Interactive Display of Large NURBS Models*

Subodh Kumar, Dinesh Manocha, Anselmo Lastra
Department of Computer Science
University of North Carolina
Chapel Hill NC 27599
{kumar,manocha,lastra}@cs.unc.edu
http://www.cs.unc.edu/~{kumar,manocha,lastra}

Abstract

We present algorithms for interactive rendering of large-scale NURBS models. The algorithms convert the NURBS surfaces to Bézier surfaces, tessellate each Bézier surface into triangles and render them using the triangle-rendering capabilities common to current graphics systems. This paper presents algorithms for computing tight bounds on surface properties in order to generate high quality tessellation of Bézier surfaces. We introduce enhanced visibility determination techniques and present methods to make efficient use of coherence between successive frames. In addition, we also discuss issues in parallelization of these techniques. The algorithm also avoids polygonization anomalies like cracks. Our algorithms work well in practice and, on high-end graphics systems, are able to display models described using thousands of Bézier surfaces at interactive frame rates.

Additional Keywords and Phrases: Intersection, NURBS, NURBS, Tessellation, Triangulation, Visibility, Interactive Display, CAD, Parallel algorithm.

1 Introduction

Current graphics systems are capable 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 geometric models are described in terms of non-uniform rational B-spline (NURBS) surfaces, not polygons. This class includes Bézier surfaces and other rational parametric surfaces like tensor product and triangular patches. Description of large-scale models like automobiles, submarines, airplanes, etc. contain tens of thousands of surfaces. Surface fitting over scattered data or surface reconstruction are other techniques that produce a large number of NURBS surfaces. Current renderers of sculptured models on commercial graphics systems, while faster than ever before, are not able to render at interactive rates for applications involving virtual worlds, walkthroughs and immersive design.

Curved surface rendering has been an active area of research for more than two decades. 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 graphics systems. As a result, algorithms based on polygonization are, in general, faster. A number of different methods, based on adaptive or uniform subdivision, have been proposed for polygonization of surfaces [LR81, Baj90, AES93, BEWD91, DBB+89, Luk93, LC93, Che93, SC88, SL87, FK90, RHD89, Roc87, AES91, FMM86]. These are based on adaptive or uniform subdivision of NURBS surfaces. In particular, Rockwood et. al. [RHD89] have proposed a real time algorithm for trimmed surfaces. A variant of this algorithm has been implemented as SGI GL and OpenGL primitives. However, the bounds used for tessellating the Bézier surfaces are, in general, not tight for rational surfaces and in some cases even undersample the surface. Some techniques to improve the quality of tessellation and the efficiency of its computation are presented in [AES93, AES91, FMM86]. The algorithm presented in this paper exhibits considerable improvements over previous algorithms.

This paper presents the components of an algorithm for interactive[1] display of large-scale NURBS models (see Color plate B) on current graphics systems. At an abstract level, the NURBS representation is converted to Bézier and the resulting Bézier surfaces are uniformly tessellated and triangulated based on the current viewing position. The algorithm computes *tight bounds* for on-line tessellation. It performs *back-patch culling*, an extension of back-face culling to curved-surface solid models, and makes use of *coherence* between successive

*Supported in part by Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, DARPA ISTO Order A410, NSF Grant MIP-9306208, DARPA Contract DABT63-93-C-0048, NSF Grants CCR-9319957 and CCR-9625217, ONR Contract N00014-94-1-0738 and NSF/DARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract 8920219.

[1] By interactive display we mean a rendering rate of more than 10–15 frames a second.

frames. It is portable and its actual performance is a function of the resources available on a system (memory, CPUs, and special purpose rendering hardware etc.). Our current implementation on a 1 processor Silicon Graphics Onyx with RealityEngine 2 can display more than *a thousand Bézier surfaces* and on Pixel-Planes 5 [Fea89], about *thirty thousand Bézier surfaces* at interactive frame rates. These surfaces were of degrees varying from two to 15. See color images for examples. On multiple-processor machines the algorithm statically partitions the model, distributing it to the processors to balance the load. A preliminary version of this paper has appeared as [KML95].

In the rest of this paper, a basic familiarity with NURBS and Bézier surfaces is assumed. 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 explain *back-patch culling*. The algorithm for dynamic tessellation of Bézier surfaces, based on tight bounds, is presented in Section 4. The use of coherence is demonstrated in Section 5. We discuss our implementation in Section 6 and compare its performance with that of earlier algorithms.

2 Polygonal Approximation of Surfaces

Any polygonization-based surface rendering algorithm first needs to allocate resources to generate these polygons. The desirable tessellation must be computed, and vertices and normals evaluated. The total time is thus a function of the number of polygons generated. The performance of polygon rendering is system dependent and typically is a function of the number of polygons and the size and distribution of these polygons on the screen.

It is possible to compute a one-time highly dense polygonization and to render all the polygons for each frame. In this case, almost no time is spent in polygon generation and all of the time is spent on 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 approach requires hundreds of megabytes of storage, and the capability of rendering hundreds of millions of polygons per second. We can reduce the demand on polygon rendering capability by computing different *levels of detail* of each surface and, for each frame, choosing one of the approximations as a function of the viewing parameters. Unfortunately, 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. This minimizes the demand on the graphics hardware. The resulting algorithm is based on adaptive subdivision and spends considerable time on the polygon generation for each frame. As a result, the generation of polygons may become the bottleneck, thus preventing interactive performance.

Depending on the system architecture, there are two possible design goals for polygonal approximation:

- *Minimize total time:* If the two stages are performed sequentially on the same processor, we only need to minimize the sum of times spent in each stage. This is typically the case when rendering without specialized graphics hardware.
- *Balance individual time:* If the two stages are pipelined, we need to balance the time spent in generating with that in rendering the polygons, while minimizing the time in each stage. This case, being the norm for interactive graphics today, is the subject of this paper.

2.1 Overview

Our approach to interactive display of large-scale models has the following features:

1. *Uniform Tessellation:* We tessellate a surface patch uniformly in its parametric domain and generate the triangles in $\mathcal{R}^3$. Empirical tests show that triangle generation time far exceeds the triangle rendering time for adaptive tessellation of patches on current graphics systems. Therefore we chose the simpler and faster uniform tessellation.
2. *Visibility Processing:* We perform simple on-line computations to isolate patches not visible from the current viewpoint. This includes use of viewing frustum culling as well as a new technique, *back-patch culling*.
3. *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 store the triangulation of large-scale models. We use tight bounds to optimize the number of polygons generated.
4. *Coherence:* We make use of coherence between successive frames to minimize the overall computations for triangle generation. In particular, we perform incremental triangulation, re-triangulating only when necessary.
5. *Parallelization:* The computation is distributed over all available processors such that any data that

a processor needs is locally available. This is done without any significant duplication of data.

2.2 Overall Pipeline

An overall pipeline of the polygonization algorithm is shown in Fig. 1. It consists of four phases. Initially, we perform a visibility-based rejection check — back-patch culling. It compares a volume corresponding to a superset of normals of the Bézier patch with the viewing direction and rejects the patch if the entire volume is not visible. Otherwise the patch is tessellated into triangles as a function of the viewing parameters. The resulting set of triangles are transformed and rendered. The actual implementation and performance of each phase varies with the graphics system. In particular, we demonstrate the performance on an SGI Reality Engine and a Pixel-Planes 5 graphics system.

2.3 Background

This section introduces our notation and the mathematical theory on which some of our methods are based. The 3D coordinate system in which the NURBS model is defined is referred to as *object space*. We assume a left handed system. Viewing transformations, like rotation, translation and perspective, map it onto a viewing plane known as *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.

Given a NURBS model, we use knot insertion to decompose each NURBS surface into a series of rational Bézier patches [Far93]. Knots spaced closer than a user specified tolerance[2] are coerced to the same value before knot insertion. Each NURBS surface is thus decomposed into a set of Bézier surfaces.

A Bézier patch $\mathbf{F}$ of degree $m \times n$ defined by parameters $u, v \in [0, 1]$, is specified by a mesh of control points $C_{ij}, 0 \leq i \leq m, 0 \leq j \leq n$:

$$\mathbf{F}(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} C_{ij} B_i^m(u) B_j^n(v)$$

where the Bernstein polynomial B is given by

$$B_i^n(t) = \begin{pmatrix} n \\ i \end{pmatrix} t^i (1-t)^{n-i}$$

We drop u and v from the notation, whenever it is implicit in the context. In homogeneous coordinates a Bézier patch $\mathbf{F}$ includes four components, (X, Y, Z, W). The fourth coordinate, referred to as the weight, is assumed to be either all positive or all negative for all control points. We use concepts of Gauss maps from differential geometry and of resultants from elimination theory. A brief introduction to these topics may be found in the appendix.

