Practical Ray Tracing of Trimmed NURBS Surfaces

William Martin **Elaine Cohen** **Russell Fish** **Peter Shirley**

Computer Science Department
50 S. Central Campus Drive
University of Utah
Salt Lake City, UT 84112
[wmartin | cohen | fish | shirley] @cs.utah.edu

Abstract

A system is presented for ray tracing trimmed NURBS surfaces. While approaches to components are drawn largely from existing literature, their combination within a single framework is novel. This paper also differs from prior work in that the details of an efficient implementation are fleshed out. Throughout, emphasis is placed on practical methods suitable to implementation in general ray tracing programs.

1 Introduction

The modeling community has embraced trimmed NURBS as a primitive of choice. The result has been a rapid proliferation in the number of models utilizing this representation. At the same time, ray tracing has become a popular method for generating computer graphics images of geometric models. Surprisingly, most ray tracing programs do not support the direct use of untessellated trimmed NURBS surfaces. The direct use of untessellated NURBS is desirable because tessellated models increase memory use which can be detrimental to runtime efficiency on modern architectures. In addition, tessellating models can result in visual artifacts, particularly in models with transparent components.

Although several methods of generating ray-NURBS intersections have appeared in the literature [3, 5, 8, 14, 17, 25, 27, 28], widespread adoption into ray tracing programs has not occurred. We believe this lack of acceptance stems from both the intrinsic algebraic complexity of these methods, and from the lack of emphasis in the literature on clean and efficient implementation. We present a new algorithm for ray-NURBS intersection that addresses these issues. The algorithm modifies approaches already in the literature to attain efficiency and ease of implementation.

Our approach is outlined in Figure 1. We create a set of boxes that bound the underlying surface over a given parametric range. The ray is tested for intersection with these boxes, and for a particular box that is hit, a parametric value within the box is used to initiate root finding. The key issues are determining which boxes to use, how to efficiently manage computing intersections with them, how to do the root finding, and how to efficiently evaluate the geometry for a given parametric value.

We use refinement to generate the bounding volume hierarchy, which results in a shallower tree depth than other subdivision-based methods. We also use an efficient refinement-based point evaluation method to speed root-finding. These choices turn

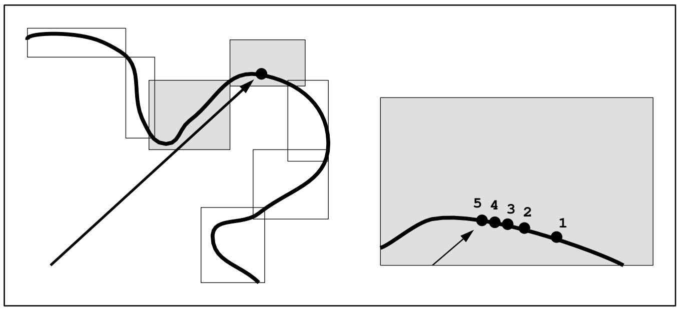


Figure 1: *The basic method we use to find the intersection of a ray and a parametric object shown in a 2D example. Left: The ray is tested against a series of axis-aligned bounding boxes. Right: For each box hit an initial guess is generated in the parametric interval the box bounds. Root finding is then iteratively applied until a convergence or a divergence criterion is met.*

out to be both reasonable to implement and efficient.

In Section 2 we present the bulk of our method, in particular how to create a hierarchy of bounding boxes and how to perform root finding within a single box to compute an intersection with an untrimmed NURBS surface. In Section 3 we describe how to extend the method to trimmed NURBS surfaces. Finally in Section 4 we show some results from our implementation of the algorithm.

2 Ray Tracing NURBS

In ray tracing a surface, we pose the question "At what points does a ray intersect the surface?" We define a ray as having an origin and a unit direction

$$\mathbf{r}(t) = \mathbf{o} + \hat{\mathbf{d}} * t.$$

A non-uniform rational B-spline (NURBS) surface can be formulated as

$$\mathbf{S}^w(u,v) = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \mathbf{P}^w_{i,j} B_{j,k_u}(u) B_{i,k_v}(v)$$

where the superscript w denotes that our formulation produces a point in rational four space, which must be normalized by the homogeneous coordinate prior to display. The $\{\mathbf{P}^w_{i,j}\}_{i=0,j=0}^{M-1,N-1}$ are the control points $(w_{i,j}x_{i,j}, w_{i,j}y_{i,j}, w_{i,j}z_{i,j}, w_{i,j})$ of the $M \times N$ mesh, having basis functions B_{j,k_u}, B_{i,k_v} of orders k_u and k_v defined over knot vectors

$$\tau_u = \{u_j\}_{j=0}^{N-1+k_u}$$

$$\tau_v = \{v_i\}_{i=0}^{M-1+k_v}$$

formulated as

$$B_{j,k_u}(u) \equiv \begin{cases} 1 & \text{if } k_u = 1 \text{ and } u \in [u_j, u_{j+1}) \\ 0 & \text{if } k_u = 1 \text{ and } u \notin [u_j, u_{j+1}) \\ \frac{u-u_j}{u_{j+k_u-1}-u_j} B_{j,k_u-1}(u) + \frac{u_{j+k_u}-u}{u_{j+k_u}-u_{j+1}} B_{j+1,k_u-1}(u) & \text{otherwise} \end{cases}$$

$$B_{i,k_v}(v) \equiv \begin{cases} 1 & \text{if } k_v = 1 \text{ and } v \in [v_i, v_{i+1}) \\ 0 & \text{if } k_v = 1 \text{ and } v \notin [v_i, v_{i+1}) \\ \frac{v-v_i}{v_{i+k_v-1}-v_i} B_{i,k_v-1}(v) + \frac{v_{i+k_v}-v}{v_{i+k_v}-v_{i+1}} B_{i+1,k_v-1}(v) & \text{otherwise.} \end{cases}$$

Such a surface $\mathbf{S}$ is defined over the domain $[u_{k_u-1}, u_N) \times [v_{k_v-1}, v_M)$. Each non-empty subinterval $[u_j, u_{j+1}) \times [v_i, v_{i+1})$ corresponds to a surface patch.

In this discussion, we assume that the reader has a basic familiarity with B-Splines. For further introduction, please refer to [2, 7, 10, 20].

Following the development by Kajiya [12], we rewrite the ray $\mathbf{r}$ as the intersection of two planes, $\{\mathbf{p} \mid \mathbf{P_1} \cdot (\mathbf{p}, 1) = 0\}$ and $\{\mathbf{p} \mid \mathbf{P_2} \cdot (\mathbf{p}, 1) = 0\}$, where $\mathbf{P_1} = (\mathbf{N_1}, d_1)$ and $\mathbf{P_2} = (\mathbf{N_2}, d_2)$. The normal to the first plane is defined as

$$\mathbf{N_1} = \begin{cases} (\hat{\mathbf{d}}_y, -\hat{\mathbf{d}}_x, 0) & \text{if } |\hat{\mathbf{d}}_x| > |\hat{\mathbf{d}}_y| \text{ and } |\hat{\mathbf{d}}_x| > |\hat{\mathbf{d}}_z| \\ (0, \hat{\mathbf{d}}_z, -\hat{\mathbf{d}}_y) & \text{otherwise.} \end{cases}$$

Thus, $\mathbf{N_1}$ is always perpendicular to the ray direction $\hat{\mathbf{d}}$, as desired. $\mathbf{N_2}$ is simply

$$\mathbf{N_2} = \mathbf{N_1} \times \hat{\mathbf{d}}.$$

Since both planes contain the origin $\mathbf{o}$, it must be the case that $\mathbf{P_1} \cdot (\mathbf{o}, 1) = \mathbf{P_2} \cdot (\mathbf{o}, 1) = 0$. Thus,

$$\begin{aligned} d_1 &= -\mathbf{N_1} \cdot \mathbf{o} \\ d_2 &= -\mathbf{N_2} \cdot \mathbf{o}. \end{aligned}$$

An intersection point on the surface $\mathbf{S}$ must satisfy the conditions

$$\begin{aligned} \mathbf{P_1} \cdot (\mathbf{S}(u,v), 1) &= 0 \\ \mathbf{P_2} \cdot (\mathbf{S}(u,v), 1) &= 0. \end{aligned}$$

The resulting implicit equations can be solved for u and v using numerical methods.

