Accelerated Walkthrough of Large Spline Models

Subodh Kumar *
Johns Hopkins University

Dinesh Manocha † Hansong Zhang† ‡ Kenneth Hoff† §
University of N. Carolina

Spline surfaces are routinely used to represent large-scale models for CAD and animation applications. In this paper, we present algorithms for interactive walkthrough of complex NURBS models composed of tens of thousands of patches on current graphics systems. Given a spline model, the algorithm precomputes simplification of a collection of patches and represents them hierarchically. Given a changing viewpoint, the algorithm combines these simplifications with dynamic tessellations to generate appropriate levels of detail. We also propose a system pipeline for parallel implementation on multi-processor configurations. Different components, such as visibility and dynamic tessellation, are divided into independent threads. We describe an implementation of our algorithm and report its performance on an SGI Onyx with RealityEngine^2 graphics accelerator and using three R4400 processors. It is able to render models composed of almost than $40,000$ Bézier patches at $7-15$ frames a second, almost an order of magnitude faster than previously known algorithms and implementations.

Keywords: Computer Graphics, Image Generation, NURBS Surface display, Performance, Algorithms

1 Introduction

Spline surfaces are commonly used to represent models for computer graphics, geometric modeling, CAD/CAM and animation. Large scale models consisting of tens of thousands of such surfaces are commonly used to represent shapes of automobiles, submarines, airplanes, building architectures, sculptured models, mechanical parts and in applications involving surface fitting over scattered data or surface reconstruction. Many applications like interactive walkthroughs and design validation need to interactively visualize these surface models.

In order to exploit the recent advances in triangle rendering capabilities of graphics systems, it is common to generate a one-time polygonal approximation of surfaces and discard the analytic representation. Unfortunately such polygonal approximations require an unduly large number of polygons, thus necessitating the need for polygon simplification. What is more, due to discretization of geometry, even a dense tessellation is sometimes inadequate for zoomed up views. On the other hand, recent advances [18, 21, 26] in efficient tessellation of surfaces now allow us to dynamically generate an appropriate number of triangles from the analytic representation based in the user's location in a virtual environment.

The problem of rendering splines has been well-studied in the literature and a number of techniques based on polygonization, ray-tracing, scan-line conversion and pixel-level subdivision have been proposed. The fastest algorithms are based on polygonization. Algorithms based on view-dependent polygonization have been proposed to render spline surfaces. However, the fastest algorithms and systems can only render models composed of a few thousand spline patches at interactive rates on high-end graphics systems (e.g. SGI Onyx with RealityEngine^2 graphics accelerator)

Main Contribution: In this paper, we present algorithms for interactive walkthrough of complex spline models composed of tens of thousands of patches on current graphics systems. This is almost *one order of magnitude speed-improvement* over previously known algorithms and systems. We achieve such an improvement by using spline surface simplification, parallel processing and effective combination of static levels of detail and dynamic tessellation. Our main contributions are:

- **Surface Simplifications:** Most surface triangulation algorithms produce at least two triangles for each tensor product Bèzier patch. Furthermore, each trimming curve must be tessellated into at least one edge, and it adds further to the triangle count. This may result in too many triangles for parts of a model. For example, a small part like the Utah teapot consists of 32 Bèzier patches. However, 64 triangles are not required to represent it when its size is, say, a few pixels on screen. As part of pre-processing, we compute polygonal simplifications of a mesh of trimmed Bèzier patches. We present techniques to combine adjacent surfaces and compute polygonal approximations for these *super-surfaces*. For a super-surface with n Bèzier patches, we are able to generate polygonal simplifica-

*Department of Computer Science, Johns Hopkins University, Baltimore MD 21218-2694. Email: subodh@cs.jhu.edu, Web: http://www.cs.jhu.edu/~subodh

†Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. Email: manocha@cs.unc.edu, Web: http://www.cs.unc.edu/~manocha.

‡Email: zhangh@cs.unc.edu, Web: http://www.cs.unc.edu/~zhangh

§Email: hoff@cs.unc.edu, Web: http://www.cs.unc.edu/~hoff

tions with less than $2n$ triangles. Moreover, simplification of 'super-surfaces', rather than complete objects, affords us finer control on our levels of detail and allows us to significantly reduce the number of triangles needed to approximate parts of a model. This allows us to spend resources on the generation of high detail for parts close to the viewer without throttling the triangle rendering pipeline.

