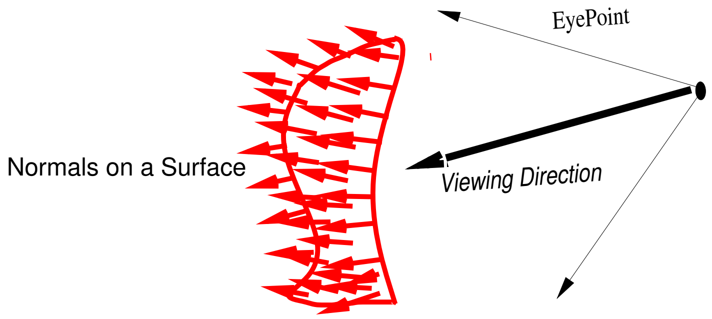


Figure 4: Patch Visibility

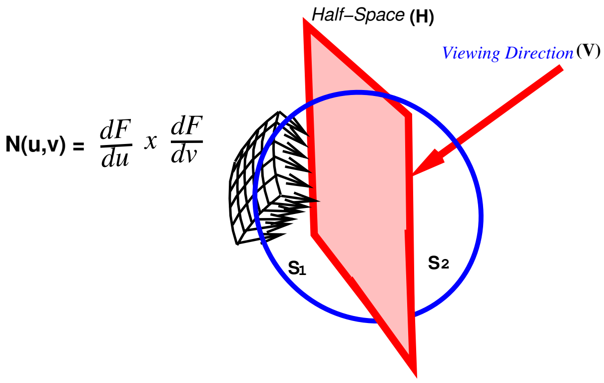


Figure 5: Visibility Computation

testing whether the control points or the bounding box of the pseudo-normal surface lie in the transformed half-space.

For most solid models, back-patch culling eliminates about 30-40% of the model. Since most patches are relatively flat, we have found that using bounding boxes B_N are good enough. For a rational bicubic patch the control polytope of $\mathbf{N}(u, v)$ consists of 81 control points, whereas the bounding box has only eight points. Each patch is individually tested at each frame for visibility. Using back-patch culling the performance of the overall algorithm improves by $20-35\%$, depending on the model. For the Pencil model it improves by about 23% and for the Dragon model (Fig. 19) the speedup is about 33%.

4 Polygonization

We dynamically compute the polygonization of the surfaces as a function of the viewing parameters. Polygonization can be computed using uniform or adaptive subdivision for each frame. Uniform tessellation involves choosing constant step sizes along each parameter. Adaptive tessellation uses a recursive approach to subdivision based on "flatness criteria" and surface areas. For large scale models, we found that uniform subdivision methods are much faster in practice for the following reasons:

1. Simplicity of the algorithm. Uniform tessellation involves a precomputation of bounds and evaluation only.

2. Simple algorithms based on uniform forward differencing and modified Horner's rule of average complexity $O(n)$ as opposed to $O(n^2)$ based on de Casteljau's algorithm (for a curve of degree n).

3. No good and simple algorithms for quick determination of the flatness of parts of a patch.

4. Ability to easily combine uniform tessellation with spatial and temporal *coherence*.

5. Simplicity of handling trimmed curves intersections, coving and tiling and visibility determination.

Most of the large scale models consist of relatively flat surfaces. This is indeed the case after converting B-spline models into Bézier surfaces. Adaptive subdivision does well on surfaces with highly varying curvatures and large areas. On such models the uniform tessellation may supersample the surface. The performance of uniform tessellation algorithms is a direct function of the step sizes.

4.1 Uniform Subdivision

There is considerable literature on computation of bounds on polynomials [LR81, FMM86, Roc87, AES91]. There are two main criteria for computing bounds for step sizes: *size criterion* and *deviation criterion*. The size criterion determines a bound on the size of the resulting triangles in screen space and the deviation criterion computes a bound on the deviation of these triangles from the curved surface. Further, the size and deviation criteria are functions of the first and second order derivatives, respectively, of the surface vector. The size criterion works well only if the size parameter is small and the surface does not have small area and high curvature. In the latter case, these bounds undersample the surface (as shown in Fig. 6a). The deviation criterion generates good approximation but is computationally expensive. In particular for rational surfaces, the degree of the second order derivative vector goes up by a factor of four and therefore, any kind of computation for the deviation criterion takes a large fraction of the overall time. These bounds can be applied in two ways for step size computation:

1. Compute the bounds on the surface in the object space as a preprocessing step. The step size is computed as a function of these bounds and viewing parameters [LR81, FMM86, AES91].

2. Transform the surface into screen space based on the transformation matrix. Use the transformed representation to compute the bounds and , the step size as a function of these bounds [Roc87, RHD89].

We start with the size criterion for bound computations. To avoid undersampling highly curved surfaces with low areas, we use an additional estimate based on the geometry of the control points.

A rational Bézier surface is given as $\mathbf{F}(u,v) = (X(u,v), Y(u,v), Z(u,v), W(u,v)) =$

$$(\Sigma_{i=0}^{m}\Sigma_{j=0}^{n} w_{ij}\mathbf{r}_{ij} B_i^m(u) B_j^n(v), \Sigma_{i=0}^{m}\Sigma_{j=0}^{n} w_{ij} B_i^n(u) B_j^m(v)), \tag{1}$$

where $\mathbf{r}_{ij}$ are the control points in the object space, w_{ij} are the weights and B_i^m, B_j^n are the Bernstein polynomials. After applying all the viewing transformations (rotations, translation, perspective), let the new control points in the screen space be: $\mathbf{R}_{ij} = (X_{ij}, Y_{ij}, Z_{ij})$ and let W_{ij} be the corresponding new weights. Let TOL be the user specified tolerance in screen space. The step sizes along the u and v directions are given as [Roc87]:

$$n_u = m \ \max \frac{\parallel W_{ij}\mathbf{R}_{ij} - W_{i+1,j}\mathbf{R}_{i+1,j} \parallel}{TOL * min(W_{ij})}$$

$$n_v = n \ \max \frac{\parallel W_{ij}\mathbf{R}_{ij} - W_{i,j+1}\mathbf{R}_{i,j+1} \parallel}{TOL * min(W_{ij})}$$

for $(1 \leq i \leq m, \ 1 \leq j \leq n)$.

In practice these bounds are good for polynomial surfaces only, when $W_{ij} = 1$. However after perspective transformation, all the polynomial parametrizations are transformed into rational formulations. Furthermore, since the weights tend to vary considerably (typically by an order of three to four), these bounds *oversample* the curved surface for a given TOL. As a result, the polygon rendering becomes a bottleneck for the overall algorithm.

4.2 Bound Computation

We compute improved bounds for the rational surfaces in the object space as part of a preprocessing phase. They are used to compute the step sizes as a function of the viewing parameters as shown in [FMM86, AES93]. An algorithm for computation of bounds based on the size criterion has been highlighted in [AES91]. However, the derivation of bounds in [AES91] is inaccurate and for a given TOL, our bounds are tighter. We illustrate the derivation on a Bézier curve (it is applied in a similar manner to the surfaces). Given a rational curve $\mathbf{C}(t) = (x(t), y(t), z(t), w(t)$. Let $X(t) = \frac{x(t)}{w(t)}, \ldots, Z(t) = \frac{z(t)}{w(t)}$. Given a step size δ in the domain, we want to come up with tight bounds on the length of the vector $\mathbf{C}(t+\delta) - \mathbf{C}(t)$. It follows from the Mean Value Theorem:

$$(\mathbf{C}(t+\delta) - \mathbf{C}(t)) = \delta\ (X^{'}(t_1), Y^{'}(t_2), Z^{'}(t_3)),$$

where $t_1, t_2, t_3 \in [t, t+\delta]$. t_1, t_2 and t_3 need not be equal. As a result

$$\| \mathbf{C}(t+\delta) - \mathbf{C}(t) \| = \delta \parallel X^{'}(t_1), Y^{'}(t_2), Z^{'}(t_3) \|$$

$$\leq \delta\ \ \| \overline{X}^{'}(t), \overline{Y}^{'}(t), \overline{Z}^{'}(t) \|,$$

where $\overline{X}^{'}(t), \overline{Y}^{'}(t), \overline{Z}^{'}(t)$ represent the maximum magnitude of $X'(t), Y'(t), Z'(t)$, respectively, in the domain $[0,1]$ and $\|V\|$ is the L_2 norm of the vector V. Given these maximum values of the derivatives and TOL, we choose the step size δ satisfying the relation

$$\delta \leq \frac{TOL}{\| \overline{X}^{'}(t), \overline{Y}^{'}(t), \overline{Z}^{'}(t) \|}.$$