Ray tracing a NURBS surface proceeds in a series of steps. As a preprocess, the control mesh is flattened using refinement. There are several reasons for this. For Newton to converge quadratically, our initial guess for the root (u_*, v_*) must be close. By refining the mesh, we can bound the various sub-patches, and use the bounding volume hierarchy (BVH) both to cull the rays, and also to narrow the prospective parametric domain and so yield a good initial root estimate. It is important to note that the refined mesh does not persist in memory. It is used to generate the BVH and then is discarded.

During the intersection process, if we reach a leaf of the BVH, we apply traditional numerical root finding to the implicit equations above. The result will determine either a single (u_*, v_*) value or that no root exists.

In the sections that follow, we discuss the details of flattening, generating the BVH, root finding, evaluation, and partial refinement. Together these are all that is needed for computing ray-NURBS intersections.

2.1 Flattening

For Newton to converge both swiftly and reliably, the initial guess must be suitably close to the actual root. We employ a flattening procedure — i.e., refining/subdividing the control mesh so that each span meets some flatness criteria — both to ensure that the initial guess is a good one, and for the purpose of generating a bounding volume hierarchy for ray culling.

A wealth of research exists on polygonization of spline surfaces, e.g. [1, 13, 19, 22], and for the most part, these approaches can be readily applied to the problem of spline flattening. Some differences merit discussion. First, in the case of finding the numerical ray-spline intersection, we are not so much interested in flatness as in guaranteeing that there are not multiple roots within a leaf node of the bounding volume hierarchy. We note that this guarantee cannot always be made, particularly for nodes which contain silhouettes according to the ray source. Fortunately, the convergence problems which these boundary cases entail can also be improved with the mesh flattening we prescribe. We would also like to avoid any local maxima and minima that would serve to delay or worse yet prevent the convergence of our scheme. The flatness testing utilized by tessellation routines can be used to prevent these situations.

As ray tracing splines is at the outset a complicated task, we recommend the application of as simple a flattening procedure as possible. We have examined two flattening techniques in detail. The first of these is an adaptive subdivision scheme given by Peterson in [19]. As the source for the *Graphics Gems* is publicly available we will not discuss that method here, but instead refer the reader to the source.

The second approach we have considered is curvature-based refinement of the knot vectors. The number of knots to add to a knot interval is based on a simple heuristic which we now describe.

Suppose we have a B-spline curve $\mathbf{c}(t)$. An oracle for determining the extent to which the span $[t_i, t_{i+1})$ should be refined is given by the product of its maximum curvature and its length over that span. Long curve segments should be divided in order to ensure that the initial guess for the numerical solver is reasonably close to the actual root. High curvature regions should be split to avoid multiple roots. As we are utilizing maximum curvature as a measure of flatness, our heuristic will be overly conservative for curves other than circles. The heuristic value for the number of knots to add to the current span is given by

$$n_1 = C_1 * \max_{[t_i, t_{i+1})} \{\text{curvature}(\mathbf{c}(t))\} * \text{arclen}(\mathbf{c}(t))_{[t_i, t_{i+1})}.$$

We also choose to bound the deviation of the curve from its linear approximation. This notion will imbue our heuristic with scale dependence. Thus, for example, large circles will be broken into more pieces than small circles. Erring again on the conservative side, suppose our curve span is a circle with radius r, which we are approximating with linear segments. A measure of the accuracy of the approximation can be phrased in terms of a chord height h which gives the maximum deviation of the facets from the circle. Observing Figure 2, it can be seen that

$$\begin{aligned} h &= r - d \\ &= r - r\cos\frac{\theta}{2} \\ &\approx r(1 - (1 - \frac{\theta^2}{8})) \\ &= \frac{r\theta^2}{8}. \end{aligned}$$

The number of segments n_2 required to produce a curve within this tolerance is computed by $2\pi/\theta$. Thus, we have

$$n_2 = \frac{2\pi}{\sqrt{\frac{8h}{r}}}$$

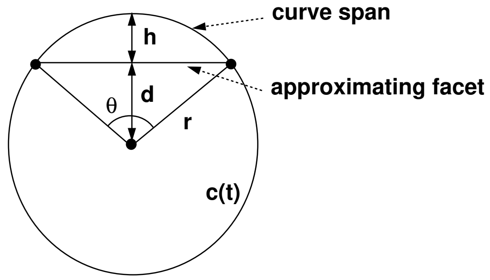


Figure 2: Illustration of the chord height tolerance heuristic.

$$
\begin{aligned}
&= \frac{2\pi\sqrt{r}}{\sqrt{8h}} \\
&\approx C_2 * \sqrt{\operatorname{arclen}(\mathbf{c}(t))_{[t_i,t_{i+1})}}.
\end{aligned}
$$

Combining the preceding oracles for curve behavior, our heuristic for the number of knots n to add to an interval will be $n_1 * n_2$:

$$n = C * \max_{[t_i,t_{i+1})}\{\operatorname{curvature}(\mathbf{c}(t))\} * \operatorname{arclen}(\mathbf{c}(t))^{3/2}_{[t_i,t_{i+1})}$$

where C allows the user to control the fineness. Since the maximum curvature and the arc length are in general hard to come by, we will estimate their values. The arc length of $\mathbf{c}$ over the interval is given by

$$\int_{t_i}^{t_{i+1}} |\mathbf{c}'(t)|dt = \operatorname{avg}_{[t_i,t_{i+1})}\{|\mathbf{c}'(t)|\} * (t_{i+1} - t_i).$$

Curvature is defined as

$$
\begin{aligned}
\operatorname{curvature}(\mathbf{c}(t)) &= \frac{|\mathbf{c}''(t) \times \mathbf{c}'(t)|}{|\mathbf{c}'(t)|^3} \\
&= \frac{|\mathbf{c}''(t)||\mathbf{c}'(t)||\sin\theta|}{|\mathbf{c}'(t)|^3} \\
&= \frac{|\mathbf{c}''(t)||\sin\theta|}{|\mathbf{c}'(t)|^2} \\
&\leq \frac{|\mathbf{c}''(t)|}{|\mathbf{c}'(t)|^2}
\end{aligned}
$$

We make the simplification $\operatorname{curvature}(\mathbf{c}(t)) \approx \frac{|\mathbf{c}''(t)|}{|\mathbf{c}'(t)|^2}$. In general, this estimate of the curvature will be overstated. The error will be on the side of refining too finely rather than not finely enough, so it is an acceptable trade-off to get the speed of computing second derivatives instead of curvature.

We are interested in the maximum curvature over the interval $[t_i, t_{i+1})$

$$\max_{[t_i,t_{i+1})}\{\operatorname{curvature}(\mathbf{c}(t)\} \approx \frac{\max_{[t_i,t_{i+1})}\{|\mathbf{c}''(t)|\}}{\operatorname{avg}_{[t_i,t_{i+1})}\{|\mathbf{c}'(t)|\}^2}.$$

If we assume the curve is polynomial, then the first derivative restricted to the interval $[t_i, t_{i+1})$ is given by

$$\mathbf{c}'(t) = \sum_{j=i-k+2}^{i} \frac{(k-1)(\mathbf{P_j} - \mathbf{P_{j-1}})}{t_{j+k-1} - t_j} B_{j,k-1}(t)$$

where $\{\mathbf{P_j}\}$ are the control points of the curve. The derivative of a rational curve is considerably more complicated. While the polynomial formulation of the derivative is in general a poor approximation to the rational derivative, in the case of flattening we have obtained reasonable results when applying the former to rational curves. For the remainder of this section we shall assume that rational control points have been projected into Euclidean 3-space.