- **Dynamic Tessellation and LOD Management:** The algorithm represents the surface patches and their simplifications using spatial hierarchies. Given a viewpoint, the algorithm computes an appropriate polygonal approximation based on surface simplification and incremental triangulation. Since two adjacent super-surfaces may be rendered at different detail, we present algorithms to prevent cracks between super-surfaces.

- **Multi-Processor NURBS Pipeline:** It is common for current graphics system to be equipped with multiple general purpose processors in addition to specialized graphics accelerating hardware. We present a novel pipeline for rendering NURBS (Non Uniform Rational B-Splines) on multi-processor shared-memory architectures. In particular, we allocate different components of our algorithm (e.g. visibility, dynamic tessellation etc.) to independent threads. Various threads communicate using only a few locks per frame. Our pipeline helps reduce the rendering latency as well.

- **System Implementation:** We demonstrate the effectiveness of our algorithm by implementing it on an SGI Onyx with RealityEngine^2. Using a configuration with three 200MHz R4400 processors we are able to render an architectural model composed of almost $40,000$ Bèzier patches at interactive frame rate.

Organization: The rest of the paper is organized in the following manner. In Section 2, we briefly describe various techniques for rendering NURBS models. Section 3 present techniques to spatially organize spline patches into super-surfaces and compute simplifications with guaranteed error bounds. Section 4 discusses the management of levels of detail. We present our system pipeline and parallel algorithm in Section 5. Section 6 describes our implementation and highlights its performance on an architectural model. Finally Section 7 concludes our presentation and offers some future research directions.

2 Background

While the techniques presented in this paper are generally applicable to any analytic representation of models, we describe it in terms of Bèzier surfaces. Indeed, Bèzier and NURBS surfaces are among the most popular modeling tools for computer aided design. For efficient display, we decompose each NURBS surface into a collection of Bèzier patches using knot insertion [8].

A tensor product rational Bèzier patch, $\mathbf{F}(u, v)$, of degree $m \times n$, defined for $\mathcal{D} = u \times v$, $(u, v) \in [0,1] \times [0,1]$, is specified by a mesh of control points, $\mathbf{p}_{ij}$, and their weights, $w_{ij}, 0 \leq i \leq m, 0 \leq j \leq n$:

$$\mathbf{F}(u,v) = \frac{\displaystyle\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\mathbf{p}_{ij}\mathcal{B}_i^m(u)\mathcal{B}_j^n(v)}{\displaystyle\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij}\mathcal{B}_i^m(u)\mathcal{B}_j^n(v)}$$

where the Bernstein function $\mathcal{B}$ is given by

$$\mathcal{B}_i^n(t) = \left(\begin{array}{c} n \\ i \end{array} \right) t^i (1-t)^{n-i}$$

A trimmed Bézier patch has trimming Bézier curves associated with it. A rational Bèzier curve $\mathbf{f}(t)$, of degree n, defined for parameter $t \in [0,1]$, is specified by a sequence of control points, $\mathbf{p}_k$, and their weights, $w_k, 0 \leq k \leq n$:

$$\mathbf{f}(t) = \frac{\displaystyle\sum_{k=0}^{n} w_k \mathbf{p}_k \mathcal{B}_k^n(t)}{\displaystyle\sum_{k=0}^{n} w_k \mathcal{B}_k^n(t)}$$

The basic thrust of our approach is to combine dynamic tessellation of surfaces, for high detail, with a few discrete levels of detail, for efficiency. We do not generate levels of detail for each object or solid, but rather for a collection of Bèzier patches. This affords us better control on allocation of detail for large models, but requires extra processing for seamless integration of parts.