[2] We used a value of 2×10^{-5}

3 Visibility Computations

Given a large model consisting of Bézier patches, not all patches are typically visible from a particular 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 detect 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 [KM94]. In this section, we show that it is relatively simple to perform a visibility check to find most of the patches that are completely invisible. A Bézier surface,

$$\mathbf{F}(u, v) = (X(u, v), Y(u, v), Z(u, v), W(u, v)),$$

is contained in the convex polytope of the control points [Far93]. 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 whether any part of it lies inside the viewing volume.

3.2 Back-patch Culling

Large-scale models typically consist of a large number of small patches. Given a closed solid model whose boundary is composed of Bézier patches, many of the patches are occluded because they are located on the part of the model opposite to the viewer. Back-facing polygons are commonly culled out to improve the rendering performance. Analogously, for a Bézier patch, if all the surface normals point away from the eye point we refer to it as a *back patch* (Fig. 2a).

In general, a point p on a patch with normal $\vec{n}$ is backfacing if

$$\vec{ep} \cdot \vec{n} > 0,$$

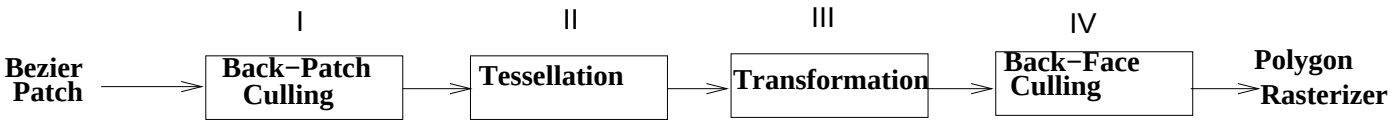


Figure 1: *Overall Pipeline for Rendering NURBS Models*

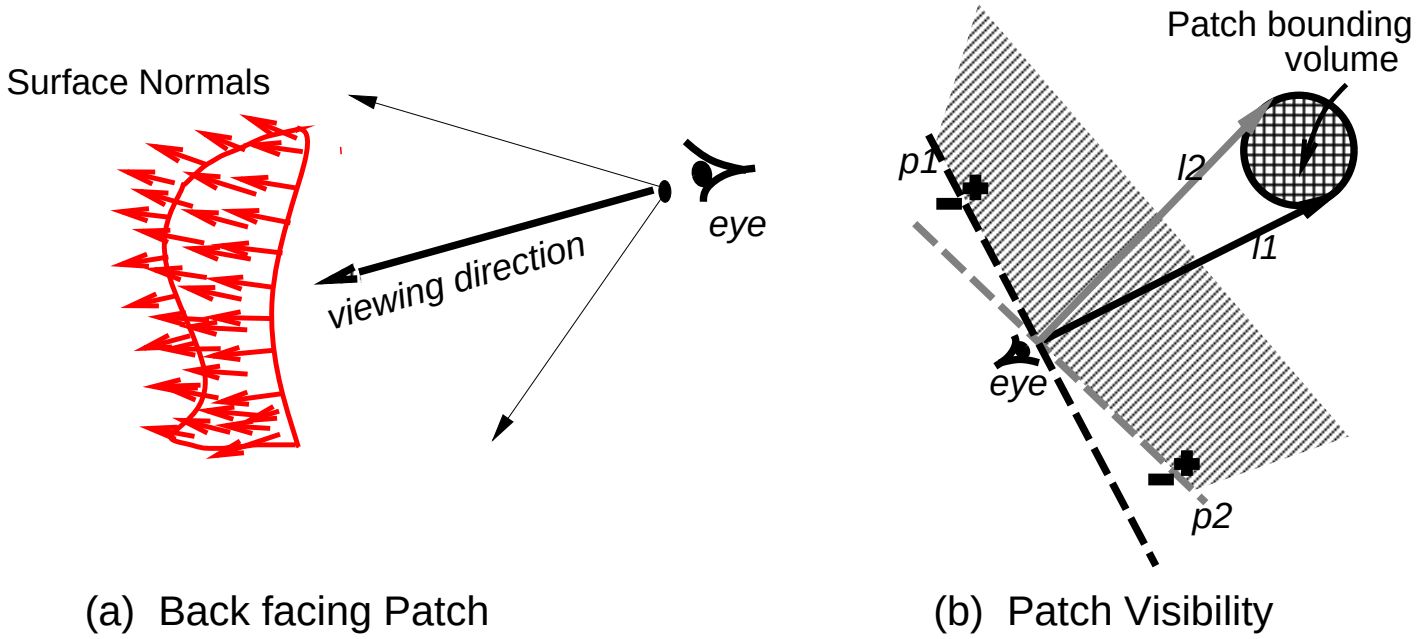


Figure 2: *Back Facing Normals Visibility Computation*

where e is the eye point. In other words, a patch, $\mathbf{F}$, is backfacing if, $\forall u, v \in [0,1]$, $\vec{n}(u,v)$ makes an acute angle with the vector joining the eye to $\mathbf{F}(u,v)$.

If S is a bounding sphere for the patch, with radius r and center C, we can find the region in space such that a the point $p(u,v)$ in S is backfacing if $\vec{n}(u,v)$ lies in that region. Indeed, if all $\vec{n}(u,v)$'s lie in this region, the entire patch must be back facing. We call this region the backpatch region. This is demonstrated in 2D in Fig. 2(b). The rays $l1$ and $l2$ bound the patch, and lines $p1$ and $p2$ are, respectively, perpendicular to them. The half spaces $p1^-$ and $p2^-$ potentially contain normal directions corresponding to visible points: normal directions lying outside this cone cannot form an acute angle to any ray bounded by $l1$ and $l2$, which bound the direction $\vec{ep}$ for all points on the patch. Hence the intersection of halfspaces $p1^+$ and $p2^+$, call it $\mathbf{H}$, must form the backpatch region. If the entire pseudo-normal surface lies in this region, the entire patch is back-facing.

We compute a minimum-volume, eight vertex, axis-aligned box, B_N, bounding the control points of the pseudo-normal surface, $\mathbf{N}$. Each point on $\mathbf{N}(u,v)$ corresponds to a direction on $\mathbf{F}(u,v)$ and P_N and B_N define a multi-faced pyramid in which all these directions lie [KM96]. Testing for visibility reduces to checking whether each of these control points, or just the bounding box $\mathbf{B}_N$, is in the half-space $\mathbf{H}$.

In fact, we may use a rectangular box or the convex hull to bound the surface as well. This increases the effectiveness of the technique but also increases the number of tests performed. Similarly, for the normal patch, a spherical bounding volume can be used. For patches with highly varying normal directions, the pseudo-normal surface seldom lies in the backpatch region, rendering the test ineffective. Indeed there exist few view points from which all parts of such patches are back-facing. Subdividing such patches and performing visibility test on each sub-patch yields increased effectiveness.

For most solid models we have examined, back-patch culling eliminates about 30-40% of the patches. Table 1 lists the performance of backpatch culling for some representative models. Note that some models have almost half of the patches culled away. Since most patches do not have highly varying curvatures, we have found that using bounding boxes is good enough. Using back-patch

Model	Figure	Num. Patches	% patches culled	% time reduction per frame
Pencil	Color Plate A	570	33%	24%
Dragon	Color Plate A	5354	39%	31%
Car	Color Plate A	10,012	32%	19%

Table 1: *The effectiveness of back patch culling*

culling improves the frame rate by $20 - 35\%$.

For small systems where calculating or keeping extra bounding boxes is too expensive, the view-frustum can be used as the bounding volume of the patch, especially for applications that use a narrow field of view.

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 tessellation for each frame. Uniform tessellation involves choosing constant step sizes along each parameter, and evaluating points on a regular grid in the parametric domain of a patch. It normally generates more polygons than are necessary for a given viewpoint. By comparison, adaptive tessellation may generate fewer polygons. In general, adaptive tessellation algorithms [For79, Luk93, Che93, SC88, SL87, FK90, Far93] recursively subdivide patches until each patch is flat or small enough. Empirical results, which we describe next, suggest that, for lage models, uniform subdivision methods are faster in practice than adaptive subdivision methods. (Also see [FMM86].)

4.1 Adaptive Tessellation

Different algorithms for adaptive subdivision are presented in the literature [For79, Luk93, Che93, SC88, SL87, FK90, Far93]. We stored the sequence of viewpoints from a user-runof the system. For each of these viewpoints, we computed, off-line, an adaptive tessellation of each patch using the following algorithm:

1. Transform the control points of the patch to screen space.
2. Approximate the patch by a quadrilateral defined by its four corner control points.
3. – If the maximum deviation of the patch from the approximating rectangle is more than 10 pixels, say at the parametric point (u_1, v_1), subdivide the patch into four sub-patches at isoparametric lines $u = u_1$ and $v = v_1$ and recursively tessellate each sub-patch.
   – if the diagonal of the approximating rectangles is longer than 50 pixels, subdivide the patch into four sub-patches at isoparametric lines $u = 0.5$ and $v = 0.5$ and recursively tessellate each sub-patch.
4. Store the (u, v) values of the tessellation of the patch, when all its subpatches have been satisfactorily tessellated.