Thus for the Bézier surface, $\mathbf{F}(u,v)$, the tessellation parameters are computed in the object space as:

$$n_u = \frac{\left\| \overline{\frac{X(u,v)}{W(u,v)}}_u, \overline{\frac{Y(u,v)}{W(u,v)}}_u, \overline{\frac{Z(u,v)}{W(u,v)}}_u \right\|}{TOL},$$

where $\overline{\frac{X(u,v)}{W(u,v)}}_u$ corresponds to the maximum magnitude of the partial derivative of $\frac{X(u,v)}{W(u,v)}$ with respect to u in the domain $[0,1] \times [0,1]$. n_v is computed analogously.

The maximum values of the partial derivates are computed in the following way. Let

$$fx(u,v) = \left(\frac{X(u,v)}{W(u,v)}\right)_u = \frac{(X_u(u,v)W(u,v) - X(u,v)W_u(u,v))}{W(u,v)^2}.$$

$fy(u,v)$ and $fz(u,v)$ are defined in a similar manner. The maximum of $fx(u,v)$ in the input domain corresponds to one of the roots of $fx_u(u,v) = 0$ and $fx_v(u,v) = 0$ or occurs at the boundary of the domain. The maximum at the boundary correspond to one of the roots of $fx(0,v) = 0$, $fx(1,v) = 0$,

$fx(u,0) = 0$ or $fx(u,1) = 0$ or occurs at $fx(0,0)$, $fx(0,1), fx(1,0)$ or $fx(1,1)$. Thus the problem of computing the maximum derivative vector reduces to computing zeros of polynomial equations. In fact, it geometrically corresponds to curve intersection [MD92, Sed89]. In the first case, the two curves are algebraic plane curves, given as:

$$X_{uu}W^2 - XWW_{uu} - 2W_uX_uW + 2XW_u^2 = 0,$$

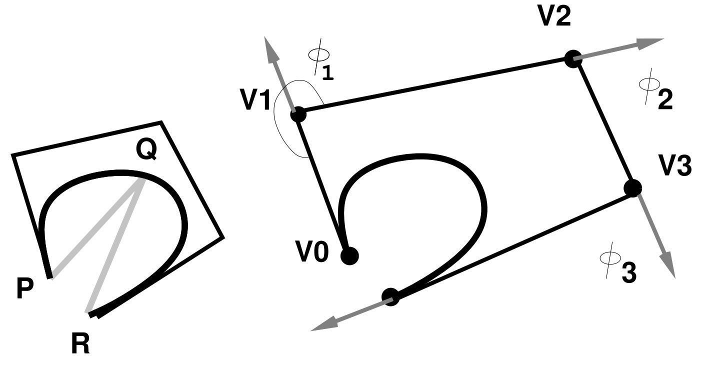
$$X_{vv}W^2 - XWW_{vv} - 2W_vX_vW + 2XW_v^2 = 0.$$

The degrees of these curves are $(3m-2, 3n)$ in (u,v) for the first curve and $(3m, 3n-2)$ in (u,v) for the second curve. This is rather high. However, we are able to compute accurate solutions in double precision arithmetic using the algorithm described in [MD92]. In particular, it reduces the problem to computing eigenvalues of a matrix. Good implementations of eigenvalue evaluators are available as part of numerical libraries like EISPACK and LAPACK. The resulting algorithms are fast, accurate and need no initial guess to the solutions. It takes about a second on the SGI-VGX to find the solutions for a rational bicubic surface. Note that all these computations are part of the preprocessing stage. Similarly, the maximum of $fx(0,v)$ corresponds to computing the roots of $fx_v(0,v) = 0$, which can be computed using root-finders or subdivision properties of Bézier curves [LR81]. Based on the solutions of these equations, we compute the maximum values of $fx(u,v)$ in the domain $[0,1] \times [0,1]$. Let the maximum value be at $[u_x, v_x]$. Similar computations are performed on $fy(u,v)$ and $fz(u,v)$. In case the domain parameters $([u_x, v_x], [u_y, v_y], [u_z, v_z])$, for the maximum of these three functions differ significantly, we subdivide the surface patch and compute the maximum in the subdivided domains using the roots of the equations shown above. Each of the subdivided surfaces are handled separately. The complexity of the bounds computation reduces significantly for polynomial surfaces as $W(u,v) = 1$ and the resulting equations have much lower degrees.

Given these bounds in the object space, we compute the step size in the screen space as a function of the viewing transformations. These bounds are invariant to rigid body transformations like rotations and translations. They vary with the perspective transformation matrix as shown in [AES93].

4.3 Curvature Bounds

For small values of TOL the size criterion bounds, derived above, work very well. In case the surface area is small and curvature is high, they may undersample the surface. For example see Fig 6a:



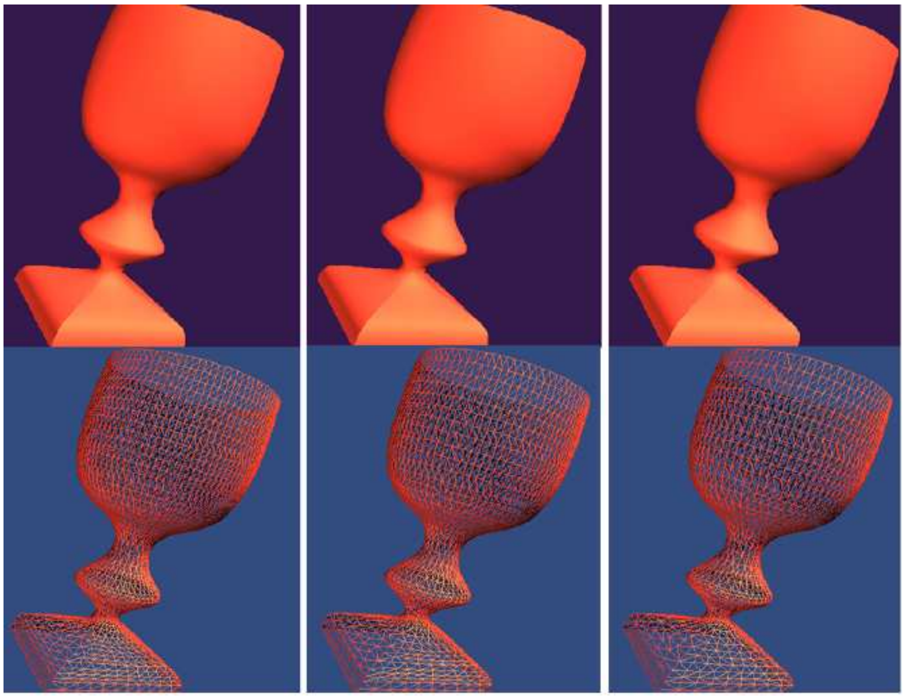
(a) Undersampling (b) Curvature Estimation