2.1 Related Work

There is considerable literature on rendering splines, polygon simplification and parallel rendering algorithms. A number of surface rendering techniques are based on ray tracing [14, 25, 31], scan-line generation [3, 22, 30], and pixel level subdivision [4, 28, 27]. However, due to recent advances in hardware based triangle rendering techniques, algorithms based on polygonal decomposition are much faster in practice. In particular, a number of algorithms based on uniform and adaptive subdivision have been proposed [5, 17, 9, 26, 2, 10, 1, 23, 16]. Uniform subdivision, in general, is more efficient [18]. Recently Rockwood et al. [26] and Kumar et al. [18, 21] have proposed uniform subdivision based algorithms for interactive display of trimmed surfaces. A variant of [26]'s algorithm has been implemented in SGI GL and OpenGL libraries. It can display models composed of a few hundred patches at interactive frame rates on an SGI Onyx with RealityEngine^2. Kumar et al. [18, 21] proposed a faster algorithm that includes back-patch visibility, improved bounds for polygonization and incremental triangulation exploiting frame-to-frame coherence. The resulting pipeline is shown in Figure 1. On an SGI Onyx with 250MHz R4400 and RealityEngine^2 it is able to render models composed of a few thousand patches at 10-15 frames a second.

In our experience, many real world models of ships, submarines etc. are composed of many more than a few thousand Bèzier patches. While surface models typically have areas of high detail and curvature, often each individual patch is relatively flat and can be rendered at very low detail, specially if they occupy a small area on the screen. Unfortunately, the patch based rendering algorithms produce at least two triangles even for such patches resulting, at times,

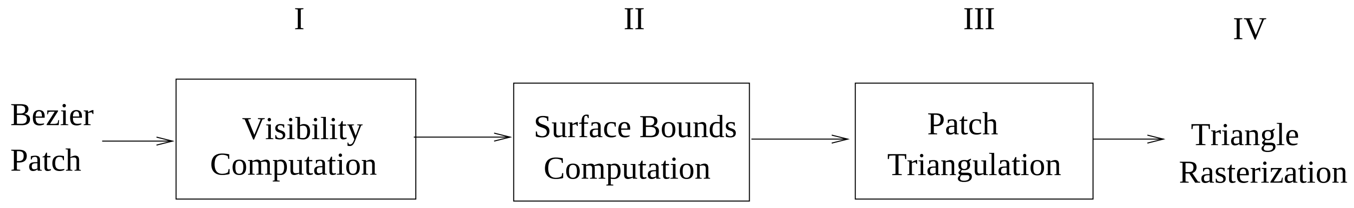

Figure 1: On-line Polygonization of NURBS [21]

in triangle proliferation. Simplifying the spline models could ameliorate this problem. A number of techniques have been proposed to simplify polygonal models. Some topology preserving techniques include [6, 7]. However, they do not generalize to spline surfaces directly. It is not straightforward to devise a technique that:

1. simplifies splines while preserving topology and guaranteeing small error, and

2. results in a drastic simplification of models.

Note that the major application of splines in walkthroughs is to allow arbitrary levels of detail. However, we need to simplify splines only when the corresponding part requires a low-detail rendering. Hence, the simplification obtained by using the polygonal approximation is sufficient, and we do not require "true" spline simplification. Our approach is to group Bèzier patches into super-surfaces and, generate triangular approximations for each super-surface, and generate a number of discrete levels of detail thereof. At rendering time, we pick an appropriate approximation for each super-surface and stitch adjacent super-surfaces together.

In addition to reducing the number of triangles sent to the triangle rendering pipeline, we also present techniques to efficiently map the walkthrough application onto multiple processors on a shared memory architecture. A number of parallelization techniques are known in the literature [11, 12, 15, 19]. In particular Kumar et al. [19] present a dynamic load balancing technique for surface tessellation with negligible overhead. We augment their technique with modified pipeline described in Section 5.

2.2 Dynamic Tessellation

Previous algorithms for interactive display [1, 26] spend considerable time in computing appropriate view-dependent tessellation bounds and polygonal triangulations. The incremental algorithm in [18, 21] overcome these problems using a combination of off-line and on-line bound computations and incremental triangulations. Their algorithm, in brief, is as follows:

1. Determine visibility. This includes computation of both view-frustum visibility and back-patch visibility [20].

2. For each Bèzier patch, determine the number of uniform tessellation steps in the u and v dimensions, respectively, required for a smooth image from the current view point. Similarly, for each trimming curve, determine the number of steps.