Since $\sum B_{j,k-1}(t) = 1$, we can approximate the average velocity by averaging the control points $\mathbf{V_j}$ of the derivative curve:

$$\text{avg}_{[t_i,t_{i+1})}\{\mathbf{c}'(t)\} \approx \frac{1}{k-1}\sum_{j=i-k+2}^{i}(k-1)\frac{(\mathbf{P_j}-\mathbf{P_{j-1}})}{t_{j+k-1}-t_j} = \frac{1}{k-1}\sum_{j=i-k+2}^{i}\mathbf{V_j}$$

where $\mathbf{V_j} = (k-1)\frac{(\mathbf{P_j}-\mathbf{P_{j-1}})}{t_{j+k-1}-t_j}$. The average speed is therefore

$$\text{avg}_{[t_i,t_{i+1})}\{|\mathbf{c}'(t)|\} \approx \frac{1}{k-1}\sum_{j=i-k+2}^{i}|\mathbf{V_j}|.$$

The second derivative over $[t_i, t_{i+1})$ is given by

$$\mathbf{c}''(t) = \sum_{j=i-k+3}^{i}(k-2)\frac{[\mathbf{V_j}-\mathbf{V_{j-1}}]}{t_{j+k-2}-t_j}B_{j,k-2}(t) = \sum_{j=i-k+3}^{i}\mathbf{A_j}B_{j,k-2}(t)$$

where $\mathbf{A_j} = (k-2)\frac{[\mathbf{V_j}-\mathbf{V_{j-1}}]}{t_{j+k-2}-t_j}$. Using the convex hull property again, the maximum magnitude of the second derivative is approximated by the maximum magnitude of the second derivative curve control points $\mathbf{A_j}$:

$$\max_{[t_i,t_{i+1})}\{|\mathbf{c}''(t)|\} \approx \max_{i-k+3\le j\le i}\{|\mathbf{A_j}|\}.$$

Our heuristic is finally:

$$\begin{aligned}
n &= C * \frac{\max_{[t_i,t_{i+1})}\{|\mathbf{c}''(t)|\} * [\text{avg}_{[t_i,t_{i+1})}\{|\mathbf{c}'(t)|\} * (t_{i+1}-t_i)]^{3/2}}{\text{avg}_{[t_i,t_{i+1})}\{|\mathbf{c}'(t)|\}^2} \\
&= C * \frac{\max_{[t_i,t_{i+1})}\{|\mathbf{c}''(t)|\} * (t_{i+1}-t_i)]^{3/2}}{\text{avg}_{[t_i,t_{i+1})}\{|\mathbf{c}'(t)|\}^{1/2}} \\
&= C * \frac{\max_{i-k+3\le j\le i}\{|\mathbf{A_j}|\}(t_{i+1}-t_i)^{3/2}}{(\frac{1}{k-1}\sum_{j=i-k+2}^{i}|\mathbf{V_j}|)^{1/2}}.
\end{aligned}$$

For each row of the mesh, we apply the above heuristic to calculate how many knots need to be added to each u knot interval, the final number being the maximum across all rows. This process is repeated for each column in order to refine the v knot vector. The inserted knots are spaced uniformly within the existing knot intervals.

As a final step in the flattening routine, we "close" all of the knot intervals in the refined knot vectors $\mathbf{t_u}$ and $\mathbf{t_v}$. By this, we mean that we give multiplicity $k_u - 1$ to each internal knot of $\mathbf{t_u}$ and multiplicity k_u to each end knot. Similarly, we give multiplicities of $k_v - 1$ and k_v to the internal and external knots, respectively, of $\mathbf{t_v}$. The result is a Bézier surface patch corresponding to each non-empty interval $[u_i, u_{i+1}) \times [v_j, v_{j+1})$ of $\mathbf{t_u} \times \mathbf{t_v}$, which we can bound using the convex hull of the corresponding refined surface mesh points. This becomes critical in the next section.

The refined knot vectors determine the refinement matrix used to transform the existing mesh into the refined mesh. There are many techniques for generating this "alpha matrix." As it is not critical that this be fast, we refer the reader to several sources [4, 6, 7, 16].

Both adaptive subdivision and curvature-based refinement should yield acceptable results. Both allow the user to adjust the resulting flatness via a simple intuitive parameter. We have preferred the latter mainly because it produces its result in a single pass, without the creation of unnecessary intermediate points. Adaptive subdivision does have the advantage of inserting one knot value at a time, so one does not necessarily need to implement the full machinery of the Oslo algorithm [6]. Instead, one can opt for a simpler approach, such as that of Boehm [4]. It is not clear which method produces more optimal meshes in general. On the one hand, adaptive subdivision computes intermediate results which it then inspects to determine where additional subdivision is required. On the other hand, our method utilizes refinement (we only subdivide in the last step), and this converges more swiftly to the underlying surface than does subdivision.

Neither technique is entirely satisfactory. Each considers the various parametric directions independently, while subdivision and refinement clearly impact both directions. The curvature-based refinement method refines a knot interval without considering the impact of that refinement on neighboring intervals. This can lead to unnecessary refinement. Neither makes any attempt to find optimal placement of inserted knots.

The adaptive subdivision and curvature-based refinement methods are the products of the inevitable compromise between refinement speed and quality. Both satisfy the efficiency and accuracy demands of the problem at hand.

A point which we do not wish to sweep under the carpet is that the selection of the flatness parameter is empirical and left to the user. As this parameter directly impacts the convergence of the root finding process, it should be carefully chosen. Choosing a value too small may cause the numerical solver to fail to converge or to converge to one of several roots in the given parametric interval. This effect will probably be most noticeable along silhouette edges and patch boundaries. On the other hand, choosing a value too large will result in over-refinement of the surface leading to a deeper bounding volume hierarchy, and therefore, potentially more time per ray. We have found that after some experimentation, one develops an intuition for the sorts of parameters which work for a surface. For an example of a system which guarantees convergence without user intervention, see Toth [27]. This guarantee is made at the price of linear convergence in the root finding procedure.

2.2 Bounding Volume Hierarchy

We build a bounding volume hierarchy using the points of the refined control mesh we found in the previous section. The root and internal nodes of the tree will contain simple primitives which bound portions of the underlying surface. The leaves of the tree are special objects, which we call *interval objects*, are used to provide an initial guess (in our case, the midpoint of the bracketing parametric interval) to the Newton iteration. We will now examine the specifics in more detail.

The convex hull property of B-spline surfaces guarantees that the surface is contained in the convex hull of its control mesh. As a result, any convex objects which bound the mesh will bound the underlying surface. We can actually make a stronger claim; because we closed the knot intervals in the last section [made the multiplicity of the internal knots $k-1$], each non-empty interval $[u_i, u_{i+1}) \times [v_j, v_{j+1})$ corresponds to a surface patch which is completely contained in the convex hull of its corresponding mesh points. Thus, if we produce bounding volumes for each of these intervals, we will have completely enclosed the surface. We form the tree by sorting the volumes according to the axis direction which has greatest extent across the bounding volumes, splitting the data in half, and repeating the process.

There remains the dilemma of which primitive to use as a bounding volume. Many different objects have been tried including spheres [8], axis-aligned boxes [8, 25, 28], oriented boxes [8], and parallelepipeds [3]. There is generally a tradeoff between speed of intersection and tightness of fit. The analysis is further complicated by the fact that bounding volume performance depends on the type of scene being rendered.

We have preferred simplicity, narrowing our choice to spheres and axis-aligned boxes. Spheres have a very fast intersection test. However, spheres, by definition, can never be flat. Since our intersection routines require surfaces which are locally "flat," spheres did not seem to be a natural choice.

Axis-aligned boxes have many advantages. First, they can become flat [at least along axis directions], so they can provide a tighter fit than spheres. The union of two axis-aligned boxes is easily computed. This computation is necessary when building the BVH from the leaves. With many other bounding volumes, the leaves of the subtree must be examined individually to produce reasonable bounding volumes. Finally, many scenes are axis-aligned, especially in the case of architectural walkthroughs. Axis-aligned boxes are nearly ideal in this circumstance.

A simple ray-box intersection routine is intuitive, and so we omit its discussion. An optimized version can be found in the paper by Smits [24].

2.3 Root Finding

Given a ray as the intersection of planes $\mathbf{P_1} = (\mathbf{N_1}, d_1)$ and $\mathbf{P_2} = (\mathbf{N_2}, d_2)$, our task is to solve for the roots (u_*, v_*) of

$$\mathbf{F}(u,v) = \begin{pmatrix} \mathbf{N_1} \cdot \mathbf{S}(u,v) + d_1 \\ \mathbf{N_2} \cdot \mathbf{S}(u,v) + d_2 \end{pmatrix}.$$

A variety of numerical methods can be applied to the problem. An excellent reference for these techniques is [21, pp 347-393]. We use Newton's method for several reasons. First, it converges quadratically if the initial guess is close, which we ensure by constructing a bounding volume hierarchy. Furthermore, the surface derivatives exist and are calculated at cost comparable to that of surface evaluation. This means that there is likely little computational advantage to utilizing approximate derivative methods such as Broyden.