Figure 6:

the curve C is tessellated into two segments PQ and QR, each of magnitude less than TOL. The optimal solution to that problem would be based on the deviation criterion. However, in practice it typically oversamples the surface and its computation is expensive. We use a simple bound based on the geometry of the transformed control points in the screen space. Let $(V_0, V_2, \ldots, V_n)$ be the control points of a planar Bézier curve. The curve is defined in the screen space. The geometry of the curve is determined by the geometry of the control polygon. Let $\phi_i = \pi - Angle(V_{i-1}, V_i, V_{i+1})$, $1 \leq i \leq n-1$, be the angle in radians (Fig. 6b). Moreover let the area of the control polygon be A. We add a factor of $n_t^{'} = c(\Sigma_{i=1}^{n-1}\phi_i)/A$ to the bound parameter computed in the last section. c is a user-defined constant. Intuitively it works in the following manner: For a given curve, the tangent vector at $t = 0$ is in the direction of $V_1 - V_0$ and at $t = 1$ is in the direction of $V_n - v_{n-1}$. As a result, the term $\Sigma\phi_i$ reflects this variation in the derivative vectors over the control polytope. For curves with high curvature this value is relatively high. We only need to add this parameter to the size criterion if the relative size of the curve is small. As a result, division by the area parameter serves that purpose. Given the representation of the surface in (1), we consider the Bézier curves defined as $(\mathbf{R}_{i1}, \mathbf{R}_{i2}, \ldots, \mathbf{R}_{in})$ for all $1 \leq i \leq m$ and take the maximum of $n_{u_i}^{'}$ to add to n_u. $n_v^{'}$ is computed in a similar manner.

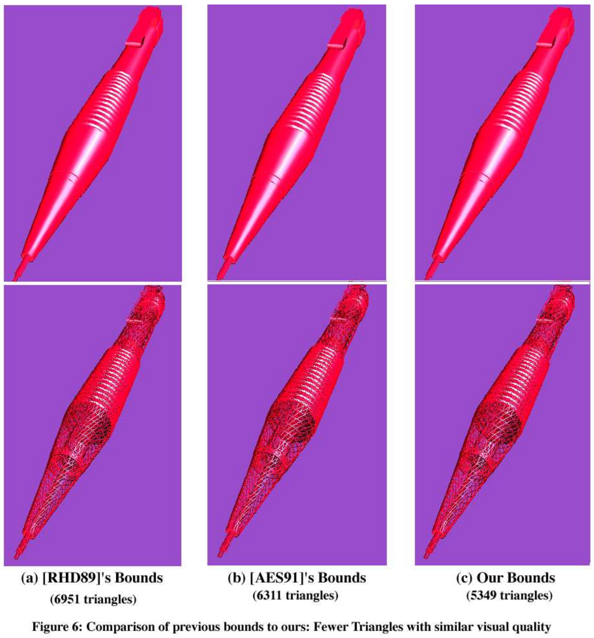
4.4 Comparison of Methods

We empirically compared our bound with those of Rockwood [Roc87] and Abi-Ezzi/Shirman [AES91]. These comparisons were performed over a number of models and we computed the averages of the



(a) [RHD89]'s Bounds (b) [AES91]'s Bounds (c) Our Bounds

I. Alpha 1 Goblet



(a) [RHD89]'s Bounds (b) [AES91]'s Bounds (c) Our Bounds

II. Alpha 1 Pencil

Figure 7: Comparison of previous bounds to ours: Fewer triangles with similar visual quality

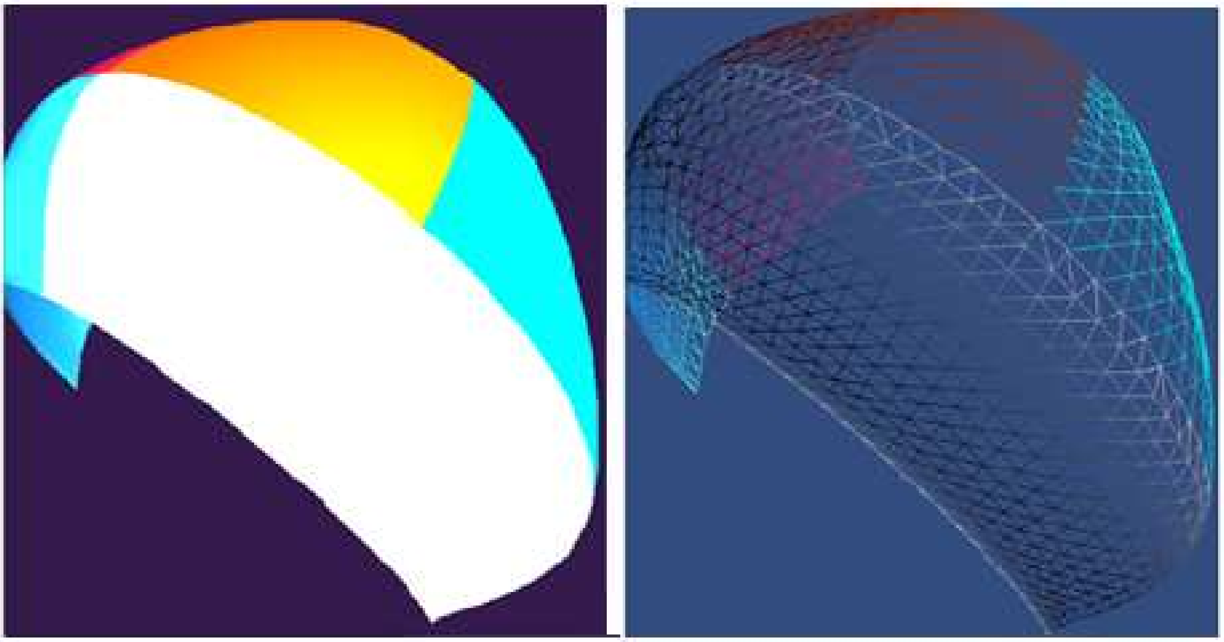


Figure 8: T-Joint

number of polygons generated. The average has been taken over seven models and the number of patches varied from 72 to 5354. The degrees of the models were between two and three in u as well as v. For the same tolerance, our bounds result in about 33% fewer triangles than [Roc87] and about 20% fewer than [AES91]. Fig. 7 compares the wireframes and shaded images of a car panel using the three methods.

Model	Our Algorithm	[Roc87]	[AES91]
Goblet	1	1.48	1.26
Pencil	1	1.51	1.20

Table 1: Relative comparison of the number of triangles generated for a given tolerance

4.5 Crack Prevention

Since the bound for required tessellation for each patch is evaluated independently, we could mandate different tessellations on two adjacent patches. This could result in cracks in the rendered image. To address this issue [FMM86, RHD89] suggested that the amount of tessellation at the boundary be based solely on the boundary curve, and a strip of coving triangles be generated at the boundary. But this method does not work if the common boundary curves of the two adjacent patches do not have exactly the same parametric representation in terms of the control points. One common example occurs when one of the patches is subdivided into two, resulting in a T-joint at the common boundary (Fig. 8). In such cases there is no way to prevent cracks without using any information about the adjacent patches. Suppose there was – say, we calculate the required tessellation for the boundary curve $u = 0$ of a patch F. We can always subdivide one of the patches adjacent at this boundary, say F', into two, say F'_1 and F'_2, and reparametrize each one of them in such a way that the tessellation points on the boundaries of F'_1 and F'_2 are different from F' and hence from F.

The algorithm computes the adjacency information between the patches in the preprocessing phase. We assume that the patches share the same boundary curves geometrically. The algorithm computes the bounding boxes of the boundary curves and sorts them along their projections on the X, Y and Z axis to compute the overlapping pairs. For each pair of overlapping boxes the algorithm checks whether the two curves are geometrically the same (and form a common boundary) as follows:.

Let us represent the two boundary curves as $C_1(t)$ and $C_2(t)$. They are Bézier curves defined using control points. Let $P_1 = C_1(0), P_2 = C_2(0)$. The curves are common iff either P_1 lies on $C_2(t)$ or P_2 lies on $C_1(t)$ for $0 \leq t \leq 1$. This query reduces to an *inversion* problem: given a point P and a curve C, find the parameter value t such that $C(t) = P$. We solve it using techniques from elimination theory [MD92].