3. Triangulate the samples picked in the previous step, so that there are no cracks at the boundaries between adjacent patches.

4. At each subsequent frame, update the triangulation incrementally making small changes to the current triangulation and taking advantage of coherence.

5. Send the updated triangles to the triangle rendering pipeline.

This pipeline is represented schematically in Figure 1. [21] maps stages I and II on the host CPU of the system. Stages III and IV are parts of the triangle rendering subsystem.

We implemented this pipeline and discovered that approximately half the polygon generation time is spent in step size determination in stage II [18]. This number is relatively high; the computational cost of incremental triangulation and tessellation is low due to frame-to-frame coherence. As we apply the algorithm to large models, this becomes a significant bottleneck in the overall pipeline. We employ a spatial hierarchy to solve this problem. Instead of determining the step size and generating triangles for each Bèzier patch, we perform these computations for groups of patches. Note that this algorithm suffers from the limitation that each Bèzier patch must be approximated by at least one quad (for tensor-product patches), i.e. 2 triangles. The minimum count is even higher for trimmed patches. Thus n tensor-product Bèzier patches must result in more than $2n$ triangles. On the other hand, by tessellating super-surfaces, we can obtain many fewer triangles.

However, in order to reduce the number of triangles generated, we introduce extra processing. As a result stage II becomes a bottleneck. We eliminate this bottleneck by decoupling this stage from the pipeline. We take advantage of the multiple processors to allow such processing to occur in the "background".

2.3 Simplification Envelopes

In order to perform polygonal simplification of polygonal surfaces we employ the technique of Simplification envelopes [6, 29]. In brief, for a given ϵ, it guarantees that the resulting simplification is at most ϵ distant from the original surface:

1. For a given surface, generate two offset surfaces, one on either side, each at most ϵ distant from it.

2. Remove a vertex from the given surface, and retriangulate the hole such that none of the resulting triangles intersect the offset surfaces.

3. Stop if no more vertices can be removed.

Each of our super-surfaces is C^0 continuous and contain borders. For each border we construct the border-tubes described by Cohen et al. [6] to facilitate crack-prevention. In addition, the properties of super-surfaces allow us to obtain high vertex reduction.