Newton's method is built from a truncated Taylor's series. Our iteration takes the form

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} u_n \\ v_n \end{pmatrix} - \mathbf{J}^{-1}(u_n, v_n) * \mathbf{F}(u_n, v_n)$$

where $\mathbf{J}$ is the Jacobian matrix of $\mathbf{F}$, defined as

$$\mathbf{J} = (\mathbf{F_u}, \mathbf{F_v}).$$

$\mathbf{F_u}$ and $\mathbf{F_v}$ are the vectors

$$\begin{aligned} \mathbf{F_u} &= \begin{pmatrix} \mathbf{N_1} \cdot \mathbf{S_u}(u,v) \\ \mathbf{N_2} \cdot \mathbf{S_u}(u,v) \end{pmatrix} \\ \mathbf{F_v} &= \begin{pmatrix} \mathbf{N_1} \cdot \mathbf{S_v}(u,v) \\ \mathbf{N_2} \cdot \mathbf{S_v}(u,v) \end{pmatrix}. \end{aligned}$$

The inverse of the Jacobian is calculated using a result from linear algebra:

$$\mathbf{J}^{-1} = \frac{\mathrm{adj}(\mathbf{J})}{\det(\mathbf{J})}.$$

The adjoint $\mathrm{adj}(\mathbf{J})$ is equal to the transpose of the cofactor matrix

$$\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where $\mathbf{C}_{ij} = (-1)^{i+j}\det(\mathbf{J}_{ij})$ and $\mathbf{J}_{ij}$ is the submatrix of $\mathbf{J}$ which remains when the ith row and jth column are removed. We find that

$$\mathrm{adj}(\mathbf{J}) = \begin{pmatrix} \mathbf{J}_{22} & -\mathbf{J}_{12} \\ -\mathbf{J}_{21} & \mathbf{J}_{11} \end{pmatrix}.$$

We use four criteria, drawn from Yang [28], to decide when to terminate the Newton iteration. The first condition is our success criterion: if we are closer to the root than some predetermined ϵ

$$||\mathbf{F}(u_n, v_n)|| < \epsilon$$

then we report a hit. Otherwise, we continue the iteration. The other three criteria are failure criteria, meaning that if they are met, we terminate the iteration and report a miss. We do not allow the new (u_*, v_*) estimate to take us farther from the root than the previous one:

$$||\mathbf{F}(u_{n+1}, v_{n+1})|| > ||\mathbf{F}(u_n, v_n)||.$$

We also do not allow the iteration to take us outside the parametric domain of the surface:

$$u \notin [u_{k_u-1}, u_N), v \notin [v_{k_v-1}, v_M).$$

We limit the number of iterations allowed for convergence:

$$iter > MAXITER.$$

We set $MAXITER$ around 7, but the average number of iterations needed to produce convergence is 2 or 3 in practice.

A final check is made to assure that the Jacobian $\mathbf{J}$ is not singular. While this would seem to be a rare occurrence in theory, we have encountered this problem in practice. In the situation where $\mathbf{J}(u_k, v_k)$ is singular, either the surface is not regular ($\mathbf{S_u} \times \mathbf{S_v} = 0$) or the ray is parallel to a silhouette ray at the point $\mathbf{S}(u_k, v_k)$. (A proof of this assertion is available at the JGT paper website.) In either situation, to determine singularity, we test

$$|\det(\mathbf{J})| < \epsilon.$$

If the Jacobian is singular we perform a jittered perturbation of the parametric evaluation point,

$$\begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} = \begin{pmatrix} u_k \\ v_k \end{pmatrix} + .1 * \begin{pmatrix} \mathrm{drand48}() * (u_0 - u_k) \\ \mathrm{drand48}() * (v_0 - v_k) \end{pmatrix}$$

and initiate the next iteration. This operation tends to push the iteration away from problem regions without leaving the basin of convergence.

Figure 3: Failure to adjust tolerances may result in surface acne.

Because any root (u_*, v_*) produced by the Newton iteration is approximate, it will almost definitely not lie along the ray $\mathbf{r} = \mathbf{o} + t * \hat{\mathbf{d}}$. In order to obtain the closest point along $\mathbf{r}$ to the approximate root (u_*, v_*) we perform a projection

$$t = (\mathbf{P} - \mathbf{o}) \cdot \hat{\mathbf{d}}.$$

The approximate nature of the convergence also impacts other parts of the ray tracing system. Often, a tolerance ϵ is defined to determine the minimum distance a ray can travel before reporting an intersection. This prevents self-intersections due to errors in numerical calculation. The potential for error is larger in the case of numerical spline intersection than, say, ray-polygon intersection. Thus, the tolerances will need to be adjusted accordingly. Failure to make this adjustment will result in "surface acne" [9] (see Figure 3).

An enhanced method for abating acne would test the normal at points less than ϵ along the ray to determine whether these points were on the originating surface. Unfortunately, we have found that we cannot rely on modeling programs to produce consistently oriented surfaces. Therefore, our system utilizes the coarser ϵ condition above.

2.4 Evaluation

The Newton iteration requires us to perform surface and derivative evaluation at a point (u, v) on the surface. In this section we examine how this can be accomplished efficiently. Prior work on efficient evaluation can be found in [15, 16].

We begin by examining the problem in the context of B-spline curves. We then generalize the result to surfaces. The development parallels that found in [26].

We evaluate a curve $\mathbf{c}(t)$ by using refinement to stack $k-1$ knots (where k is the order of the curve) at the desired parameter value t_*. The refined curve is defined over a new knot vector $\mathbf{t}$ with basis functions $N_{i,k}(t)$ and new control points $\omega_i \mathbf{D}_i$.

Recall the recurrence for the B-Spline basis functions:

$$N_{i,k}(t) = \begin{cases} 1 & \text{if } k = 1 \text{ and } t \in [t_i, t_{i+1}) \\ 0 & \text{if } k = 1 \text{ and } t \notin [t_i, t_{i+1}) \\ \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t) & \text{otherwise.} \end{cases}$$

Let $t_* \in [t_\mu, t_{\mu+1})$. As a result of refinement, $t_* = t_\mu = \ldots = t_{\mu-k+2}$. According to the definition of the basis functions, $N_{\mu,1}(t_*) = 1$. There are only two potentially non-zero basis functions of order $k = 2$, namely those dependent on $N_{\mu,1}$: $N_{\mu,2}$ and $N_{\mu-1,2}$. From the recurrence,

$$\begin{aligned} N_{\mu,2}(t_*) &= \frac{t_* - t_\mu}{t_{\mu+k-1} - t_\mu} N_{\mu,1}(t_*) + \frac{t_{i+k} - t_*}{t_{\mu+k} - t_{\mu+1}} N_{\mu+1,1}(t_*) \\ &= 0 * 1 + \frac{t_{i+k} - t_*}{t_{\mu+k} - t_{\mu+1}} * 0 \\ &= 0 \end{aligned}$$

and,

$$\begin{aligned} N_{\mu-1,2}(t_*) &= \frac{t_* - t_{\mu-1}}{t_{\mu+k-2} - t_{\mu-1}} N_{\mu-1,1}(t_*) + \frac{t_{\mu+k-1} - t_*}{t_{\mu+k-1} - t_\mu} N_{\mu,1}(t_*) \\ &= 0 * 0 + 1 * 1 \\ &= 1. \end{aligned}$$

Likewise, the only non-zero order $k = 3$ terms will be those dependent on $N_{\mu-1,2}$: $N_{\mu-1,3}$ and $N_{\mu-2,3}$.

$$
\begin{aligned}
N_{\mu-1,3}(t_*) &= \frac{t_* - t_{\mu-1}}{(\cdots)} * 1 + (\cdots) * 0 \\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
N_{\mu-2,3}(t_*) &= (\cdots) * 0 + \frac{t_{\mu-2+k} - t_*}{t_{\mu-2+k} - t_{\mu-1}} * 1 \\
&= 1
\end{aligned}
$$

The pattern that emerges is that $N_{\mu-k+1,k}(t_*) = 1$. A straightforward consequence of this result is

$$\mathbf{c}(t_*) = \frac{\sum_i N_{i,k}(t_*)\omega_i \mathbf{D_i}}{\sum_i N_{i,k}(t_*)\omega_i} = \frac{\omega_{\mu-k+1}\mathbf{D}_{\mu-\mathbf{k+1}}}{\omega_{\mu-k+1}} = \mathbf{D}_{\mu-\mathbf{k+1}}.$$

The point with index $\mu - k + 1$ in the refined control polygon yields the point on the curve.