For each patch boundary, we *associate* one of the adjacent patches (chosen arbitrarily) with it. If we have two different representations for the same curve, we store the representation of the associated patch's boundary curve with both the patches. To calculate the bounds on the curves and tessellate them we use this stored representation.

5 Tessellation

Given n_u and n_v, it is straightforward to construct the grid in the (u, v) domain. The grid points divide the domain into rectangles (we just need to draw a diagonal to get the desired triangles). At the patch boundaries we construct coving triangles. Let us call these rectangles and triangles, *cells*.

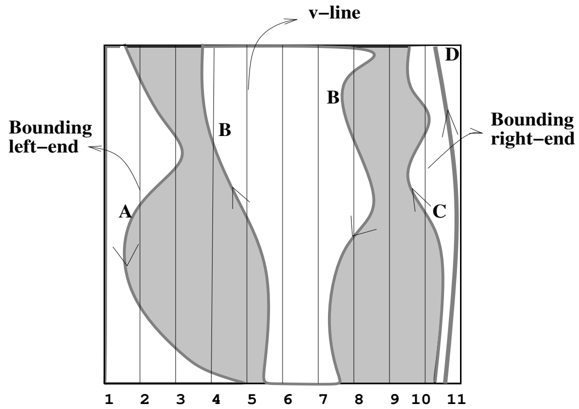


Figure 9: Active v-lines

It is over the canvas of these cells that we need to draw the trimming curves.

The idea is to do special processing only for the *partially trimmed cells*, the cells that have a region trimmed out. The points on the cells that are fully trimmed out need not be evaluated at all. Since no trim curve passes through the fully untrimmed cells, they can be triangulated the old way by drawing a diagonal. For partially trimmed cell we need to make sure that all triangles have vertices that are either grid points of the patch or the tessellation points (tessellants) on the trimming curves.

A grid line perpendicular to the v axis, the isocurve $v = K$, is called a *v-line* (Fig. 9), K being the *v-value* of the v-line. The side of a cell that lies on a v-line is called its *v-edge* and the side that lies on a u-line is called its *u-edge*. The region of the patch lying between two adjacent v-lines is called a *v-strip*. *u-line, u-strip* and *u-value* are defined similarly.

For simplicity, the algorithm is presented here for rectangular cells only (The extension for the triangular cells at the patch boundaries is straightforward.):

1. Find the mutual intersection of trim curves (preprocessing step).

2. Compute n_u and n_v for the patch and n_t for each trim curve.

3. Trace each trimming curve and find its intersection with u-lines and v-lines. Also prepare set of *active* v-line and cells. An *active cell* has at least one of its vertices on the untrimmed part

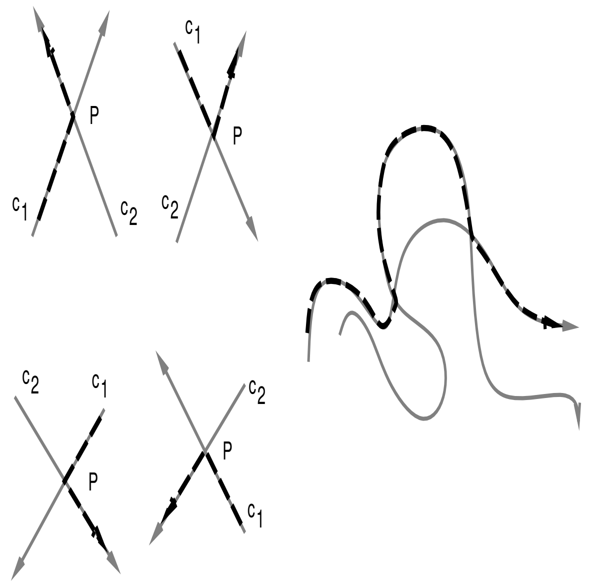


Figure 10: Trimming Curve Intersection

of the patch. An *active v-line* is adjacent to at least one active cell.

4. For each active cell in the strip, triangulate the cell.

5.1 Intersection

We merge two intersecting curves into one by eliminating sections of the curves. When a part of a curve lies in a region trimmed out by another curve, it is redundant and can be ignored. Fig. 10 demonstrates how this is done. The basic idea is very simple. Let two curves c_1 and c_2 intersect at a point $\mathbf{P}$. Turn the picture around, so that both curves are oriented upwards. Below $\mathbf{P}$, keep the curve that is to the left of $\mathbf{P}$ and above $\mathbf{P}$, keep the other curve, which is also to the left of $\mathbf{P}$. For this test we should only look in a small neighborhood of $\mathbf{P}$ i.e. just look at the slopes of c_1 and c_2 at $\mathbf{P}$.

Once we process all intersections, each resulting trim curve is a sequence of piecewise curves and no trim curves intersect except at the patch boundaries. Note that some redundant curves that do not intersect any curve (curve D in Fig. 9). may still remain. These curves are also detected off-line and deleted from the list of curves to be processed.

Now we can tessellate each individual curve on-line and get a piecewise representation: a sequence of points $p_0 \ldots p_n$. While the actual curves do not intersect, their piecewise approximations

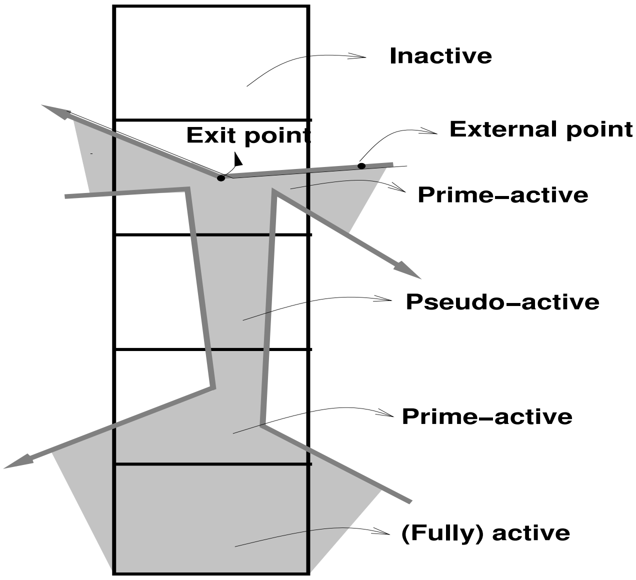


Figure 11: Active v-cells

might. If TOL_d is smaller than the minimum distance between two curves, this does not occur.

5.2 Tracing

For each curve, we need to trace from p_0 to p_n marking any *cell-crossings*: the points where the curve crosses a u-line or a v-line. There are two main purposes of this tracing:

1. Find the range of active v-lines, and for each active v-strip (the strip between two active v-lines), find the active cells.

2. For each cell, find the crossing and the $p_i s$ that lie inside the cell, the *in-points*, and those that are outside the cell and are adjacent to that cell's in-points – the *external points* (Fig 11). The in-points that are adjacent to an external point are called *exit points* of the cell. All cells that lie on the segment between an external and exit point are called *external cells*. Further, if there is a curve that doesn't intersect any cell boundary, and lies completely within a cell, that cell is marked as having a hole. While triangulating such cells, we need to be careful to not triangulate the hole.

We start at the first point p_0 of the curve and move from p_i to p_{i+1}. The minimum and maximum valued v-line a curve crosses are called the *left-end* and *right-end* of the curve respectively. If, further, at left-end , the curve moves from a higher value of u to a lower value, the left-end is called *bounding left-end*. Similarly if at a right-end , the curve moves from a lower value of u to a higher value, the right-end is called *bounding right-end*. The active range of v-lines is defined by the bounding left-end with the maximum v-value and the bounding right-end with the minimum v-value. This is a necessary, but not sufficient, condition for being active. But the only case in which this condition is not sufficient is when parts of a clockwise trimming loop are coincident with the boundary curves $u = 0$ and $u = 1$ e.g. loop B in Fig. 9. The inactive v-lines in such cases lie in the smallest rectangle inside the loop that does not intersect it. For example, in Fig. 9, the v-lines that intersect the loop at $u = 0$ and $u = 1$, the v-lines 4-8, are candidates for being inactive. Of these candidates, the v-lines that do not intersect the trimming loop (v-lines 6 and 7) must be inactive.