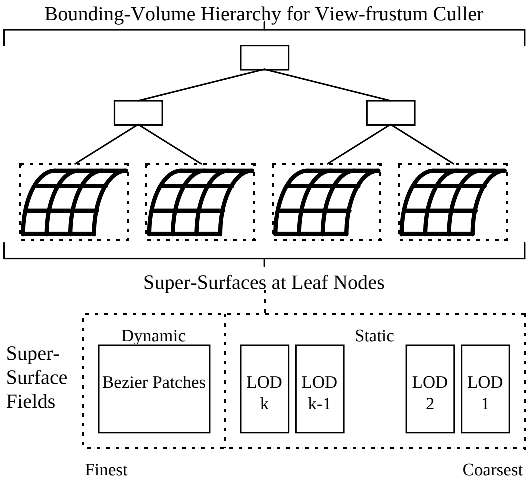

Figure 2: Model Representation

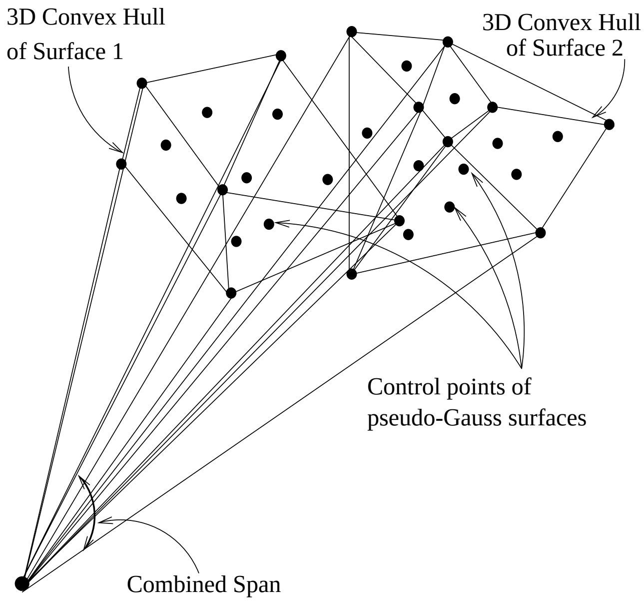

Figure 3: Span of a Super-surface

3 Super-surfaces

We assume that the input model is a collection of trimmed Bèzier patches. As a part of pre-processing, our algorithm forms collection of patches into super-surfaces, and computes a spatial hierarchy that is later used for view-frustum culling and controlling the level of detail. It also tessellates each super-surface into triangles and generates discrete approximations for each super-surface.

Our clustering algorithm partitions an input model into super-surfaces based on surface normals and patch adjacency. A super-surface corresponds to a C^0 continuous mesh of Bèzier patches. Furthermore, our algorithm ensures that the variation in normals along the surface boundary is bounded. While algorithm of Cohen et al. [6] still guarantees bounded error for surfaces with high curvature and sharp edges, it is not able to remove many vertices for such surfaces. Using pseudo-Gauss maps [21], we are able to ensure that a super-surface is does not have much variation in curvature. We also make use of any natural structure or hierarchy of the input model to speed up the super-surface construction. For example, a NURBS surface is decomposed into a mesh of Bèzier patches by knot-insertion. The resulting set of Bèzier patches may be grouped into one super-surface.

In order to limit the curvature of each super-surface, we restrict the extent of its Gauss-map. The Gauss-map of a surface corresponds to the projection of its normals on the unit sphere. Fortunately we do not have to compute the actual projection, which is relatively expensive; we only have to consider the pseudo-Gauss map, $\mathbf{G}$, which is easy to write in the Bèzier form:

$$\mathbf{G} = \frac{\mathbf{P}_u \times \mathbf{P}_v W - \mathbf{P}_u \times \mathbf{P} W_v - \mathbf{P} W_u \times \mathbf{P}_v}{W^3}. \qquad (1)$$

The pseudo-Gauss map of a rational Bèzier surface of degree $m \times n$ is itself a Bèzier surface of degree $3m \times 3n$, and hence is bounded by the convex hull of its control points. For each point on the hull, consider the ray from the origin to that point (we ignore any hull points that lie on the origin). To compute the extent of the Gauss-map, we only need to find the maximum angle between any two rays. We refer to the maximum angle as the patch's *span*. To compute the variation of curvature across two (or more) Bèzier patches, we first compute the convex hull of the union of control points of respective pseudo-Gaus maps and then compute its span (see Figure 3).

The super-surface construction algorithms starts with a patch as a seed and grows it by merging with neighboring patches forming a super-surface, if the resulting span is within user-specified bounds. If a hierarchy of adjacent patches is available, the algorithm first groups together each patch in a hierarchy before attempting to grow the group further. If the span of the group is too big, the algorithm subdivides it into two or more super-surface. The major components of the algorithm are described below:

1. Consider all patches adjacent to a super-surface, not added to another super-surface:
   (a) Pick the one that increases the span by the minimum amount.
   (b) If the minimum span is greater than an input parameter α, output a super-surface; start a new super-surface.
   (c) Add the patch to the super-surface.
   (d) Update the adjacency of the new super-surface.
   (e) Compute spans for newly adjacent patches.
   (f) Recursively apply 1a.
2. To start a new super-surface, randomly pick a new seed to grow.

4 LOD Generation

Given a super-surface composed of n patches, the LOD generation algorithm generates discrete levels of detail, each with a different value of (user prescribed) ϵ, the maximum error. Initially, the algorithm tessellates each Bèzier patch into triangles. These triangular approximations are then simplified using offset surfaces. Although the dynamic surface tessellation algorithm performs uniform subdivision for