A further analysis can be used to yield the derivative. Given a rational curve

$$\mathbf{c}(t) = \frac{\sum_i N_{i,k}(t)\omega_i \mathbf{D_i}}{\sum_i N_{i,k}(t)\omega_i} = \frac{\sum_i N_{i,k}(t)\mathbf{D_i^\omega}}{\sum_i N_{i,k}(t)\omega_i} = \frac{\mathbf{D}(t)}{\omega(t)},$$

where $\mathbf{D_i^\omega} = \omega_i \mathbf{D_i}$, the derivative is given by the quotient rule

$$\mathbf{c}'(t) = \frac{\omega(t)(\mathbf{D}^\omega)'(t) - \mathbf{D}^\omega(t)\omega'(t)}{\omega(t)^2}.$$

By the preceding analysis $\mathbf{D}^\omega(t_*) = \mathbf{D}^\omega_{\mu-\mathbf{k+1}}$. Likewise, $\omega(t_*) = \omega_{\mu-k+1}$. The derivative of the B-Spline basis function is given by

$$N'_{i,k}(t) = (k-1)\left[\frac{N_{i,k-1}(t)}{t_{i+k-1} - t_i} - \frac{N_{i+1,k-1}(t)}{t_{i+k} - t_{i+1}}\right].$$

Evaluating the derivative at t_*, we have

$$
\begin{aligned}
(\mathbf{D}^\omega)'(t_*) &= \sum_i N'_{i,k}(t_*)\mathbf{D_i^\omega} \\
&= (k-1)\sum_i \mathbf{D_i^\omega}\left(\frac{N_{i,k-1}(t_*)}{t_{i+k-1} - t_i} - \frac{N_{i+1,k-1}(t_*)}{t_{i+k} - t_{i+1}}\right) \\
&= (k-1)\left[\sum_i \mathbf{D_i^\omega}\frac{N_{i,k-1}(t_*)}{t_{i+k-1} - t_i} - \sum_i \mathbf{D_i^\omega}\frac{N_{i+1,k-1}(t_*)}{t_{i+k} - t_{i+1}}\right].
\end{aligned}
$$

We know the only non-zero basis function of order $k - 1$ is $N_{\mu-k+2,k-1}(t_*) = 1$. Therefore,

$$(\mathbf{D}^\omega)'(t_*) = (k-1)\left[\frac{\mathbf{D}^\omega_{\mu-\mathbf{k+2}}}{t_{\mu+1} - t_{\mu-k+2}} - \frac{\mathbf{D}^\omega_{\mu-\mathbf{k+1}}}{t_{\mu+1} - t_{\mu-k+2}}\right]. \tag{1}$$

Analogously,

$$\omega'(t_*) = (k-1)\left[\frac{\omega_{\mu-k+2}}{t_{\mu+1} - t_{\mu-k+2}} - \frac{\omega_{\mu-k+1}}{t_{\mu+1} - t_{\mu-k+2}}\right].$$

Plugging in for $\mathbf{c}'(t)$

$$
\begin{aligned}
\mathbf{c}'(t_*) &= (k-1)\left[\frac{\frac{\mathbf{D}^\omega_{\mu-\mathbf{k+2}} - \mathbf{D}^\omega_{\mu-\mathbf{k+1}}}{t_{\mu+1} - t_{\mu-k+2}}\omega_{\mu-k+1} - \mathbf{D}^\omega_{\mu-\mathbf{k+1}}\frac{\omega_{\mu-k+2} - \omega_{\mu-k+1}}{t_{\mu+1} - t_{\mu-k+2}}}{\omega^2_{\mu-k+1}}\right] \\
&= \frac{k-1}{(t_{\mu+1} - t_{\mu-k+2})\omega_{\mu-k+1}}\left[\mathbf{D}^\omega_{\mu-\mathbf{k+2}} - \mathbf{D}^\omega_{\mu-\mathbf{k+1}}\frac{\omega_{\mu-k+2}}{\omega_{\mu-k+1}}\right] \\
&= \frac{k-1}{(t_{\mu+1} - t_{\mu-k+2})\omega_{\mu-k+1}}\left[\omega_{\mu-k+2}\mathbf{D}_{\mu-\mathbf{k+2}} - \omega_{\mu-k+1}\mathbf{D}_{\mu-\mathbf{k+1}}\frac{\omega_{\mu-k+2}}{\omega_{\mu-k+1}}\right] \\
&= \frac{(k-1)\omega_{\mu-k+2}}{(t_{\mu+1} - t_*)\omega_{\mu-k+1}}[\mathbf{D}_{\mu-\mathbf{k+2}} - \mathbf{D}_{\mu-\mathbf{k+1}}].
\end{aligned}
$$

The result for surface evaluation follows directly from the curve derivation, due to the independence of the parameters in the tensor product, so we shall simply state the results:

$$
\begin{aligned}
\mathbf{S}(u_*, v_*) &= \mathbf{D}_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{1},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{1}} \\
\mathbf{S_u}(u_*, v_*) &= \frac{(k_u-1)\omega_{\mu_u-k_u+2,\mu_v-k_v+1}}{(u_{\mu_u+1}-u_*)\omega_{\mu_u-k_u+1,\mu_v-k_v+1}}[\mathbf{D}_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{2},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{1}} - \mathbf{D}_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{1},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{1}}] \\
\mathbf{S_v}(u_*, v_*) &= \frac{(k_v-1)\omega_{\mu_u-k_u+1,\mu_v-k_v+2}}{(v_{\mu_v+1}-v_*)\omega_{\mu_u-k_u+1,\mu_v-k_v+1}}[\mathbf{D}_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{1},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{2}} - \mathbf{D}_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{1},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{1}}].
\end{aligned}
$$

The normal $\mathbf{n}(u, v)$ is given by the cross product of the first order partials:

$$\mathbf{n}(u_*, v_*) = \mathbf{S_u}(u_*, v_*) \times \mathbf{S_v}(u_*, v_*).$$

If the surface is not regular (i.e., $\mathbf{S_u} \times \mathbf{S_v} = 0$), then our computation may erroneously generate a zero surface normal. We avoid these problem areas in our numerical solver by perturbing the parametric points (see Section 2.3).

2.5 Partial Refinement

We still need to explain how to calculate the points in the refined mesh so that we can evaluate surface points and derivatives. What follows is drawn directly from Lyche et al. [16], tailored to our specialized needs. We again formulate our solution in the context of curves, and then generalize the result to surfaces.

Earlier we proposed to evaluate the curve $\mathbf{c}$ at t_* by stacking $k-1$ t_*-valued knots in its knot vector τ to generate the refined knot vector $\mathbf{t}$. The B-Spline basis transformation defined by this refinement yields a matrix $\mathbf{A}$ which can be used to calculate the refined control polygon $\mathbf{D}^\omega$ from the original polygon $\mathbf{P^w}$:

$$\mathbf{D}^\omega = \mathbf{A}\mathbf{P^w}.$$

We are not interested in calculating the full alpha matrix $\mathbf{A}$, but merely rows $\mu - k + 2$ and $\mu - k + 1$, as these are used to generate the points $\mathbf{D}^\omega_{\mu-\mathbf{k}+\mathbf{2}}$ and $\mathbf{D}^\omega_{\mu-\mathbf{k}+\mathbf{1}}$ which are required for point and derivative evaluation.