The same idea is also used to find the active cells on active v-lines. A cell that is intersected by a trimming curve is definitely active. These cells are called *prime-active* cells (Fig. 11) and can be easily detected. A prime-active cell that has one of its v-edges intersected by a curve is potentially at the boundary of a range of active and inactive cells in its v-strip. If the direction of the curve is away from the cell on the lower valued v-edge or towards the cell on the higher valued v-edge, the cell switches from active to inactive while going from the lower valued u-edge to the higher valued one. In the complimentary case, the cell switches form inactive to active. The first and the last such switch within a cell decide whether the adjacent cells are inactive. Some active cells do not have any in-points and all its corners are trimmed out. These are *pseudo-active* and do not need any processing in the triangulation step. For active cells that are neither prime or pseudo active, joining one of the diagonals of the cell is an appropriate triangulation.

Apart from updating the range of active v-lines and v-cells, the tracing step also marks the cells that the segment $p_i p_{i+1}$ crosses. Further, it updates the current list of in-points of the cell and stores a pointer to the external points. The exit points are also marked.

5.3 Triangulation

Once we know the distribution of points within (and adjacent to) a cell we need to connect these points into a set of triangles that can be sent down the rendering pipeline. Even though the rendered part of the cells can be potentially concave, most of them remain convex in practice and have few edges. Therefore, we optimize our triangulation algorithm for the most general case even if the worst case complexity is a little high.

For each cell we know the polygons that needs to be shaded: these consist of its in-points and

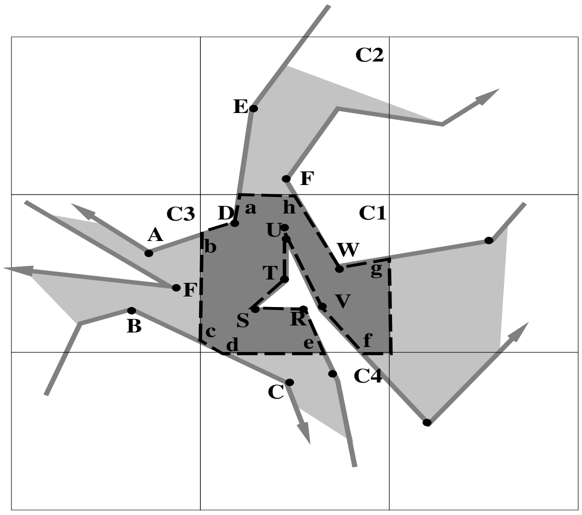


Figure 12: Trimmed Cell

the cell corners that are not trimmed out. To delineate these, we need the cell crossings sorted in counter clockwise order (starting at any crossing at which the curve enters the cell). In practice, since most cells have few crossing of an edge, this sorting step doesn't become a bottleneck. Also, the order of crossing doesn't change from one cell to the next, since trim curves do not intersect any more. Only new crossings need to be inserted to, or deleted from, the sorted list. Furthermore, we do not need the actual intersection point of curves and edges. Just looking at the external and exit points of two crossings, we can decide their order. The polygons are constructed in the following manner:

1. Start at the first cell crossing, X_0, by the curve c_0.

2. Add all in-points on the curve c_0, till it crosses out of the cell at X_c.

3. Add the crossing X_1 next to X_c in the sorted order. If X_c and X_1 lie on different edges of the cell, add the intermediate corners of the cell before X_1. (If no corners are added, and there are no in-points in the cell, this is a pseudo active cell.)

4. If X_1 is the same as X_0, one polygon is complete. Output the polygon, delete all its points from the sorted list. Start a new polygon with the crossing next to X_c as the new X_0.

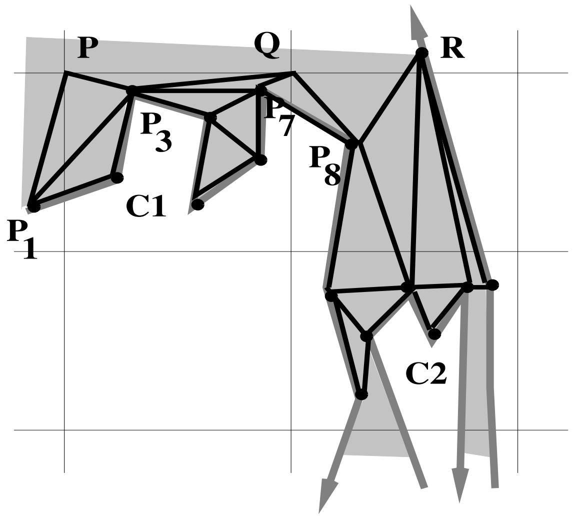


Figure 13: Triangulation

In Fig. 12, if we start at the crossing **a**, we add **D** before moving to the crossings **b-e** in that order. Points **R-V** are added next before the crossing **f** is encountered. The cell corner **P** is added before we move on to **g** since between **f** and **g** we switch edges as the curve went out of the cell at **f**. When we reach back to **a** thus, the polygon is complete. There are no more crossings to process, hence there are no more polygons to generate.

We can triangulate this polygon and send the triangles down for rendering[1]. There is one problem, though. The dark shaded polygon in Fig. 12 has vertices **a-h** that are not on the original tessellation of the curve. This extra points can be at two different parameter values of the curve for two different surface tessellations. While we can avoid cracks by generating a degenerate triangle between each pair of corresponding external and exit points and the crossing, e.g. **AbD** in Fig. 12, the rendered image isn't smooth near these skinny triangles.

The technique can be slightly modified to avoid introducing these extra points. Let us consider the intersection point **b**. If we use the external point **A** instead of **b**, we get a correct triangulation there. Similarly we can replace all crossings with the corresponding external points, and avoid creating extra tessellants on the curve. This polygon can now be triangulated in the following way:

[1]Note that if the capability to render general polygons is available, it will be faster to send these polygons directly down the pipeline. (We will see later that they cannot be rendered yet, though.)

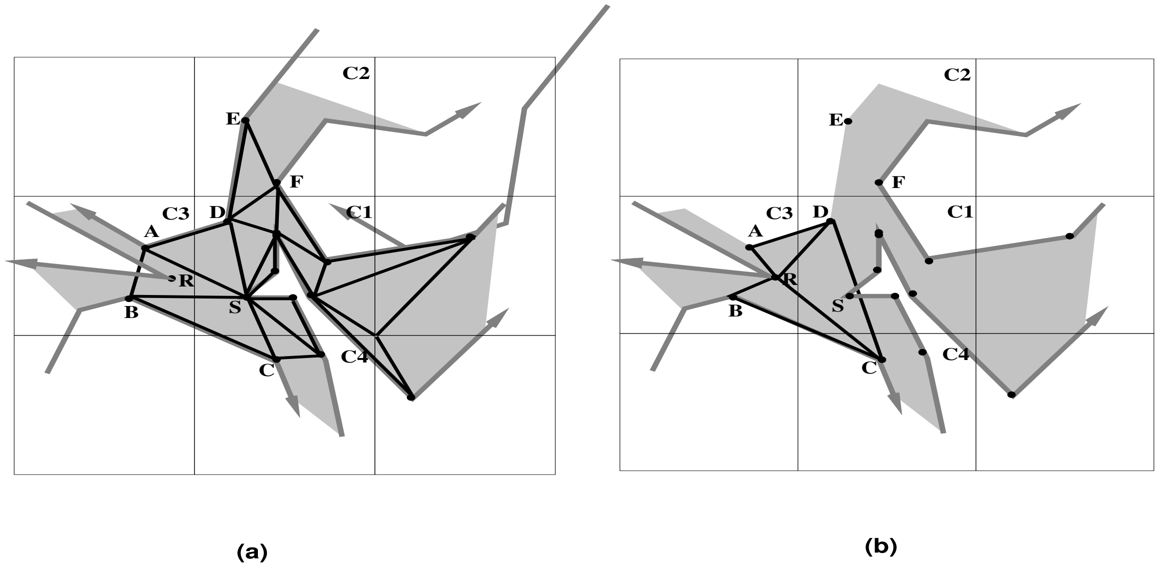


Figure 14: Conflicting Triangulation

- Compute convex envelope inside it and triangulate the region between the envelope and the curve. For example, in the cell C1 in Fig. 13, we find the convex envelope, $P_1P_3P_7P_8$, of P_1P_8, P_1 and P_8 being the consecutive external points.