Thus, on-line, only a few polygons are generated and no tests to check whether the tessellation is satisfactory need to be performed, but the points on the surfaces and their normals are evaluated. We recorded the time spent in the four phases of the pipeline shown in Fig.1. For a given patch, all these computations were performed by the same processor. The total number of processors used in an experiment was determined by the amount of memory needed for the model. This polygonization time, including the transformation time, was compared with the overall time, including the rasterization. As Table 2 shows, the adaptive tessellation stage was already the bottleneck. This means that any more computation that is of the order of the number of polygons generated will considerably slow down the system. It also implies that any time spent in computation of uniform tessellation must be relatively insignificant. In fact, as shown in Section 5, using frame-to-frame coherence we are able to achieve just that.

4.2 Uniform Subdivision

The following are some of the reasons that rendering time of algorithms that use uniform tessellation tend to out-perform those that use adaptive tessellation:

1. Simplicity of tessellation algorithm. Uniform tessellation involves one bound computation per patch, while adaptive tessellation typically requires a number of them.
2. Simple evaluation algorithms based on uniform forward differencing or 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).

Model	Number of Patches	Number of Frames	Number of Processors	Polygonization Time	Total Time
Pencil	570	100	1	5.4 sec	6.1 sec
Dragon	5354	100	2	17.1 sec	17.8 sec
Car	10,012	100	3	19.2 sec	19.4 sec

Table 2: *Adaptive tessellation: The processor computing the polygons is almost always busy on Pixel-Planes 5 with one rasterizer.*

3. No good and simple algorithms are known for quick determination of the flatness of parts of a patch, a measure necessary for dynamic tessellation.

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

5. Most of the large-scale models consist of surfaces that do not have highly varying curvatures. This is indeed the case after converting B-spline models to Bézier surfaces. Adaptive subdivision does well on surfaces with highly varying curvatures. Uniform tessellation may oversample such surfaces.

The performance of uniform tessellation algorithms is a direct function of the step sizes. A number of related criteria can be used to define the image quality and thus fix the step size of tessellation.

1. *Size criterion*: The size of the approximating triangles in screen space should be bounded.

2. *Deviation criterion*. The deviation of triangles from the surface should be bounded.

3. *Tangent criterion*: The change of tangent between consecutive tessellants should be bounded.

4. *Normal criterion*: The change of normal between consecutive tessellants should be bounded.

5. *Normal Deviation criterion*: The deviation of triangle normals from the surface normals should be bounded.

To compute the bound on the step size in the parametric domain to satisfy these criteria, we need to compute bounds on polynomials. There is considerable literature on computation of such bounds. [LR81, Roc87, AES91, FMM86].

The criteria listed above are related to each other and not all of them need to be used. Further, these criteria are functions of the first, second and higher order derivatives 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. The deviation criterion generates good approximations but is computationally expensive. The other criteria are even more costly to compute, and may not have significant impact on the image quality. 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 object space as a preprocessing step. The step size is computed as a function of these bounds and viewing parameters [LR81, AES91, FMM86].

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 [RHD89, Roc87].

We require only two mathematical formulations: one for the computation of the 'deviation' bounds, and another for the 'difference' bounds. Let us first consider the bound computations for the size criterion.

A rational Bézier surface

$$\mathbf{F}(u,v) = (X(u,v), Y(u,v), Z(u,v), W(u,v)) = \left(\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\mathbf{r}_{ij}B_i^m(u)B_j^n(v), \sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}B_i^n(u)B_j^m(v)\right), \quad (1)$$

where $\mathbf{r}_{ij}$ are the control points in object space, w_{ij} are the weights and B_i^m, B_j^n are the Bernstein polynomials. After applying all of the viewing transformations (rotations, translation, perspective), let the new control points in 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]:

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

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

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 the perspective transformation, all of the polynomial parametrizations are transformed into rational formulations. Furthermore, since the weights tend to vary considerably (typically by an order of two to three), these bounds *oversample* the curved surface for a given TOL. As a result, the polygon rendering becomes a bottleneck for the overall process.

4.3 Bound Computation

We compute improved bounds for the rational surfaces in 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 [AES93, FMM86]. An algorithm for computation of bounds based on the size criterion is presented in [AES91]. However, we must modify the bounds presented in [AES91] to use the mean value theorem for vector valued functions. By performing exact extrema computation, for a given TOL, we improve the tightness of their bounds. We illustrate the derivation of tighter bounds for Bézier curves. It is applied in a similar manner to surfaces. Given a rational curve

$$\mathbf{C}(t) = \big(x(t), y(t), z(t), w(t)\big)\,,$$

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 compute tight bounds on the length of the vector $\mathbf{C}(t+\delta) - \mathbf{C}(t)$. It follows from the Mean Value theorem that

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

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

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

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 magnitudes 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 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 common roots of

$$\begin{aligned} fx_u(u,v) &= 0 \\ \text{and } fx_v(u,v) &= 0 \end{aligned}$$

or it occurs at the boundary of the domain. The maximum at the boundary corresponds to one of the roots of one of the following equations

$$\begin{aligned} fx_v(0,v) &= 0 \\ fx_v(1,v) &= 0 \\ fx_u(u,0) &= 0 \\ fx_u(u,1) &= 0 \end{aligned}$$

It may also occur at one of $fx(0,0), fx(0,1), fx(1,0)$ or $fx(1,1)$. We compute all roots of these equations and pick the maxima of those.

Therefore, the problem of computing the maximum derivative vector is equivalent 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:

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

$$X_{uv}W^2 - XWW_{uv} - W_vX_uW + 2W_uXW_v - W_uX_vW = 0.$$

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 but using the method described in the appendix we are able to compute accurate solutions without numerical problems. Note that all these computations are part of the preprocessing stage.

Similarly, the maxima 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 maxima of these three functions differ significantly, we subdivide the surface patch and compute the maxima in the subdivided domains using the roots of the equations shown above. Each of the subdivided surfaces are handled separately. The complexity of the bound computations reduces significantly for polynomial surfaces as $W(u, v) = 1$ and the resulting equations are of lower degrees.

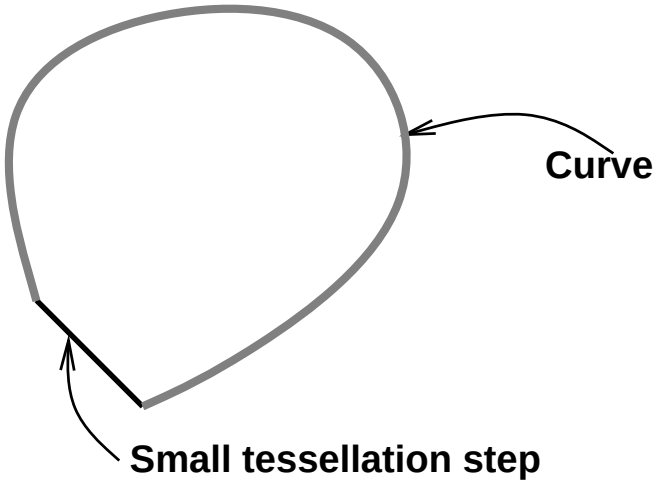


Figure 3: Undersampling of a curve with high curvature

Given these bounds in object space, we compute the step size in 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].

The bounds for the other difference criteria can be computed similarly. In one case we seek to bound the length of the vector connecting two points on the surface, and in the others we seek to bound the angle between tangents or normals. These can be computed by starting with $\mathbf{F}_u, \mathbf{F}_v$ and $\mathbf{F}_u \times \mathbf{F}_v$. Owing to the inefficiency of such computations, we do not explicitly satisfy any of these criteria.

4.4 Deviation Bounds

For small values of TOL the size-criterion bound, derived above, works quite well. In case the surface area is small and curvature is high, they may undersample the surface. For example see Fig 3: 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. The bound on deviation is computed using results from [FMM86]: For a linearly parametrized triangle $T = l(u, v)$ between three points on a surface at $l(0, 0)$, $l(l_1, 0)$ and $l(0, l_2)$:

$$\max_{(u,v)\in T} \|\mathbf{F}(u, v) - l(u, v)\| \leq \frac{1}{8}(l_1^2 M_1 + 2 l_1 l_2 M_2 + l_2^2 M_3),$$

where

$$M_1 = \max_{(u,v)\in T} \|\mathbf{F}_{uu}(u, v)\|,$$

$$M_2 = \max_{(u,v)\in T} \|\mathbf{F}_{uv}(u, v)\|, \text{ and } M_3 = \max_{(u,v)\in T} \|\mathbf{F}_{vv}(u, v)\|,$$

We can reduce the computations of M_1, M_2 and M_3 to finding zeros of polynomials and solve them using techniques from elimination theory. However, in practice this method typically oversamples the surface as the degree of these polynomials are quite high and the bounds become loose. Therefore we instead use a simple estimate based on the geometry of the transformed control points in screen space. To the number of steps, n_u, computed for the size criterion, using the method detailed in the previous section, we add:

$$D \times \max(\|\mathbf{F}_u(\epsilon_u, 0) - \mathbf{F}_u(0, 0)\|, \|\mathbf{F}_u(\epsilon_u, 1) - \mathbf{F}_u(0, 1)\|,$$

$$\|\mathbf{F}_u(1, 0) - \mathbf{F}_u(1 - \epsilon_u, 0)\|, \|\mathbf{F}_u(1, 1) - \mathbf{F}_u(1 - \epsilon_u, 1)\|)$$

where D is user-defined constant, and $\epsilon_u = \frac{1}{n_u}$. Similarly we add

$$D \times \max(\|\mathbf{F}_v(0, \epsilon_v) - \mathbf{F}_v(0, 0)\|, \|\mathbf{F}_v(1, \epsilon_v) - \mathbf{F}_v(1, 0)\|,$$

$$\|\mathbf{F}_v(0, 1) - \mathbf{F}_v(0, 1 - \epsilon_v)\|, \|\mathbf{F}_v(1, 1) - \mathbf{F}_v(1, 1 - \epsilon_v)\|)$$

to n_v. This works well in practice because these numbers provide a fair idea of the curvature of the surface for most real life models. Most surfaces do not have highly varying curvature and are uniformly parametrized.