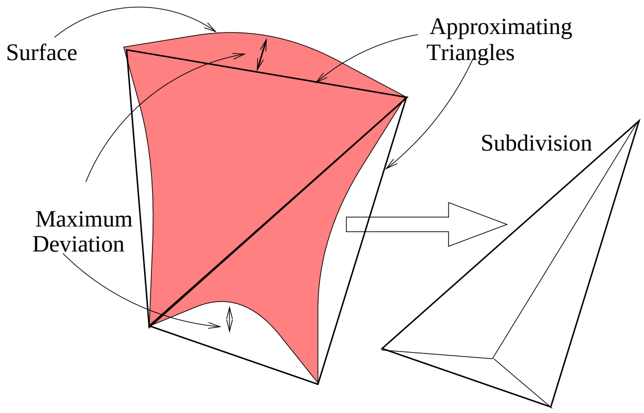

Figure 4: Adaptive Tessellation of Bèzier Patch

efficiency, we use adaptive subdivision for generating the approximation for simplification since this is a pre-processing step. We can afford to spend more time in order to generate good approximations using fewer triangles. Our adaptive tessellation algorithm ensures that the maximum deviation between the Bèzier patch and its triangular approximation is at most δ (a user-specified value). This in turn ensures that the simplified polygons are no more than $\delta + \epsilon$ away from the surface.

4.1 Adaptive Tessellation

The adaptive tessellation algorithm initially approximates each patch with two triangles. For each triangle, it computes the maximum deviation between each triangle and the surface using bounds on derivatives [9].

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 [24, 18]. Thus all local extrema of the deviation function are obtained. Each triangle Δ that we generate corresponds to a triangle $\Delta_{\mathcal{D}}$ in the domain of the patch. We denote by $Dev(\Delta)$, its maximum deviation from the part of the surface it approximates. This maximum deviation occurs either at one of the three vertices of $\Delta_{\mathcal{D}}$ or at any local extrema contained in $\Delta_{\mathcal{D}}$. The adaptive tessellation algorithm proceeds as follows: (see Figure reffig:adapt)

1. For each patch, generate two triangles by adding one of the diagonals. We choose the diagonal that minimizes deviation. Say, the diagonal d_1 generates triangles Δ_1 and Δ_2, and the diagonal d_2 generates triangles Δ_3 and Δ_4. If $\max(Dev(\Delta_1),\ Dev(\Delta_2)) < \max(Dev(\Delta_3),\ Dev(\Delta_4))$, we choose d_1, otherwise we choose d_2.

2. Divide each triangle $\Delta = p_1p_2p_3$ with $Dev(\Delta) > \delta$ as follows:

   Let (u_1, v_1) be the point on the domain of patch $\mathbf{F}$ at which $\mathbf{F}_{uv}$ is maximized, and let $p_4 = \mathbf{F}(u_1, v_1)$. The new set of triangles is $\{p_1p_4p_2, p_2p_4p_3, p_3p_4p_1\}$. In general, (u_1, v_1) may lie on one of edges, e.g. p_1p_2, of the Δ. Instead of generating degenerate triangles, we generate only two triangles by adding an edge from p_4 to the opposite vertex, p_3 in our example. Note that (u_1, v_1) cannot lie on a vertex of Δ – the deviation at a vertex is zero.

3. Subdivide each triangle recursively into smaller triangles until the Dev of all triangles is less than δ.

After generating the triangular approximation we simplify it. Our construction algorithm ensures that each super-surface is relatively flat and has low variance in normals and curvature. As a result, the offset envelope approach is able to simplify the models by $30-80\%$.

4.2 Super-surface Boundary

In the terminology of [6], our super-surfaces are surfaces with *borders*, as opposed to closed surfaces. In order to ensure that the approximation to the border also has a small error, we simplify each border first and then simplify the interior without removing any more vertices from the border.

Furthermore, for our application, we must stitch adjacent super-surfaces together. While original model may have no cracks between super-surfaces, using different approximations for adjacent super-surfaces can result in artificial holes, overlaps or intersections. Correcting these artifacts can be quite expensive. Instead, we propose a solution that does not generate such artifacts in the first place. We always pick the same ϵ-approximation for the boundary curve between two super-surfaces. We simplify the interior separately from the boundary. It is possible to generate for each super-surface and each of its boundaries, an $\epsilon_1\epsilon_i$-approximation where the the interior is an ϵ_1-approximation and the boundary is an ϵ_i-approximation. However, memory needed to store the triangulations for all combinations of ϵ values can be quite large. We, instead, modify our adaptive tessellation algorithm to triangulate only the interior of each super-surface. We generate a boundary strip at run time. The algorithm for generation of triangular approximation is as follows:

1. Generate an adaptive tessellation of the boundary curve. Include all *corner* points in the tessellation. A corner point on the border of a super-surface is a point adjacent to two other super-surfaces (see Figure refsuperb).