- Triangulate the region outside the envelope. We connect the point P_3, that is closest to the edge PQ, to generate the triangle P_3PQ. All points to the left of P_3 are connected to P and all the points to the right are connected to Q to generate successive triangles. The same principle is used with each edge that lies in the polygon. If there is only one corner in the polygon, all the other points are connected to that corner. If there are no corners the envelope is triangulated with the corresponding external points, e.g. P_8 and R in the cell C2 in Fig. 13.

The above two steps can be merged into one, and if all the in-points are on the convex envelope, as they normally are, the complete region is triangulated in linear time. Any concave polygon triangulation algorithm [PS85, CTV89, Sei91] can be used to compute the convex envelope while triangulating the region between the envelope and the polygon. If a cell corner belongs to this polygon it should always be connected with the exit and external points corresponding to the crossings enclosing the corner in the sorted order.

There are two new problems with this method of polygon generation, though:

1. The external point of one cell is the exit point of another. If these points are included in

polygons for both the cells, some overlapping triangles can be created. e.g. Figs. 14(a) and (b) show a part of the triangulation for cells C1 and C3 respectively. We cannot draw both triangles **ADS** and **ADR**.

2. Since a cell doesn't keep any information about the triangles of other cells (the idea being that all cells could be processed in parallel on different processors), it can end up drawing wrong triangles. This is clear from Fig. 14(a). The triangle **ABS** intersects a curve in cell C3 and spans an untrimmed region.

One way to avoid these problems is to test if the triangle edges to external points intersect any curve. This is expensive, since even if there are no such intersections, this test must be made. A better way is to postpone drawing a triangle that has an external point for a vertex. Such leftover regions can be done at the end of the triangulation step, in a cleanup step. Once the cell that an external point lies in, is triangulated (C3 in this example), that external point becomes ready for the final triangulation. Consider the external point **A** of cell C1 in Fig. 14(a). We generate a new polygon **ARSD** (Fig. 15) for triangulation. This polygon has all the points in C1 that **A** is adjacent to and those in C3 that its external point **D** is adjacent to. This does waste a part of the work done earlier, but only if there are too many concavities in the region, which is not common. These points are taken in the same respective orders as they were in the individual polygons for C1 and C3. (Notice that the vertices of a polygon are circularly ordered. The points in C3 must start at **A** and the points in C1 must end at **D**, since the direction of the curve is from **D** to **A**.) When a segment spans more than two cells, the points in all intermediate cells that are connected to any of the two end points are also included in the polygon. If one of the points, say $\mathbf{P_0}$ in the generated polygon lies in a cell not considered for this polygon, that cell can be processed now for the sake of efficiency: all the points in that cell, which are adjacent to the external point corresponding to $\mathbf{P_0}$, should be added to the polygon in order[2].

Extra polygons need to be generated for each pair of external points. These extra polygons have only three vertices most of the time. We perform a simple optimization here. If the only points an external point shares a triangle with, are exit points or cell corners, we just discard the triangles generated by this cell and use the triangles generated by the cell the external point lies in. The cells at the ends of the segment must not both throw away their triangles. A simple priority scheme

[2]In fact we can process all pairs of external points. e.g. in Fig. 15, **BC** can be considered with **AD** since the corresponding crossings **b** and **c** are consecutive in the sorted order for cell C1 and the segment **bc** is not trimmed out.

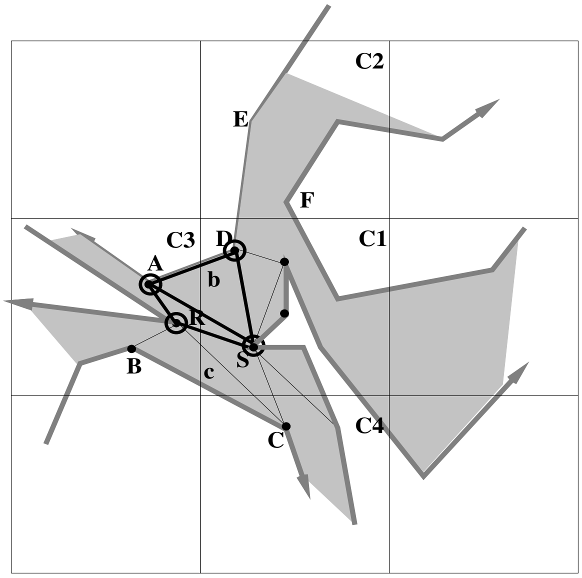


Figure 15: Cleanup Step

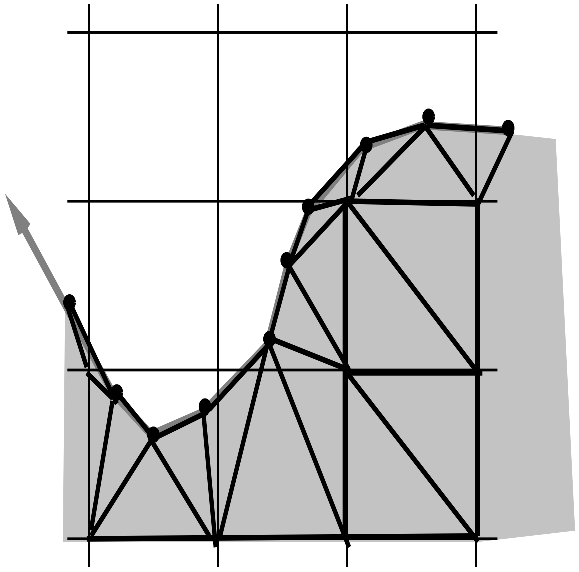


Figure 16: Triangulation: Common Behavior

takes care of that. The cell with lower priority throws away its triangles. Fig. 16 shows a normally occurring case. No extra triangulation is required for this example.

6 Coherence

In an interactive session there is only a small change in the viewing parameters or the scene between successive frames. As a result, n_u, n_v and n_t do not change much between frames. This means that the number of tessellation steps needed on the patch and the trimming curves do not change much and we can reuse many, if not all, triangles from the previous frame.

At each frame we store the evaluated points (and normals) of the surface and their triangulation information in memory. If the bounds for the next frame are higher, we evaluate some additional points and if they are lower we discard some points. When memory is not at a premium, we can retain these extra points. It turns out that, triangle rendering takes more time than triangle generation and can easily become a bottleneck. So while extra points may be retained, the triangulation must be redone using only the correct set of points. In our experience, we hardly ever need to store more than $60-70$ thousand triangles, needing about $3-4$ megabytes of memory. Thus the memory requirement is not stringent for today's graphics systems.

6.1 Incremental evaluation

When the tessellation bound for a patch increases, we always adds complete v-lines or u-lines in the middle of a v-strip or u-strip respectively. This lets us retain the advantages of uniform tessellation, even though the tessellants are not uniformly placed in the parameter space. For each trim curve, we introduce points between two existing points and split that tessellation step. The following discussion talks only about the curve, but updating the tessellation of patch is similar.

Let the tessellation bounds for the previous and current frames be n_t and $\overline{n}_t$ respectively. If we choose $K\ n_t$ steps for the current frame, where K is the smallest integer such that $K\ n_t > \overline{n}_t$, we will need to evaluate $(K-1)\ n_t$ new points. In each interval $K-1$ equi-spaced points are added. Thus we still maintain uniform tessellation. But in this manner we use $K\ n_t - \overline{n}_t$ more steps than needed. For large values of n_t, this could generate too many triangles unnecessarily.