4.5 Comparison of Methods

We empirically compared our bound with those of Rockwood et. al. [Roc87] and Abi-Ezzi & Shirman [AES91]. For each model, these comparisons were performed over a large number of user-driven model inspection runs with different user-specified tolerance values. We collated the averages of the number of triangles generated for each model. The degrees of the patches in these models were between two and three in u as well as v. Some models, e.g. Dragon, contained no rational patches, keeping [Roc87]'s bounds tight. For a given tolerance, our bounds result in about 31% fewer triangles than [Roc87] and about 20% fewer than [AES91]. Color plate B displays the wireframes and shaded images of the pencil and goblet models computed using the three methods.

Model	Number of Patches	Our Algorithm		[Roc87]'s Algorithm		[AES91]'s Algorithm	
		Num. Tris.	Ratio	Num. Tris.	Ratio	Num. Tris.	Ratio
Goblet	72	535	1	790	1.48	659	1.23
Pencil	570	4720	1	6875	1.45	5810	1.23
Dragon	5354	19220	1	22500	1.15	22755	1.18

Table 3: *Relative comparison of the number of triangles generated for a given tolerance*

4.6 Crack Prevention

Since the bound for required tessellation for each patch is evaluated independently, different tessellations on two adjacent patches are possible. This could result in cracks in the rendered image. To address this issue [RHD89, FMM86] suggested that the amount of tessellation at the boundary be based solely on the boundary curve, and a strip of filling triangles be generated at the boundary. However, 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 their control points. A common example occurs when one of the patches is subdivided into two, resulting in a T-joint at the common boundary.

At T-joints, there is no way to prevent cracks without using information about the adjacent patches. To illustrate this fact, suppose we calculated the required tessellation for a boundary curve of a patch $\mathbf{F}$. We can always subdivide one of the patches adjacent to this boundary, say $\mathbf{F}'$, into two, say $\mathbf{F}'_1$ and $\mathbf{F}'_2$, and reparametrize each of them in such a way that the tessellation points on the boundaries of $\mathbf{F}'_1$ and $\mathbf{F}'_2$ are different from $\mathbf{F}'$ and hence from $\mathbf{F}$.

Our algorithm computes the adjacency information between the patches during 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 axes to compute the overlapping pairs. For each pair of overlapping boxes the algorithm checks whether the two curves form a common boundary as follows:

Let the two boundary Bézier curves be $C_1(t)$ and $C_2(t)$. The curves are common only if

1. There exist t_0 and t_1 such that both $C_1(0) = C_2(t_0)$ and $C_1(1) = C_2(t_1)$, and
2. At least one of t_0 and t_1 lie in [0,1].

In degenerate cases these two conditions may be satisfied for boundary pairs that are not common. For example in Fig. 4a, Patch C may be marked adjacent to Patch A. Since we assume that the model does not have any holes, only one of the marked patches is actually adjacent, Patch B in this case. Such conflicts are resolved by considering a third point, arbitrarily chosen, on the boundary curve. We cannot have three common points on patches that do not have common boundaries.

Location of these points reduces to an *inversion* problem: given a point P and a curve C, find the parameter value t such that $C(t) = P$. Again using techniques from elimination theory, we first solve for t using the equation for the x coordinate:

$$X(t) = P_{\mathtt{x}}$$

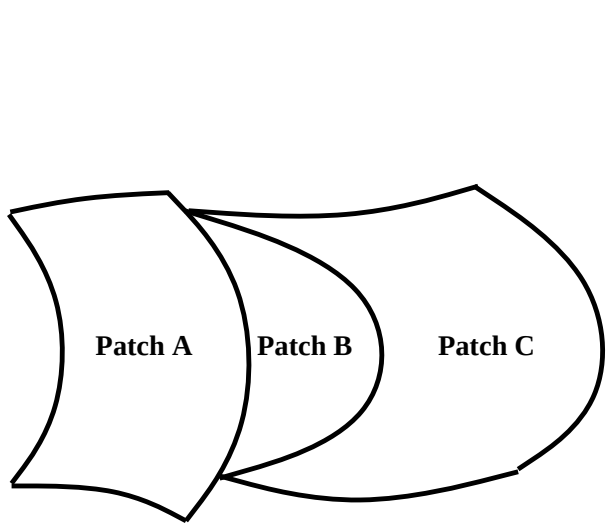
At the actual root t', the expressions $Y(t') - P_{\mathtt{y}}$ and $Z(t') - P_{\mathtt{z}}$ must also evaluate to zero. In practice the expressions do not evaluate to zero due to finite precision arithmetic. The t' that minimizes $\|C(t) - P\|$ is chosen.

We *associate* one of the adjacent patches, chosen arbitrarily, with each patch boundary. If we have two different representations for the boundary curve, we store the representation of the boundary curve of the associated patch with the other one also. To calculate the bounds on the curves and tessellate them, we use this stored representation.

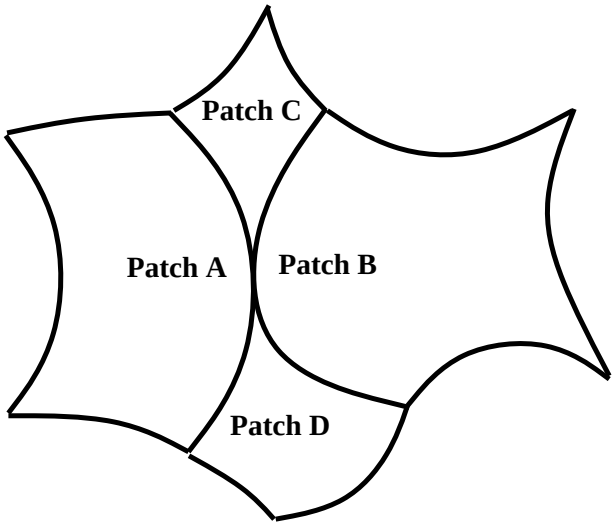
Clearly, this algorithm does not handle cases when one boundary is tangent to another. Fortunately we do not need to compute such adjacencies. The topological consistency of the approximation is still maintained if the point of adjacency is always used in the tessellation. That this actually occurs can be seen by noting that, on each side of the tangency, there exists a patch with one corner at the point of tangency (see Fig. 4b). To ensure that, we always pick the corners as tessellants.

5 Coherence

Typically, the change in the position of the model on screen between successive frames is small. As a result, the bounds for tessellation do not change much between successive frames. Most times the change in n_u and n_v is small, if not zero. We exploit this coherence by performing a minimal amount of computation to calculate the new polygonization of the curved model.



(a) Degenerate Boundary Pairs: A and C



(b) Tangent Boundary Pairs A and B
Complete adjacency for Patch A exists
without including Patch B

Figure 4: *Degenerare adjacencies*

At each frame we cache the vertices of the generated polygons, and their surface normals. As the new bounds are computed, we perform a few extra evaluations on the surface. If we need fewer triangles now as compared to the last frame, some polygons may be removed from the list. For most of our models, we typically do not need to store more than about 100,000 triangles, requiring less than 5 megabytes of memory. With more memory, some triangles that are not immediately needed can also be stored. Thus the memory requirements are not stringent for current graphics systems.

Given the tessellation size for the last frame $(\overline{n}_u, \overline{n}_v)$ and that for the current frame (n_u, n_v), we want to update the polygonization while attempting to satisfy the tolerances closely. Let $\|\overline{n}_u - n_u\| = \Delta_u$ and $\|\overline{n}_v - n_v\| = \Delta_v$. For simplicity we present a description of the algorithm for tessellation along the u-axis. It is applied in a similar manner along the v-axis.