2. Generate $\mathbf{b}_\epsilon$, an ϵ-approximation of the boundary $\mathbf{b}$ curve, such that $\mathbf{b}_\epsilon$ contains all corner points.

3. In the domain of the super-surface, construct $\mathbf{o}'_\epsilon$, an offset curve ϵ' distant from $\mathbf{b}_\epsilon$. In order to maintain similar triangles sizes in the interior and the border ϵ' should be large enough to avoid narrow triangles at the border. However, in the interest of efficient triangulation, we let $\epsilon' = \epsilon$, as that generates non-intersecting offsets (see Appendix). Note that ϵ is the relative error, hence we can use it in separate domains: the 2D parametric space of the patches and the 3D object space

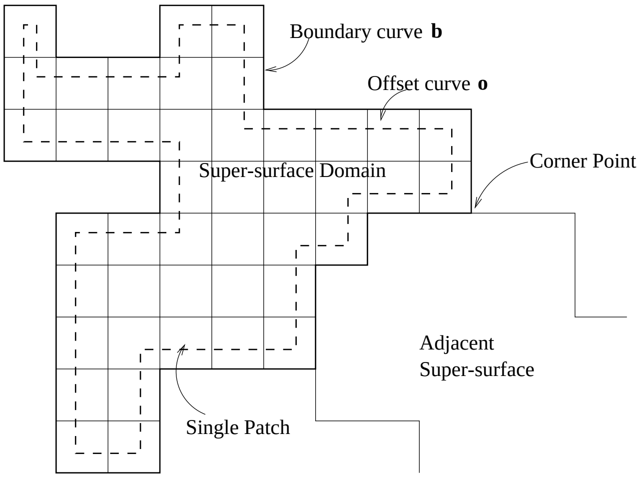

Figure 5: Super-surface Domain

It is possible to reverse the order of the two steps above. We can compute the offset curve of the original tessellated boundary, rather than the simplified boundary and then simplify the offset curve. In that case larger values of ϵ' are needed.

4. Generate an ϵ-approximation of the interior of the super-surface bounded by the offset curve (i.e. the ϵ-approximation of the offset curve).

At rendering time, for each segment of the boundary curve, we pick the smaller of the two ϵ values corresponding to the two super-surfaces adjacent to it. Thus for a given super-surface, a given border curve may be tessellated based on a value of ϵ different from that used for the interior. We triangulate this strip at rendering time. This triangulation is not a costly operation since it is performed in the domain. We triangulate two simple chains by stepping along each chain and adding edges between vertices on the chains.

If topological consistency is not crucial and drastic simplifications are desired, we have found that at small scales, approximating each super-surface by two triangles, even if the adjacent super-surface is approximated by more than two triangles, hardly distracts from the realism of the walk-through. At times, all super-surface adjacency information is not available in an input model and cannot be resolved. In such cases, it is not possible to generate approximations that are free of artifacts.

4.3 Trimmed Patches

The algorithms described above easily generalizes to trimmed Bèzier patches. Trimming curves are treated like boundary curves. In general, a trimming curve implies a large discontinuity in normals between the two patches adjacent to it. As a result, most trimming curves form boundaries between super-surfaces. It is possible for a trimming curve to lie entirely inside a super-surface i.e. both patches adjacent to a trimming curve belong to the same super-surface. Such trimming curves need no extra processing at rendering time. At pre-processing time, we generate a valid mesh for the super-surface with the trimming curve and apply the simplification algorithm.

However, we need to generate offset curves for the boundary trimming curves. This offset curve is a simple polygonal chain of line segments. We first generate the adaptive tessellation of the trimming curve, and then generate its offset curve.