So, instead, we should introduce only $\max(K-2, 0)$ extra tessellant per interval. Now we must decide where the next $\Delta_t = \overline{n}_t - (K-1)n_t$ tessellants are. The intervals that do not satisfy the tolerances are obvious candidates. If there are fewer intervals than Δ_t, we get fewer than $\overline{n}_t$

tessellations but all criteria are still satisfied. In case there are more, we have two options:

1. **Criterion intensive option:** Split all intervals that fail the criteria.

2. **rendering intensive option:** Split the first Δ_t intervals that fail the criteria.

The first option assures that all criteria are always satisfied. It take more running time. While testing whether an interval satisfies the size criterion is simple, it is not so for the deviation criterion. It remains an open problem to test the deviation criterion reliably and fast.

The second option compromises the criteria declaredly: our bounds are based on the assumption that tessellants are uniformly spaced. Of course, we must choose which of the offending intervals to split. This is done cyclically so that an interval gets split twice only after all others have been split at least once. In other words a larger (in the parametric domain) interval is always split before a smaller one. The rendered image can start looking less smooth due to imprecise tessellation. While the deterioration in image quality has a tendency to smooth out over frames, sometimes it may become noticeable. Hence, after every few frames (or while the user pauses), all the stored points are flushed out and a uniform tessellation is recomputed. This resynchronization for different patches is staggered across frames so the glitch doesn't become noticeable. To prevent cracks on patch boundary (including the trim curves), this recomputation must be done together for all patches adjacent to the boundary. A boundary curve is retessellated on all adjacent patches when its associated patch is resynchronized.

6.2 Incremental Triangulation

When most of the points are old, there is no need to retrace all trim curves and retriangulate every region. We can reuse some of the earlier effort. There are two parts to this: adding (deleting) a v-line or a u-line and adding (deleting) a point on the trim curve. While these updates can be handled together, it is easier to talk about them separately.

6.2.1 Tracing

The tracing of trim curve lends itself to coherence very well. We need to make small changes to the cell crossings without tracing through an entire curve.

If we introduce the point P between P_1 and P_2 (Fig. 17(a)), only the crossings of segment P_1P_2 change. In a sense we need to trace only the part P_1PP_2 of the curve. Similarly if we delete the

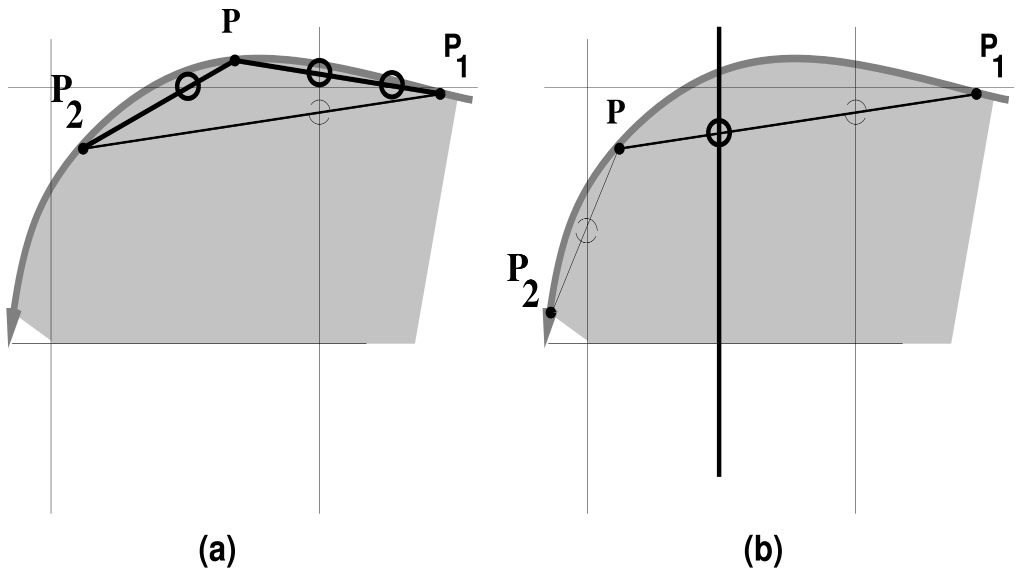


Figure 17: Coherent Tracing

point P between P_1 and P_2, we just need to retrace P_1P_2. The sorted order of crossings of most cells remain mostly unchanged. We occasionally need to insert or delete a few crossings in the sorted list.

If we add a new v-line (Fig. 17(b)), only the curves crossing the adjacent v-lines (and those contained completely within them) potentially cross this new v-line. Again we need to trace only a part of the curve: the sections within the external points of the cells on this v-strip, P_1P_2, need to be traced. Addition of u-lines is processed similarly.

6.2.2 Triangulation

Once we have the new set of crossings we can update the triangulation also. When P is added in Fig. 18, We only need to update the triangles in the cells that P_1P and PP_2 cross. Two cases can occur:

1. We need to trim out some region from the triangulated section. In the first case, P lies inside the triangulated region, and segments P_1P and PP_2 intersect some of the triangles. We need to replace all these triangles. P_1P_2 must be an edge of a triangle since it was a tessellation step of the trim curve. If P lied inside that triangle (P_1BP_2), we would just need to retriangulate (P_1BP_2) and connect each of its vertices and P to replace it with three triangles. If P intersects one of the edges of (P_1BP_2), as in the figure, we need to retriangulate

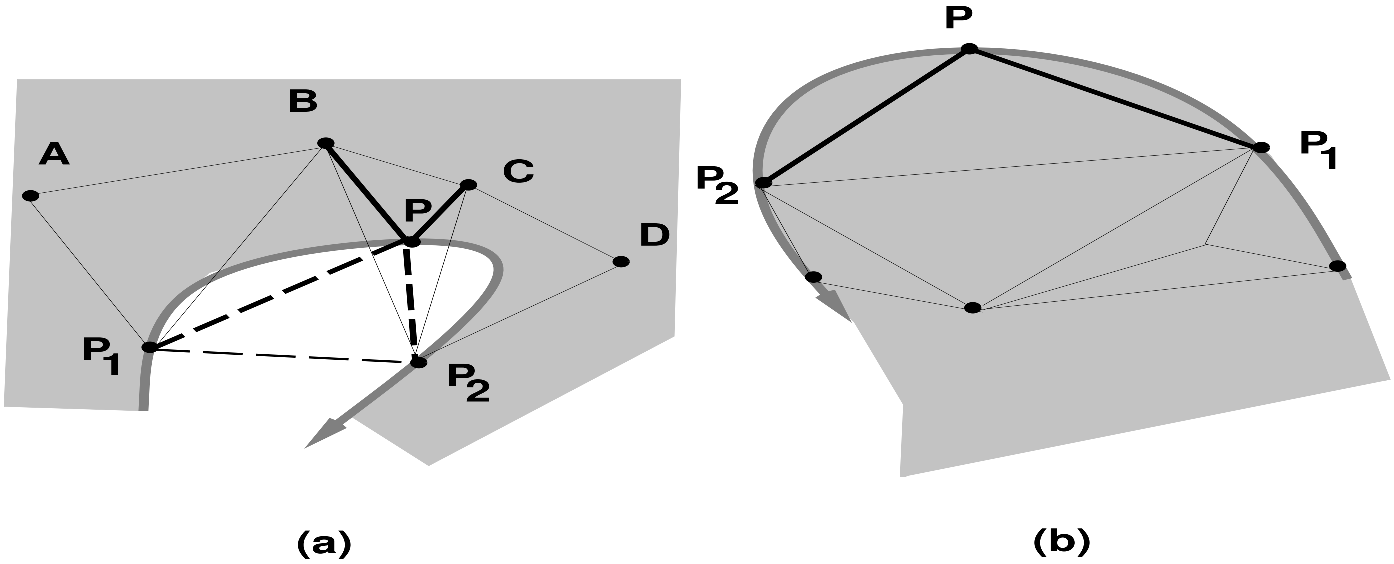


Figure 18: Coherent Triangulation

all triangles whose edges P_1P intersects.