Let us first consider the case when $n_u > \overline{n}_u$. We need to choose $n_u - \overline{n_u}$ additional points in the domain $[0, 1]$, such that the resulting polygonization is smooth. One simple solution is to use tessellations that are powers of two. Thus we need to introduce $\overline{n}_u$ new tessellants subdividing the old tessellation, thereby halving the step size. For large values of $\overline{n}_u$, this results in a much denser tessellation than is required. Hence, instead of adding all the $\overline{n}_u$ new tessellants, we pick just $n_u - \overline{n}_u$. Each of these is chosen as follows:

> Of the candidate intervals we introduce the new tessellant in the intervals across which the change in the magnitude of the derivative vector is the maximum.

Formally, let $n_{u_2} = 2^{\lceil \lg n_u \rceil}$ be the next power of 2 for n_u. Thus there are at most $\frac{n_{u_2}}{2}$ intervals to choose from, and of the $\frac{n_{u_2}}{2}$ tessellants lying in the range $[\frac{n_{u_2}}{2}+1, n_{u_2}]$, the i^{th} tessellant is added at $u = \frac{1}{n_{u_2}} + \frac{2i}{n_{u_2}}$. The chosen i is based on the derivative vectors: for tessellants at u_1, u_2 on the isoparametric line $v = v'$, let $\kappa_{v'}(u_1, u_2) = \|\mathbf{F}_u(u_2, v') - \mathbf{F}_u(u_1, v')\|^2$. We drop u_2 when it is understood to be the tessellant at smallest u greater than u_1, i.e. the one next to u_1 (see example in Fig. 5). We store the sum κ_{01} of $\kappa_0(u_1)$ and $\kappa_1(u_1)$ for each evaluated tessellant in [0,1). Of these, the $\frac{n_{u_2}}{2}$ candidate intervals are maintained in the sorted order. The next tessellant is added at the i corresponding to the head of this list.

Note that two sorted lists are maintained – a current list and a future list. The newly subdivided intervals do not become candidates till the required tessellation becomes greater than n_{u_2}. The two lists are merged and a new list created when $n_u = n_{u_2}$ i.e. it becomes a power of two. The algorithm is:

1. If $CurrentList$ is empty, $CurrentList := FutureList$, Create empty $FutureList$.
2. Subdivide the interval corresponding to head$(CurrentList)$, delete the head.

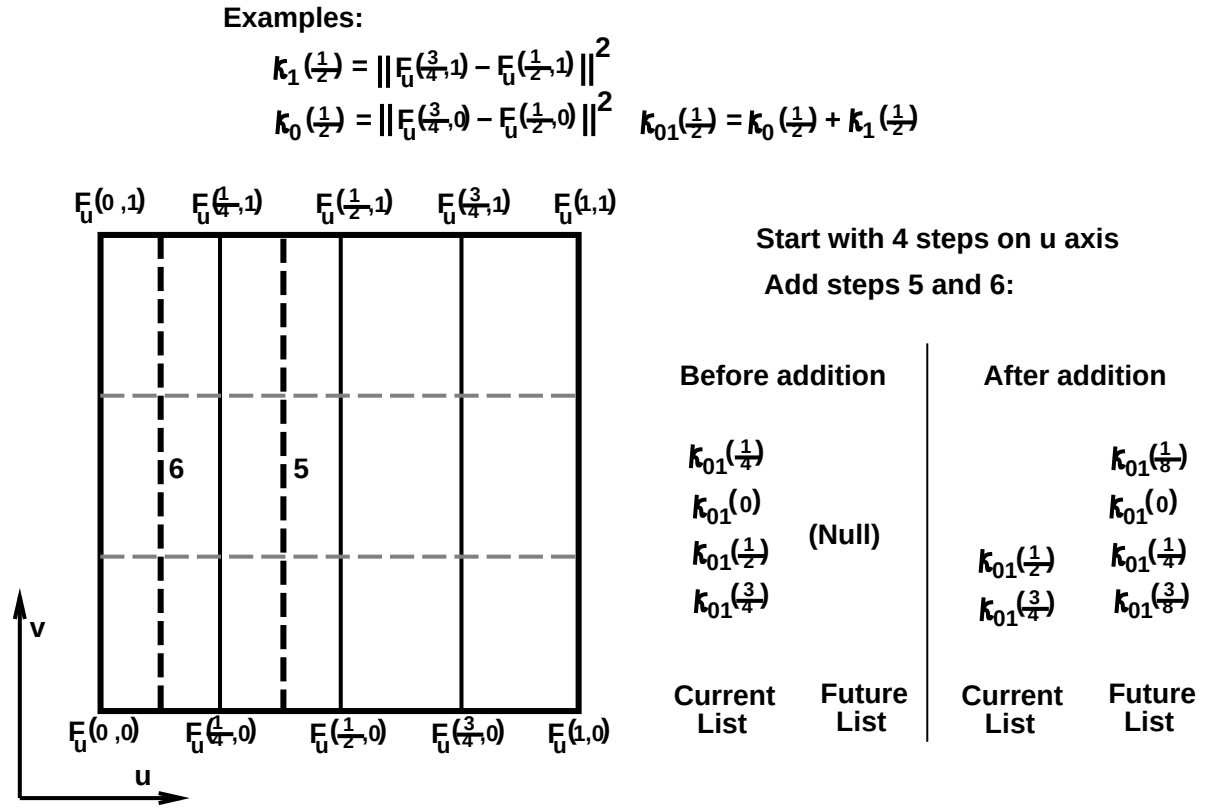


Figure 5: *Example of Incremental Tessellation*

3. Compute the derivative at the new tessellant. Compute new κ's.

4. Add the left and right half intervals to $FutureList$, in their sorted positions.

In case $n_u < \overline{n}_u$, we need to discard some tessellants. This is done analogously to the previous case. This time the last interval in the sorted order is removed first.

Notice that this algorithm does not preserve the uniformity of tessellation. However, we always decompose the domain into rectangles whose edges are parallel to the u and v axis. Furthermore, whenever we introduce an additional tessellation along the u or v axes, all the points are computed based on a generalized Horner's rule or forward differencing which still takes linear time. This technique works well in practice because it includes some benefits of adaptive tessellation, increasing tessellation where the curvature is high. In addition, in doing so, the general efficiency of uniform tessellation is not compromised.

The graph in Fig. 6 shows the effect of coherence on rendering rate. This graph shows the frame rendering times of a short inspection of the car model shown on the color plate. Notice that not only does the coherence based rendering provide a speedup of about $7-8$, but it also exhibits a more consistent frame rate. The short peak in the rendering time, when all required points were evaluated every frame, occurred when most of the model went off screen. On average, the coherence-based scheme required the tessellation of only 15-20% of the model each frame.

In fact, since even in dynamic environments only local changes are made to a model most of the time, the coherence-based optimization is quite effective in such cases as well.

6 Implementation and performance

We have implemented our algorithm on a Silicon Graphics (SGI) R3000 with a VGX graphics accelerator, a SGI Onyx (single CPU) with a RealityEngine 2, and on the Pixel-Planes 5 system. The Pixel-Planes implementation is fully parallel, using the maximum number of available processors.

The performance of the algorithm on the SGI Onyx is shown in Table 4. The images were rendered with Gouraud shading. The standard SGI-GL implementation is based on the algorithm presented in [RHD89, Nas93] and has a microcoded geometry engine implementation

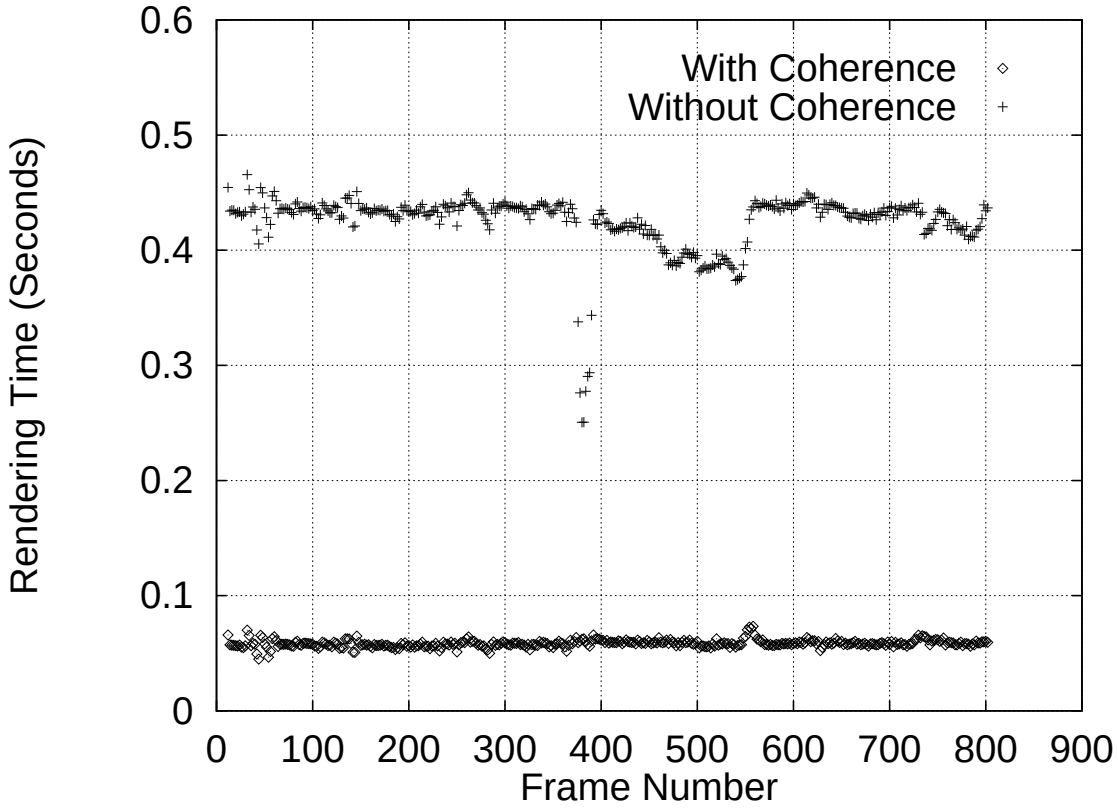


Figure 6: *Effect of Coherence: Time taken to render per frame with and without coherence*