Suppose $t_* \in [\tau_{\mu'}, \tau_{\mu'+1})$. We can generate the refinement for row $\mu + k - 1$ using a triangular scheme

$$
\begin{array}{ccc}
 & & \alpha'_{\mu',0} \\
 & \alpha'_{\mu'-1,1} & \alpha'_{\mu',1} \\
 & \vdots & \vdots \\
\alpha'_{\mu'-\nu,\nu} & \cdots & \alpha'_{\mu',\nu}
\end{array}
$$

where ν is the number of knots we are inserting and

$$
\begin{aligned}
\alpha'_{j,1} &= \delta_{j,\mu'} \\
\alpha'_{j,p+1} &= \gamma_{j,p}\alpha'_{j,p} + (1-\gamma_{j+1,p})\alpha'_{j+1,p} \\
\gamma_{j,p} &= \begin{cases} (t_* - \tau_{\mu'-p+j-(k-1-\nu)})/d, & \text{if } d = \tau_{\mu'+1+j} - \tau_{\mu'-p+j-(k-1-\nu)} > 0 \\ \text{arbitrary} & \text{otherwise.} \end{cases}
\end{aligned}
$$

$\mathbf{A}_{\mu-k+1,j} = \alpha'_{j,\nu}$ for $j = \mu' - \nu, \cdots, \mu'$ and $\mathbf{A}_{i,j} = 0$ otherwise. If n knots exist in the original knot vector τ with value t_*, then $\nu = \max\{k - 1 - n, 1\}$ – that is to say, we always insert at least 1 knot. The quantity ν is used in the triangular scheme above to allow one to skip those basis functions which are trivially 0 or 1 due to repeated knots. As a result of this triangular scheme, we generate basis functions in place and avoid redundant computation of α' values for subsequent levels.

The procedure of knot insertion we propose is analogous to Bézier subdivision. In Figure 4 a Bézier curve has been subdivided at $t = .5$, generating a refined polygon $\{\mathbf{p_i}\}$ from the original polygon $\{\mathbf{P_i}\}$. Recall that a Bézier curve is simply a B-Spline curve with open end conditions, in this case, with knot vector $\tau = \{0, 0, 0, 0, 1, 1, 1, 1\}$. The refined knot vector is then $\mathbf{t} = \{0, 0, 0, 0, .5, .5, .5, 1, 1, 1, 1\}$. According to our definitions, $\mu = 6$, $\mu' = 3$. Thus, the point on the surface should be indexed $\mu - k + 1 = 6 - 4 + 1 = 3$, which agrees with the figure. We observe that $\mathbf{p_3}$ is a convex blend of $\mathbf{p_2}$ and $\mathbf{p_4}$.

Likewise, in the refinement scheme we propose, the point on the curve $\mathbf{D}^\omega_{\mu-\mathbf{k}+\mathbf{1}}$ will be a convex blend of the points $\mathbf{D}^\omega_{\mu-\mathbf{k}}$ and $\mathbf{D}^\omega_{\mu-\mathbf{k}+\mathbf{2}}$. The blend factor will be $\gamma_{\mu',0}$. The dependency graph shown in Figure 5 will help to clarify. The factor $\gamma_{\mu',0}$ is introduced at the first level of the recurrence. The leaf terms can be written as

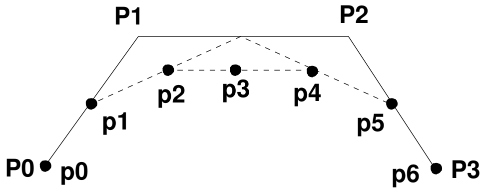


Figure 4: Original mesh and refined mesh which results from Bézier subdivision.

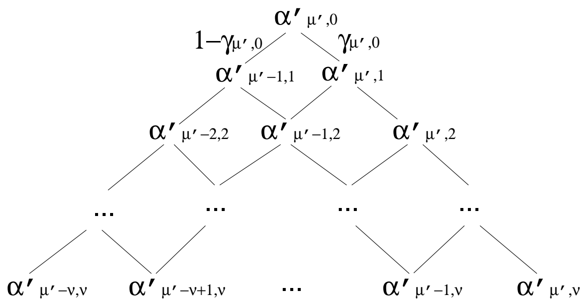


Figure 5: Graph showing how the factor $\gamma_{\mu',0}$ propagates through the recurrence.

$$\alpha'_{j,\nu} = (1 - \gamma_{\mu',0}) l_{j,\nu} + \gamma_{\mu',0} r_{j,\nu}$$

with $j = \mu' - \nu, \cdots, \mu'$. $\{l_{j,\nu}\}$ and $\{r_{j,\nu}\}$ are those terms dependent on $\alpha'_{\mu-1,1}$ and $\alpha'_{\mu,1}$ respectively. They are the elements of the alpha matrix rows $\mu - k$ and $\mu - k + 2$ with $\mathbf{A}_{\mu-k,j} = l_{j,\nu}$ and $\mathbf{A}_{\mu-k+2,j} = r_{j,\nu}$ for $j = \mu' - \nu, \cdots, \mu'$. We can generate the $\{l_{j,\nu}\}$ by setting $\alpha'_{\mu'-1,1} = 1$ and $\alpha'_{\mu',1} = 0$ and likewise, generate $\{r_{j,\nu}\}$ by setting $\alpha'_{\mu'-1,1} = 0$ and $\alpha'_{\mu',1} = 1$. Thus, $\mathbf{A}_{\mu-k,j}$ and $\mathbf{A}_{\mu-k+2,j}$ can be generated in the course of generating $\mathbf{A}_{\mu-k+1,j}$ at little additional expense.

The procedure above generalizes easily to surfaces, allowing us to generate the desired rows of the refinement matrices $\mathbf{A_u}$ and $\mathbf{A_v}$. The refined mesh $\mathbf{D}^\omega$ is derived from the existing mesh $\mathbf{P^w}$ by:

$$\mathbf{D}^\omega = \mathbf{A_u} \mathbf{P^w} \mathbf{A_v^T}.$$

To produce the desired points we only need to evaluate

$$\begin{pmatrix} \mathbf{D}^\omega_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{1},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{1}} & \mathbf{D}^\omega_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{1},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{2}} \\ \mathbf{D}^\omega_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{2},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{1}} & \mathbf{D}^\omega_{\mu_\mathbf{u}-\mathbf{k_u}+\mathbf{2},\mu_\mathbf{v}-\mathbf{k_v}+\mathbf{2}} \end{pmatrix} =$$
$$\begin{pmatrix} (\mathbf{A_u})_{\mu_u+k_u+1,[\mu'_u-\nu_u \ldots \mu'_u]} \\ (\mathbf{A_u})_{\mu_u+k_u+2,[\mu'_u-\nu_u \ldots \mu'_u]} \end{pmatrix} \mathbf{P^w}_{[\mu'_\mathbf{u}-\nu_\mathbf{u} \ldots \mu'_\mathbf{u}][\mu'_\mathbf{v}-\nu_\mathbf{v} \ldots \mu'_\mathbf{v}]} \begin{pmatrix} (\mathbf{A_v})_{\mu_v+k_v+1,[\mu'_v-\nu_v \ldots \mu'_v]} \\ (\mathbf{A_v})_{\mu_v+k_v+2,[\mu'_v-\nu_v \ldots \mu'_v]} \end{pmatrix}^T$$

This can be made quite efficient. We have been able to calculate approximately 150K surface evaluations (with derivative) per second on a 300MHz MIPS R12K using this approach.

3 Trimming Curves

Trimming curves are a common method for overcoming the topologically rectangular limitations of NURBS surfaces. They result typically when designers wish to remove sections from models which are not aligned with the underlying parameterization. In this section, we will define what we mean by trimming curves.

A trimming curve is a closed, oriented curve which lies on a NURBS surface. For our purposes, the curve will consist of piecewise linear segments in parametric space $\{\mathbf{p_i} = (u_i, v_i)\}$. (In principle, there is no reason one could not extend our hierarchical technique to higher order trimming curves. [17] deals with Bézier trims.) Often other data is available, such as the real-world coordinates over the various curve vertices, but we will not make use of this information. It is important to note that these curves are not necessarily convex.

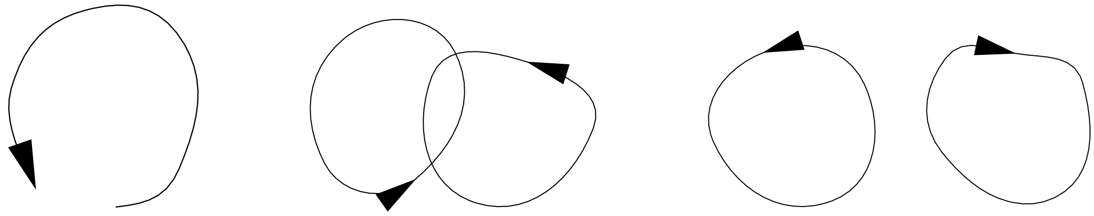


Figure 6: Invalid trimming curves: a curve which is not closed, curves which cross, and curves with conflicting orientation.