2. We need to add some more region to the triangulated section. Fig. 18(b), we can just add the triangle PP_1P_2. If a new crossing adds a cell corner to the untrimmed region, that corner is added to the triangulation.

When a point is removed, we need to do the reverse of what is done when one is added.

When a v-line is added in or removed from a v-strip, we need to retriangulate the region in the neighborhood of the v-strip. Most of the work is limited to partially trimmed cells. When a v-line is added, some new crossings are created and some new corners are added to the rendered region. These corners are connected to the closest external and exit points. The triangles lying totally inside the new cell can be left alone. The cleanup step needs to be performed on the rest. The fully untrimmed cells are just divided into half and new diagonals drawn. When a v-line is deleted two corners of each cell on the strip get removed. We simply retriangulate the region. u-lines are handled similarly.

7 Implementation and performance

We have implemented our algorithm on the SGI-VGX and also on the Pixelplanes 5 system. The Pixelplanes 5 configuration included 30 graphic processors (though one of them is a master processor

Model	# Patches	SGI-GL impl.	Our basic algorithm	Patch Culling	Coherence
Goblet	72	1(3 fps)	1.91	2.67	7.11
Pencil	576	1(1.1 fps)	1.85	2.89	8.47
Car Panel	1872	1(.08 fps)	2.13	2.19	7.82
Trimmed teapot	32	1(5 fps)	2.21	2.41	5.28

Table 2: Relative performance of the techniques on SGI-VGX

and does not perform the computations) and 14 renderers [Fea89]. The algorithm achieves load balancing by distributing neighboring patches onto different processors statically. The algorithm does not require any inter-processor communication (or shared memory) during execution. As a result it can be easily ported to any other multiple processor machine.

The performance of the algorithm on the SGI-VGX has been shown in Table 2. The SGI-GL implementation is based on the algorithm presented in [RHD89, Nas93] and has a microcoded geometry engine implementation for surface evaluations. Although it is difficult to compare two different algorithms and implementations, we tried to run the algorithms on models undergoing the same transformation between two frames.

The Dragon model consists of 5354 Bézier patches and we are able to generate about $12 - 15$ frames a second. This is the largest and most complex model we have tried so far. It turns out that more than 70% of the time during each frame is spend in polygon rendering. Due to coherence and back-patch culling the time spent in the polygon generation phase is relatively small. As a result we expect the algorithm performance to improve *directly* with future improvements in the polygon rendering capabilities of the graphics systems.

8 Conclusions

We have presented algorithms for interactive display of large scale models on current graphics systems. The algorithms are portable and make use of improved techniques based on uniform subdivision, back-patch culling, frame-to-frame coherence and trimmed patch rendering. These algorithms can be easily ported onto machines with multiple processors as well, though for large

Figure 19: Dragon: 5354 bi-cubic Bézier patches [FB88]

Figure 20: Trimmed Teapot

scale models the polygon rendering performance is the bottleneck.

Although we have significantly improved on earlier algorithms for bound computations, the algorithm at times produces dense tessellation for some models. Due to this the polygon rendering phase becomes the bottleneck. In terms of the overall performance it is worthwhile to use more sophisticated algorithms for bounds computation so that lesser time is spent in the polygon rendering phase.

9 Acknowledgements

We thank Henry Fuchs, Elaine Cohen, Anselmo Lastra, and Russ Fish for their helpful discussions. The models of teapot, goblet and the pencil were provided by the Alpha 1 group at Utah. The Dragon model is a contribution of David Forsey. We thank them for letting us use their models.

References

[AES91] S.S. Abi-Ezzi and L.A. Shirman. Tessellation of curved surfaces under highly varying transformations. *Proceedings of Eurographics'91*, pages 385–97, 1991.

[AES93] S.S. Abi-Ezzi and L.A. Shirman. The scaling behavior of viewing transformations. *IEEE Computer Graphics and Applications*, 13(3):48–54, 1993.

[Ake93] K. Akeley. Reality engine graphics. In *Proceedings of ACM Siggraph*, pages 109–1116, 1993.

[Baj90] C.L. Bajaj. Rational hypersurface display. In *Symposium on Interactive 3D Graphics*, pages 117–27, Snowbird, UT, 1990.

[Bea91] R. Bedichek et. al. Rapid low-cost display of spline surfaces. In *Proceedings of advanced reserach in VLSI*, MIT Press, 1991.

[Cat74] E. Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, University of Utah, 1974.

[Che93] F. Cheng. Computation techniques on nurb surfaces. In *SIAM Conference on Geometric Design*, Tempe, AZ, 1993.

[Cla79] J. H. Clark. A fast algorithm for rendering parametric surfaces. *Proceedings of ACM Siggraph*, pages 289–99, 1979.

[CTV89] K. Clarkson, R. E. Tarjan, and C. J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon. *Discrete Comput. Geom.*, 4:423–432, 1989.

[Dea89] T. Derose et. al. Apex: two architectures for generating parametric curves and surfaces. *The Visual Computer*, 5:264–276, 1989.

[DN93] M.F. Deering and S.R. Nelson. Leo: A system for cost effective 3d shaded graphics. In *Proceedings of ACM Siggraph*, pages 101–108, 1993.

[Far90] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.* Academic Press Inc., 1990.

[FB88] D. Forsey and R.H. Bentels. Heirarchical b-spline refinement. In *ACM SIGGRAPH*, pages 205–212, 1988.

[Fea89] H. Fuchs and J. Poulton et. al. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of ACM Siggraph*, pages 79–88, 1989.

[FK90] D.R. Forsey and V. Klassen. An adaptive subdivision algorithm for crack prevention in the display of parametric surfaces. *Proceedings of Graphics Interface*, pages 1–8, 1990.

[FMM86] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *CAGD*, 3:295–311, 1986.

[For79] A.R. Forest. On the rendering of surfaces. *Computer Graphics*, 13(2):253–59, 1979.

[Kaj82] J. Kajiya. Ray tracing parametric patches. *Computer Graphics*, 16(3):245–254, 1982.

[LC93] W.L. Luken and Fuhua Cheng. Rendering trimmed nurb surfaces. Computer science research report 18669(81711), IBM Research Division, 1993.

[LCWB80] J.M. Lane, L.C. Carpenter, J. T. Whitted, and J.F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of ACM*, 23(1):23–34, 1980.

[LR81] J.M. Lane and R.F. Riesenfeld. Bounds on polynomials. *BIT*, 2:112–117, 1981.

[Luk93] W.L. Luken. Tessellation of trimmed nurb surfaces. Computer science research report 19322(84059), IBM Research Division, 1993.

[MD92] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves. In *Graphics Interface '92*, pages 232–241, 1992.

[Nas93] R. Nash. Silicon Graphics, Personal Communication, 1993.

[NSK90] T. Nishita, T.W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics*, 24(4):337–345, 1990.

[PS85] F.P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[RHD89] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. In *Proceedings of ACM Siggraph*, pages 107–117, 1989.

[Roc87] A. Rockwood. A generalized scanning technique for display of parametrically defined surface. *IEEE Computer Graphics and Applications*, pages 15–26, August 1987.

[SC88] M. Shantz and S. Chang. Rendering trimmed nurbs with adaptive forward differencing. In *Proceedings of ACM Siggraph*, pages 189–198, 1988.

[Sed89] T.W. Sederberg. Algorithms for algebraic curve intersection. *Computer-Aided Design*, 21(9):547–555, 1989.

[Sei91] R. Seidel. A simple and fast randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry Theory & Applications*, 1(1):51–64, 1991.

[SL87] M. Shantz and S. Lien. Shading bicubic patches. In *Proceedings of ACM Siggraph*, pages 189–196, 1987.