for surface evaluations. Although it is difficult to compare two different algorithms and implementations (for example, the design constraints may be different), we performed the following experiments using identical sets of viewing parameters. Also, a count of the number of patches is not necessarily the correct measure of model or rendering complexity. However, assuming that the model was designed to solve a particular problem, such as mechanical design, and not designed for rendering speed, a count of patches gives, in general, a fair idea of performance.

Table 4 shows the relative speedups of our algorithm on the SGI Onyx. The third column shows the performance of the standard GL implementation as a baseline, while the fourth shows the performance of our algorithm with no optimizations. The fifth column shows backpatch culling only, while the sixth shows the effectiveness of coherence. Note that the visibility preprocessing optimizations improve performance significantly. Since the optimizations, phases I and II in Fig. 1, are performed on the workstation's CPU, any reduction in the number of triangles generated during visibility and tessellation result in better rasterization rate. Currently, we are able to render models consisting of seven to eight hundred Bézier patches at $12 - 16$ frames a second.

6.1 Parallel Implementation

Pixel-Planes 5 [Fea89] uses extensive parallelism to increase rendering performance. This has become the practice in high-performance graphics accelerators [Ake93]. Figure 4 presents a block diagram of the Pixel-Planes 5 system. Front-end geometry processing, such as transformation, clipping, and setup for rasterization, is performed on the Graphics Processors (GPs) which contain Intel i860 RISC microprocessors running at 40 MHz, 8 MB of main memory, and communications hardware. Triangle rasterization, and shading is performed on renderer boards which contain arrays of 128 by 128 1-bit processors with local memory [Fea89] and an instruction sequencer. The processing units are connected by a 160 million word per second ring communications network.

Since we have access to the GPs of Pixel-Planes 5, a parallel implementation of the tessellation algorithm seemed natural. Even though Pixel-Planes 5 is a retained-mode graphics accelerator, a feature of the software architecture is the ability to call user-programmed routines running on the GPs. These routines may generate arbitrary geometry in immediate mode for the rendering engine to display. This feature has been used

Model	Num. Patches	SGI-GL primitive	Our basic algorithm	Patch Culling	Coherence
Goblet	72	1(4.3 fps)	1.91	2.67	7.11
Pencil	570	1(1.9 fps)	1.85	2.89	8.47
Dragon	5354	1(.1 fps)	2.13	2.19	7.82

Table 4: *Speedup due to the techniques (on SGI-Onyx)*

successfully for problems which require close coupling between computation and the generation of geometry [Ban92].

The tessellation algorithm is implemented as a set of user functions running on the GPs. The algorithm does not require any inter-processor communication during execution. This property not only improves the parallel speedup, but also will make it easier to port the code to another multi-processor machine. Note that a disadvantage with running phases I and II of the tessellation on the same processors responsible for rendering is that any time spent executing tessellation code is subtracted from the rendering time. This makes some of the visibility optimizations less advantageous on the Pixel-Planes 5 implementation than on the SGI implementation.

The 8 MB of memory on each GP node allows us to take advantage of frame-to-frame coherence by caching the triangles generated by the tessellation for a previous frame. During display list traversal, we examine the current tessellation for each patch. If the cached tessellation is within the current bounds, it is rendered, otherwise, a new tessellation is computed. This coherence technique provides a considerable increase in performance.

It is not easy to evaluate the tessellation performance of a particular implementation of this algorithm separately from the triangle rendering performance of the machine on which it is executing. In Figure 7, we show the total system performance - tessellation and rendering as a function of the number of GPs, for three models, a simple Utah teapot modeled with 32 Bezier patches, a car body panel consisting of 1700 patches, and a dragon modeled with 5354 patches.

The experiments were run on a medium-sized Pixel-Planes 5 configuration with a maximum of 31 GPs, and 11 renderers. We varied the number of GPs to obtain an idea of the speedup obtained from parallelism. The size of the dragon model constrains us to a configuration with a minimum of 20 GPs. The graphs show the frame rate for a high resolution frame buffer (1280×1024 pixels). The update rate of the Pixel-Planes 5 frame buffer in high-resolution mode is limited to approximately 25 frames a second. We have been able to achieve $10 - 20$ frames per second on models consisting of five to ten thousand Bézier patches.

6.2 Load Balancing

Load balancing is done statically, as one of our goals is to eliminate communication between processors. Each processor is allocated a set of patches and passed the control points for each patch. Once this distribution is complete, the processors must transform their part of the model each frame. If the patches allocated to a processor, call it P, are adjacent in world space, and if the user zooms in to that part of the model, processor P becomes highly overloaded. At the same time, processors with patches occupying a small area on screen may be idle.

We experimented with a number of distribution schemes. A random distribution scheme [EGT90] does not perform well for our application. The reason is that the cost of rendering a patch can vary significantly. We made an explicit adjacency-based distribution and achieved better performance (see Table 5). This distribution ensures that adjacent patches are allocated to different processors, thus making sure that the set of 'zoomed-in' patches are well distributed across processors. We associate a cost with each patch. The cost of a patch **F** of degree $m \times n$ is given by

$$m \times n \times V_F$$

where V_F is the volume of the convex hull bounding its control points. The total cost for a processor is the sum of costs of the patches allocated to it. The load-balancing algorithm for model distribution is listed below:

For each patch i

> Let *Allocated* = Set of Processors with a patch adjacent to i
> Let the set *Candidate* = *AllProcessors - Allocated*
> if *Candidate* is `null`, let *Candidate = AllProcessors*
> Allocate i to the processor P in *Candidate* with the minimum *current_cost*
> Update the *current_cost* of P

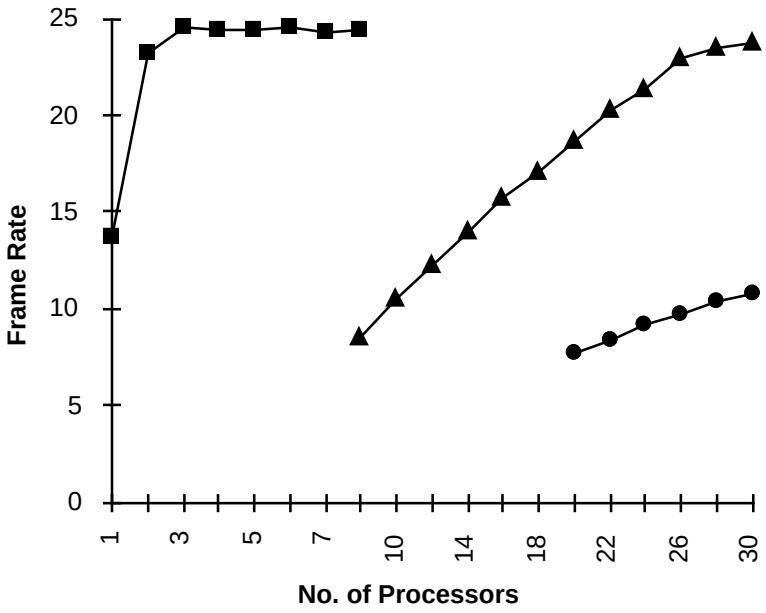


Figure 7: *Frame rate for three different curved-surface models running on Pixel-Planes 5 as the number of processors varies. Markers: Squares – Utah teapot (32 patches), Triangles – Car panel (1700 patches), Circles – Forsey's Dragon (5354 patches).*

Table 5 shows the effectiveness of the two algorithms. The values listed are the ratio of maximum load to minimum load across all processors. For each model, 10 user-driven sequences, of 500 frames each, were recorded, the maximum load-imbalance for each sequence was recorded, and the average of these 10 maxima was reported. These are runtime loads, i.e. load is equivalent to the time spent by these processors tessellating and transforming the model (see the pipeline in Fig. 1).