We calculate the orientation of the curve using the method of Rokne [23] for computing the area of a polygon. Given parametric points $\{\mathbf{p_i} = (u_i, v_i)\},\ i = 0 \ldots n$, the signed area can be computed by

$$A = \frac{1}{2}\sum_{i=0}^{n} u_i v_{(i+1) \bmod n} - u_{(i+1) \bmod n} v_i.$$

If A is negative, the curve has a clockwise orientation. Otherwise, the orientation is counter-clockwise.

The orientation of a trimming curve determines which region of the surface is to be kept. We use the convention that the part of the surface to be kept is on the right side of the curve [as you walk in the direction of its orientation]. Inconsistencies in orientation that would result in an ambiguous determination of whether to trim are not allowed (see Figure 6).

An important characteristic of the trimming curves we use is that they are not allowed to cross. Trimming curves can contain trimming curves, and can share vertices and edges. Areas inscribed by counter-clockwise curves are often termed "holes", while those inscribed by clockwise curves are termed "regions."

3.1 Building a Hierarchy

Given a set of trimming curves on a particular B-Spline surface, we can build a hierarchy based on containment (see Figure 7). Since the curves are not allowed to cross, there are only three possible relationships between two curves $\mathbf{c_1}$ and $\mathbf{c_2}$. $\mathbf{c_1}$ can contain $\mathbf{c_2}$, be contained in $\mathbf{c_2}$, or share no regions in common with $\mathbf{c_2}$. Each node in our hierarchy is a list of trims, and each trim can refer to yet another list of trims which fall inside of it. Building the hierarchy proceeds as:

Insert (Trim newtrim, TrimList tl)

for *each Trim t in TrimList tl* ***do***
  if *t contains newtrim* ***then***
    Insert (newtrim, t.trimlist)
    return
  else
    if *newtrim contains t* ***then***
      Insert(t,newtrim.trimlist)
      Remove(t,tl)
    end if
  end if
end for
tl.Add(newtrim)

The *contains* function for trims needs a bit of clarification. Since trims can share edges and vertices, proper containment tests – those that test only the vertices – will not always work. Instead we perform inside/outside tests on the midpoints of each trim segment. In comparing $\mathbf{c_1}$ and $\mathbf{c_2}$, $\mathbf{c_1}$ is judged to be contained in $\mathbf{c_2}$ if and only if the midpoint of some segment of $\mathbf{c_1}$ falls inside $\mathbf{c_2}$. Since curves cannot cross, any such midpoint will do. The inside/outside test is performed with regard to some ϵ so as to counteract round-off error.

For each trim curve, and for each trim list, we store a bounding box which we will use to speed culling in the following ray tracing step.

Once the trim hierarchy is created, we perform a quick pass through the surface patches, removing those patches which are completely trimmed away. This is an optimization step which reduces the size of the BVH and the number of patches which must be examined by the intersection routines. The procedure below can be used by encoding the parametric boundary of the patch to be tested as the trimming curve crv.

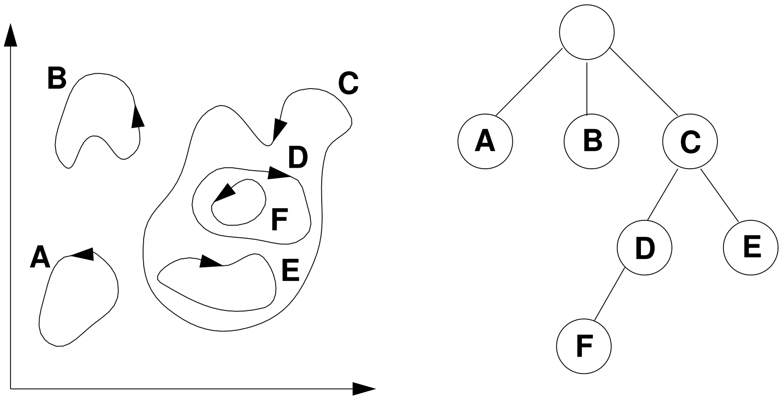


Figure 7: A set of trimming curves and the resulting hierarchy.

IsTrimmed(TrimList tl, Trim crv)

for *each Trim t in TrimList tl* ***do***
  if *t contains crv* ***then***
    return IsTrimmed(t.tl, crv)
  else
    if *t crosses crv* ***then***
      return false
    end if
  end if
end for
return !tl.is_clockwise

3.2 Ray Tracing Trimmed NURBS

We ray trace trimmed NURBS by first performing ray intersection with the untrimmed surface. If an intersection point (u_*, v_*) is found, we then look to the trim hierarchy to determine whether it is to be culled or returned as a hit.

Inside(Point p, TrimList tl, boolean& keep)

keep = !tl.is_clockwise
if *tl.boundingbox contains p* ***then***
  for *each Trim t in TrimList tl* ***do***
    if *t.boundingbox contains p* ***then***
      if *t contains p* ***then***
        Inside(p,t.tl,keep)
        return
      end if
    end if
  end for
end if

Because ambiguous orientations are not allowed, trims at the same level of the hierarchy will have the same orientation. This orientation is referenced above as *tl.is_clockwise*. The variable *keep* determines whether the point should be culled.

4 Results

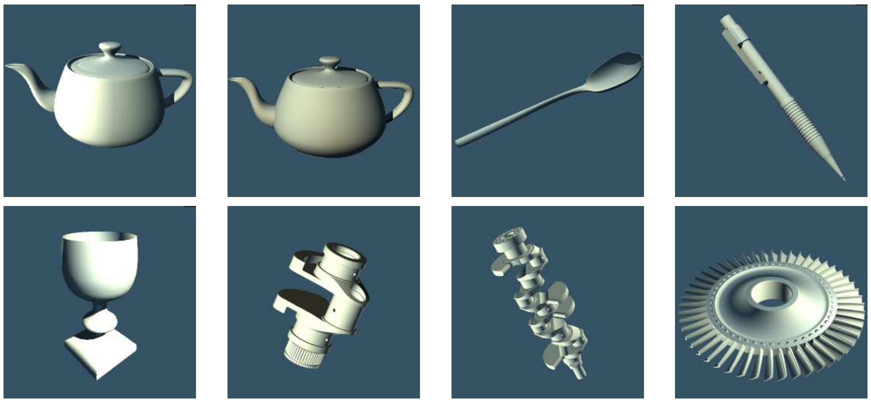
We have generated some images (see Figures 9 and 10) and timings for data sets rendered using our technique. All timings are for a single 300MHz R12K MIPS processor with an image resolution of 512x512. All models were Phong shaded and timings include shadow rays.

We have implemented our method in a parallel ray tracing system, and have obtained interactive rates with scenes of moderate geometric complexity. For a discussion of that system, we refer the reader to Parker et al. [18].

Source code and other material related to the system which we have described can be found online at http://www.acm.org/jgt/papers/MartinEtAl00.

Acknowledgments

Thanks to Michael Stark for substantial help analyzing the numerical properties of the algorithm. Thanks also to Brian Smits for helpful discussions, feedback, and encouragement, to Steve Parker for invaluable aid with parallelization and some of the nastier debugging, and to Amy Gooch for comments on the final draft. This work was supported in part by NSF grant 9720192 (CISE New Technologies), DARPA grant F33615-96-C-5621, and the NSF Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-89-20219). All opinions, findings, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.



Statistics	teapot	teapot-solid	spoon	pencil
Number of Surfaces	32	10	3	17
Number of Trims	0	5	0	9
Number of Rays	262144	262144	262144	262144
Total Time (sec.)	14	17	4.8	10
BV Intersections	12642506	10861260	3265298	3439462
Light BV Intersections	6542330	5098600	902120	1714142
NURBS tests	431620	642511	155118	475934
Total NURBS time (sec.)	9.35	12.17	3.42	8.28
Avg Time per NURBS (sec.)	2.17E-5	1.89E-5	2.20E-5	1.74E-5
NURBS hits (% of tot tests)	367307 (85.1%)	458061 (71.3%)	79387 (51.2%)	226927 (47.7%)
Reported hits (% of tot tests)	119597 (27.7%)	196051 (30.5%)	21753 (14.0%)	36577 (7.7%)
Statistics	goblet	Crank1A	crank	allblade
Number of Surfaces	1	20	73	351
Number of Trims	0	18	64	0
Number of Rays	262144	262144	262144	262144
Total Time (sec.)	7.8	78	40	61
BV Intersections	3756604	23818532	14716416	47701788
Light BV Intersections	1638038	6669190	3334258	17060564
NURBS tests	320622	2287689	1306480	2340071
Total NURBS time (sec.)	5.97	39.39	20.82	43.46
Avg Time per NURBS (sec.)	1.86E-5	1.72E-5	1.59E-5	1.86E-5
NURBS hits (% of tot tests)	226753(70.7%)	1488391 (65.1%)	542912 (41.6%)	1319143 (56.4%)
Reported hits (% of tot tests)	100001 (31.2%)	209344 (9.2%)	103525 (7.9%)	445496 (19.0%)

Figure 8: Statistics for our technique. “Light BV intersections” are generated by casting shadow rays and are treated (and measured) separately from ordinary BV intersections. “NURBS tests” gives the number of numerical NURBS surface intersections performed. “Total NURBS time”and “Avg time per NURBS” give the total and mean time spent on numerical surface intersections, respectively. “NURBS hits” denotes the number of numerical intersections which yielded a hit. “Reported hits” gives the number of successful numerical hits which were not eliminated by trimming curves or by comparison with the previous closest hit along the ray.

Figure 9: A scene containing NURBS primitives. All of the objects on the table are spline models which have been ray traced using the method presented in this paper.

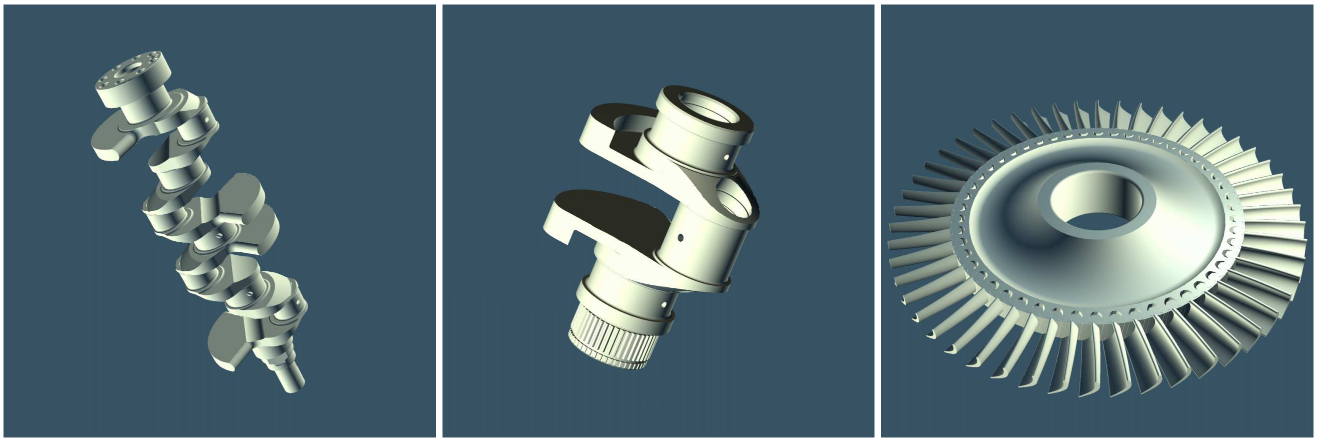


Figure 10: Mechanical parts produced by the Alpha_1 [11] modeling system (crank, Crank1A, and allblade).

References

[1] Salim S. Abi-Ezzi and Srikanth Subramaniam. Fast dynamic tessellation of trimmed NURBS surfaces. In *Eurographics '94*, 1994.

[2] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An introduction to splines for use in computer graphics and geometric modeling*. Morgan Kauffman Publishers, Inc., Los Altos, CA, 1987.

[3] W. Barth and W. Stürzlinger. Efficient ray tracing for Bézier and B-spline surfaces. *Computers and Graphics*, 17(4), 1993.

[4] W. Boehm. Inserting new knots into B-spline curves. *Computer-Aided Design*, 12:199–201, July 1980.

[5] Swen Campagna, Philipp Slusallek, and Hans-Peter Seidel. Ray tracing of spline surfaces: Bézier clipping, Chebyshev boxing, and bounding volume hierarchy – a critical comparison with new results. Technical report, University of Erlangen, IMMD IX, Computer Graphics Group, Am Weichselgarten 9, D-91058 Erlangen, Germany, October 1996.

[6] E. Cohen, T. Lyche, and R. Riesenfeld. Discrete B-splines and subdivision techniques in computer-aided geometric design and computer graphics. *Comput. Gr. Image Process.*, 14:87–111, October 1980.

[7] Gerald E. Farin. *Curves and surfaces for computer aided geometric design: a practical guide, 4th ed.* Academic Press, Inc., San Diego, CA, 1996.

[8] Alain Fournier and John Buchanan. Chebyshev polynomials for boxing and intersections of parametric curves and surfaces. In *Eurographics '94*, 1994.

[9] Andrew Glassner, ed. An introduction to ray tracing. 1989.

[10] Josef Hoschek and Dieter Lasser. *Fundamentals of computer aided geometric design*. A.K. Peters, Wellesley, MA, 1993.

[11] Integrated Graphics Modeling Design and Manufacturing Research Group. *Alpha1 geometric modeling system, user's manual*. Department of Computer Science, University of Utah.

[12] James T. Kajiya. Ray tracing parametric patches. *Computer Graphics (SIGGRAPH '82 Proceedings)*, 16(3):245–254, July 1982.

[13] Subodh Kumar, Dinesh Manocha, and Anselmo Lastra. Interactive display of large NURBS models. *IEEE Transactions on Visualization and Computer Graphics*, 2(4), December 1996.

[14] Daniel Lischinski and Jakob Gonczarowski. Improved techniques for ray tracing parametric surfaces. *The Visual Computer*, 6(3):134–152, June 1990. ISSN 0178-2789.

[15] William L. Luken and Fuhua (Frank) Cheng. Comparison of surface and derivative evaluation methods for the rendering of NURB surfaces. *ACM Transactions on Graphics*, 15(2):153–178, April 1996. ISSN 0730-0301.

[16] T. Lyche, E. Cohen, and K. Morken. Knot line refinement algorithms for tensor product B-spline surfaces. *Computer Aided Geometric Design*, 2(1-3):133–139, 1985.

[17] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. In *SIGGRAPH '90*, 1990.

[18] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. *1999 ACM Symposium on Interactive 3D Graphics*, pages 119–126, April 1999. ISBN 1-58113-082-1.

[19] John W. Peterson. Tessellation of NURBS surfaces. In Paul Heckbert, editor, *Graphics Gems IV*, pages 286–320. Academic Press, Boston, 1994.

[20] Les A. Piegl and W. Tiller. *The NURBS Book*. Springer Verlag, New York, NY, 1997.

[21] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes in C: The art of scientific computing (2nd ed.). 1992. ISBN 0-521-43108-5. Held in Cambridge.

[22] Alyn Rockwood, Kurt Heaton, and Tom Davis. Real-time rendering of trimmed surfaces. In *SIGGRAPH '89*, 1989.

[23] Jon Rokne. The area of a simple polygon. In James R. Arvo, editor, *Graphics Gems II*. Academic Press, 1991.

[24] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998. ISSN 1086-7651.

[25] Michael A.J. Sweeney and Richard H. Bartels. Ray tracing free-form B-spline surfaces. *IEEE Computer Graphics & Applications*, 6(2), February 1986.

[26] Thomas V. Thompson II and Elaine Cohen. Direct haptic rendering of complex NURBS models. In *Proceedings of Symposium on Haptic Interfaces*, 1999.

[27] Daniel L. Toth. On ray tracing parametric surfaces. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):171–179, July 1985. Held in San Francisco, California.

[28] Chang-Gui Yang. On speeding up ray tracing of B-spline surfaces. *Computer Aided Design*, 19(3), April 1987.