Model	Num. Patches	Random distribution	Adjacency based distribution
Pencil	570	5.7	2.1
Dragon	5354	8.9	1.8
Car	10012	6.4	2.9

Table 5: *Ratio of maximum load and minimum load across all processors*

6.3 Visibility Preprocessing and Bounds

On average, visibility preprocessing improves the frame rate by about 25%. The actual performance is a function of the model and the graphics system. In particular, all the four phases of the pipeline shown in Fig. 1 are implemented on the GP's on Pixel-Planes 5. On the other hand, phase I and II are implemented on the host CPU on the SGI Onyx and the tessellated triangles are transformed, checked for back-face culling and scan-converted on the hardware rendering pipeline. Therefore, all the four phases in Fig. 1 constitute the *triangle generation phase* on Pixel-Planes 5, whereas it consists of phase I and II on the SGI Onyx.

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 triangle rendering phase often becomes the bottleneck. In terms of the overall performance, it may be worthwhile to use more sophisticated algorithms for bounds computation so that fewer triangles are generated, thus alleviating the triangle rendering bottleneck. This is an especially attractive option for implementations, such as those on the SGI machines, where the tessellation is being performed independently of the graphics accelerator.

7 Conclusions

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

is the bottleneck. In this paper we have demonstrated these techniques on tensor-product surface models only. However, they are easily extended to models composed of triangular patches as well. The preprocessing is costly for applications involving interactive design. It would be useful to extend some of techniques presented in this paper to cases when models can be modified on-line.

The techniques described in this paper have been extended to trimmed Bézier surfaces [KM95]. In particular, coherence allows us to efficiently triangulate simple polygons without any artifacts. In addition, the back-patch detection scheme can be used to find patches that potentially contain the silhouette — this can be efficiently utilized by an algorithm performing more adaptive tessellation.

References

[AES91] S.S. Abi-Ezzi and L.A. Shirman. Tessellation of curved surfaces under highly varying transformations. *Proceedings of Eurographics*, pages 385–397, 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 109–116, 1993.

[Baj90] C.L. Bajaj. Rational hypersurface display. *ACM Computer Graphics*, 24(2):117–127, 1990. (Symposium on Interactive 3D Graphics).

[Ban92] D. Banks. Interactive manipulation and display of two-dimensional surfaces in four-dimensional space. *ACM Computer Graphics*, 26:197–207, 1992. (Special Issue on Symposium on Interactive 3D Graphics).

[BEWD91] R. Bedichek, C. Ebeling, G. Winkenbach, and T. DeRose. Rapid low-cost display of spline surfaces. In C. Séquin, editor, *Proceedings of advanced reseach in VLSI*, pages 340–355, MIT Press, 1991.

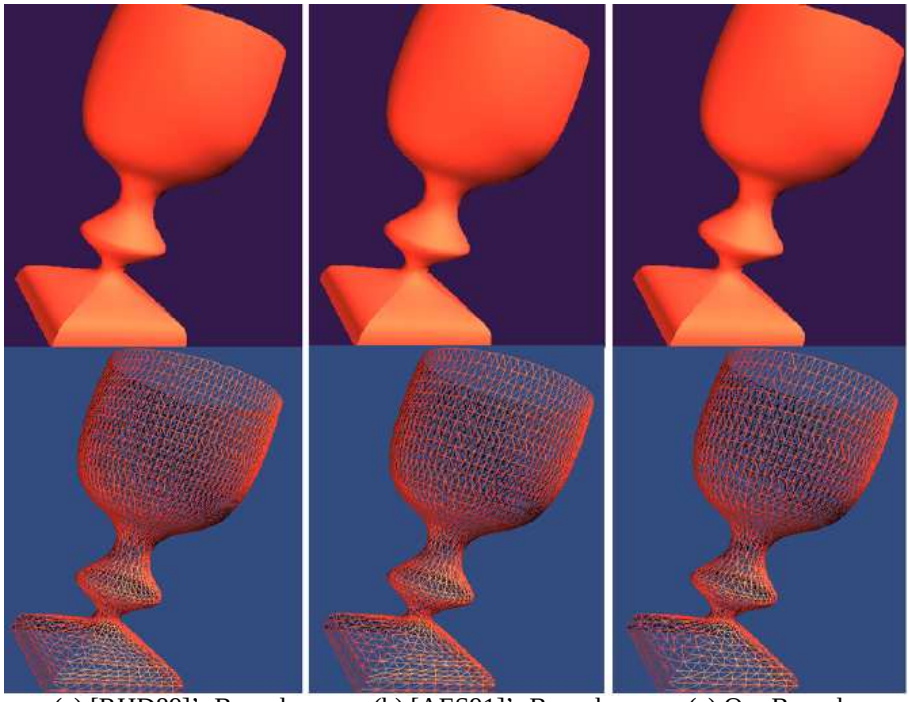
[Cat74] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

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

[Cla79] J. H. Clark. A fast algorithm for rendering parametric surfaces. *ACM Computer Graphics*, 13(2):289–299, 1979. (SIGGRAPH Proceedings).

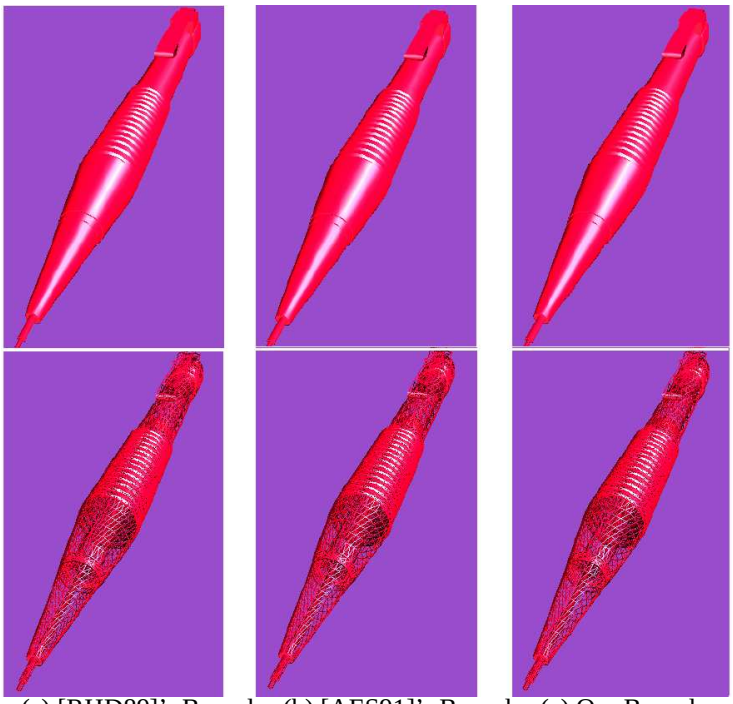
[DBB+89] T. DeRose, M. Bailey, B. Barnard, R. Cypher, D. Dobrikin, C. Ebeling, S. Konstantinidou, L. McMurchie, H. Mizrahi, and B. Yost. Apex: two architectures for generating parametric curves and surfaces. *The Visual Computer*, 5(5):264–276, 1989.

[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.



(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 8: *Comparison of previous bounds to ours: Fewer triangles with similar visual quality*

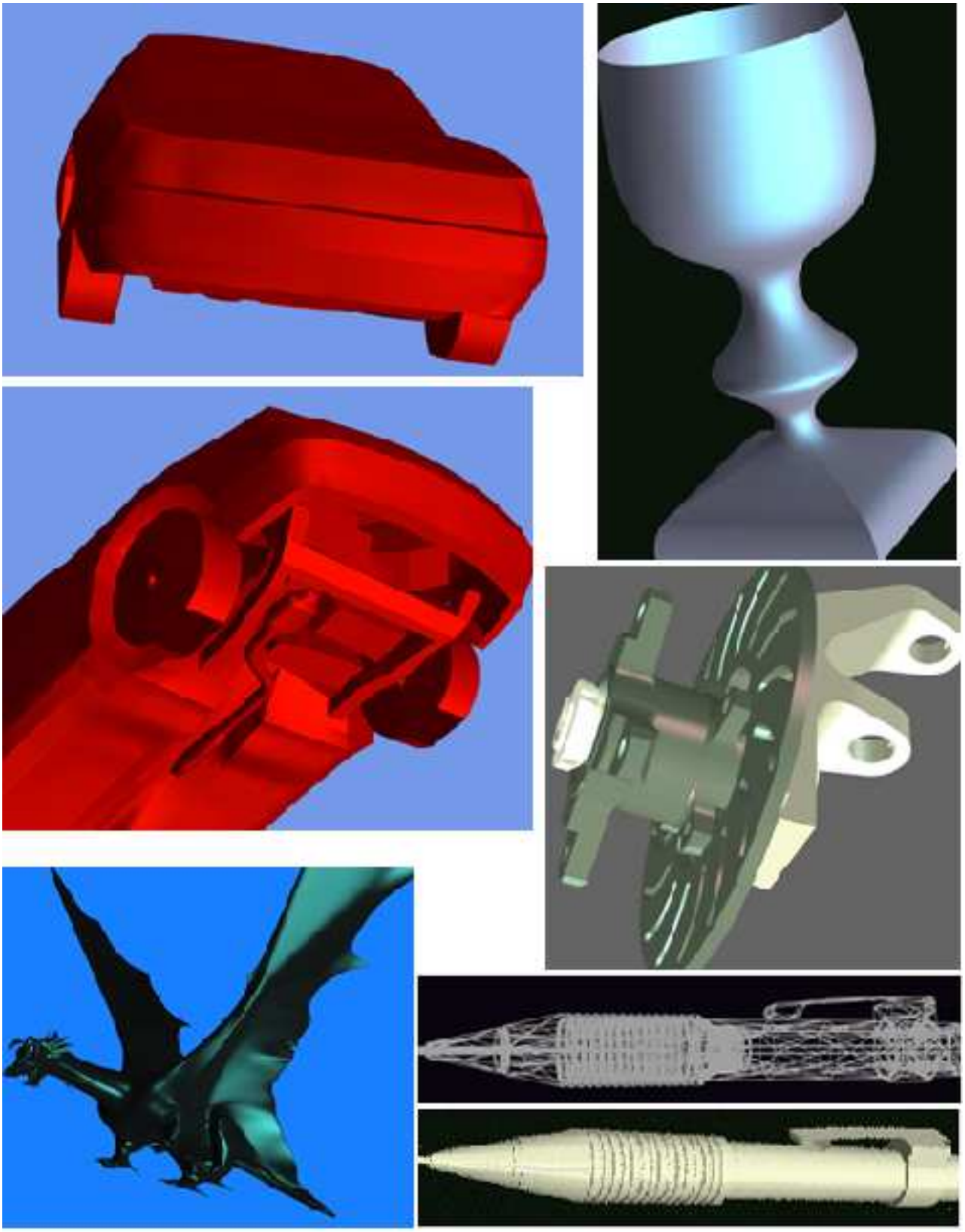


Figure 9: *Some of the models used to test our implementation. The Ford car model consists of 10,012 patches, the Alpha_1 goblet has 72 patches, the Alpha_1 brake-hub has about 600 patches, Alpha_1 pencil consists of 570 patches and Forsey's dragon has 5354 patches.*

[EGT90] D. Ellsworth, H. Good, and B. Tebbs. Distributing display lists on a multicomputer. *ACM Computer Graphics*, 24(2), 1990. (In Symposium on Interactive 3D Graphics).

[Far93] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1993.

[FB88] D. Forsey and R.H. Bartels. Heirarchical b-spline refinement. *ACM Computer Graphics*, 22(4):205–212, 1988. (SIGGRAPH Proceedings).

[Fea89] H. Fuchs and J. Poulton et al. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *ACM Computer Graphics*, 23(3):79–88, 1989. (SIGGRAPH Proceedings).

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

[FMM86] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3(4):295–311, 1986.

[For79] A.R. Forrest. On the rendering of surfaces. *ACM Computer Graphics*, 13(2):253–259, 1979. (SIGGRAPH Proceedings).

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

[KM94] S. Krishnan and D. Manocha. Global visibility and hidden surface algorithms for free form surfaces. Technical Report TR94-063, Department of Computer Science, University of North Carolina, 1994.

[KM95] S. Kumar and D. Manocha. Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design*, 27(7):509–521, July 1995.

[KM96] S. Kumar and D. Manocha. Hierarchical visibility culling for spline models. In *Proceedings of Graphics Interface*, pages 142–150, Toronto, Canada, 1996.

[KML95] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large scale NURBS models. In *Symposium on Interactive 3D Graphics*, pages 51–58, Monterey, CA, 1995.

[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*, 21(1):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 *Proceedings of Graphics Interface*, pages 232–241, 1992.

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

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

[O'N66] B. O'Neill. *Elementary Differential Geometry*. Academic Press, 1966.

[RHD89] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *ACM Computer Graphics*, 23(3):107–117, 1989. (SIGGRAPH Proceedings).

[Roc87] A. Rockwood. A generalized scanning technique for display of parametrically defined surface. *IEEE Computer Graphics and Applications*, 7(8):15–26, 1987.

[SC88] M. Shantz and S. Chang. Rendering trimmed NURBS with adaptive forward differencing. *ACM Computer Graphics*, 22(4):189–198, 1988. (SIGGRAPH Proceedings).

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

[SL87] M. Shantz and S. Lien. Shading bicubic patches. *ACM Computer Graphics*, 21(4):189–196, 1987. (SIGGRAPH Proceedings).

[Usp48] J.V. Uspensky. *Theory of Equations*. McGraw Hill, New York, 1948.

.1 Gauss Maps

The derivatives of a Bézier patch, $\mathbf{F}$, with respect to u and v, respectively $\mathbf{F}_u$ and $\mathbf{F}_v$, both lie in the tangent-plane at $\mathbf{F}(u,v)$. Thus for each u and v, the normal direction is given by

$$\mathbf{N}(u,v) = \mathbf{F}_u \times \mathbf{F}_v$$

Bézier patches belong to the class of surfaces called orientable surfaces: their normals can be oriented 'inside' or 'outside' the surface [O'N66]. For a given model, we can orient all patches by flipping the order of control points such that $\mathbf{N}(u,v)$ points outside for each u, v for each patch.

> The Gauss map $\mathbf{G}$ of a surface, $\mathbf{F}$, is a map $\mathbf{G} : \mathbf{F} \rightarrow S^2$, the 2-Sphere, which takes point $\mathbf{F}(u,v)$ into the translation of the vector $\mathbf{U}(u,v)$ to the origin, where $\mathbf{U}(u,v)$ is the unit vector in the direction of $\mathbf{N}(u,v)$ [O'N66].

Thus the function $\mathbf{G}(u,v)$ can be used to compute the unit normal of the surface at the point (u,v). This can be relatively expensive to compute. We just use a pseudo-Gauss map, which gives the normal direction for each (u,v). The pseudo map has the benefit of being a Bézier surface itself, and hence is specified by a mesh of control points.

If $\mathbf{F}$ has a polynomial representation, the pseudo-normal surface is a $(2m-1) \times (2n-1)$ Bézier patch. If $\mathbf{F}$ is rational, the degree of the cross-products is $4m \times 4n$. 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:

$$\mathbf{N} = \frac{(\mathbf{f}_u W - \mathbf{f} W_u) \times (\mathbf{f}_v W - \mathbf{f} W_v)}{W^2}.$$

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

$$\mathbf{N} = \frac{\mathbf{f}_u \times \mathbf{f}_v W - \mathbf{f}_u \times \mathbf{f} W_v - \mathbf{f} W_u \times \mathbf{f}_v}{W^3}. \qquad (2)$$

Thus, the pseudo-normal surface is a $3m \times 3n$ rational Bézier surface and can be represented by a $(3m+1) \times (3n+1)$ mesh.

.2 Elimination theory

In our application we need to find common roots of polynomials. In particular we use the method of resultants [Usp48].

> The resultant of a set of polynomials is a function of the coefficients and variables of the equations which evaluates to zero iff the polynomials have a non-trivial common root.

In particular, we use the Sylvester's resultant: Given two polynomials

$$\begin{aligned} f(x) &= a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \\ g(x) &= b_m x^m + b_{m-1} x^{m-1} + \ldots + b_1 x + b_0 \end{aligned}$$

the Sylvester's resultant (assume $n > m$) is given as

$$R = \begin{vmatrix} a_n & a_{n-1} & . & . & . & a_0 & 0 & 0 & . & . & 0 \\ 0 & a_n & a_{n-1} & . & . & . & a_0 & 0 & . & . & 0 \\ & & & & & \vdots & & & & & \\ 0 & 0 & . & . & 0 & a_n & a_{n-1} & . & . & . & a_0 \\ 0 & 0 & . & . & 0 & 0 & b_m & b_{m-1} & . & . & b_0 \\ 0 & 0 & . & . & 0 & b_m & b_{m-1} & . & . & b_0 & 0 \\ & & & & & \vdots & & & & & \\ b_m & b_{m-1} & . & . & b_0 & 0 & 0 & . & . & . & 0 \end{vmatrix}$$

If f and g are each functions of two variables, say x and y, one variable, say x, is chosen to be primary and each entry of the determinant becomes an expression in y. We need to find the zeros of this expression. This is done using the following result:

Given a polynomial

$$f(x) = x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

its roots correspond to the generalized eigenvalues

$$\begin{pmatrix} 0 & 1 & 0 & 0 & . & . & 0 \\ 0 & 0 & 1 & 0 & . & . & 0 \\ & & & \vdots & & & \\ 0 & 0 & 0 & 0 & . & . & 1 \\ -a_0 & -a_1 & -a_2 & . & . & . & -a_{n-1} \end{pmatrix}$$

Good implementations of eigenvalue evaluators are available as part of numerical libraries like EISPACK and LAPACK. The resulting algorithms are fast, accurate, need no initial guess to the solutions and do not suffer from convergence problems [MD92].