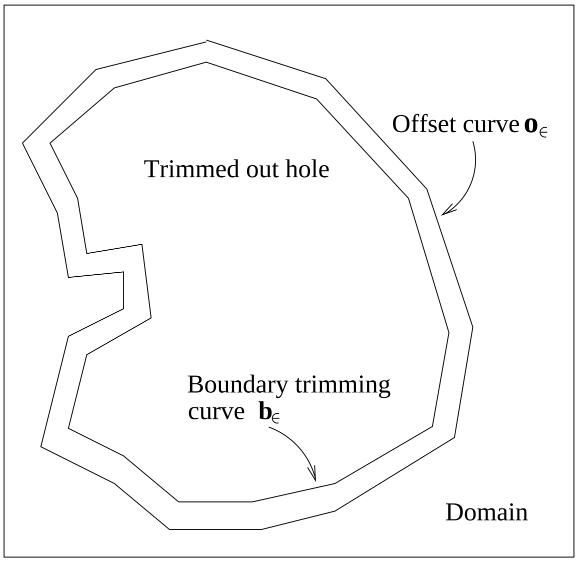

Figure 6: Trimming Curve Offset

The algorithm for the generation of this offset curve $\mathbf{o}_\epsilon$ of the boundary curve $\mathbf{b}_\epsilon$ is as follows:

1. For each segment s of $\mathbf{b}_\epsilon$, add segment s' to $\mathbf{o}_\epsilon$, such that s is parallel to s', is ϵ distant from it on the untrimmed side of $\mathbf{b}_\epsilon$, i.e. construct the offset segment on the patch.

2. For an approximation $\mathbf{b}_\epsilon$ with high curvature $\mathbf{o}_\epsilon$ may self intersect. We delete such loops from $\mathbf{o}_\epsilon$, and mark the vertex of the loop (see Figure 7).

Note that the semantics of this offset curve is different from that in [6]. Their algorithm generates offsets closer to the curve in such cases. While we can use their definition, our technique allows us to make the following claim (see Appendix for the proof):

Theorem 1 $\mathbf{o}_{\epsilon_2}$ *and* $\mathbf{b}_{\epsilon_1}$ *do not intersect, if* $\epsilon_1 \leq \epsilon_2$.

At rendering time, we again perform the triangulation between the ϵ_1-approximation of the boundary and ϵ_2-approximation of the offset curve in the 2D domain. We exploit the fact that the offset curve is quite close to the boundary and perform the triangulation by stepping along the two curves, traversing the two corresponding chains from a marked vertex to the next marked vertex.

4.4 LOD Control

Our levels of detail control is well integrated with the view frustum culling hierarchy.

Our visibility algorithm represents the model in a hierarchy of bounding volumes. Each leaf node of the view-frustum hierarchy corresponds to a super-surface (as shown in Figure 2). The algorithm uses a top down approach to build a tree.

The visibility algorithm outputs a list of leaf nodes that are visible. Note that even if the visibility decision is made at an internal node of the tree, all the corresponding leaves

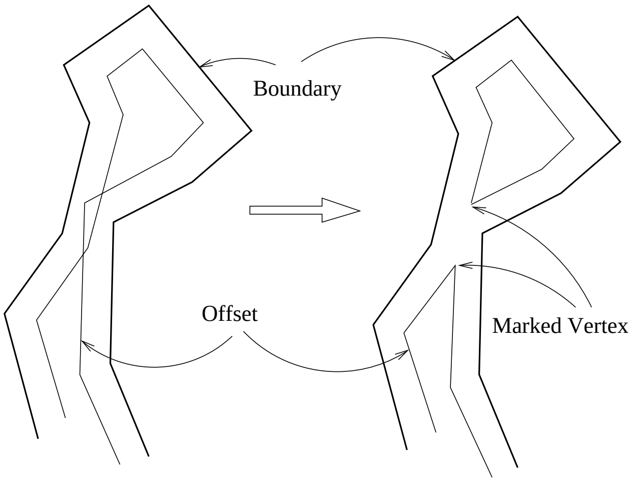

Figure 7: Offset Curves: Self Intersection

must be output, since the super-surfaces contained in the leaf nodes are the inputs to our rendering algorithm. For each visible leaf node, corresponding to a super-surface with n Bèzier patches, our algorithm computes a crack-free tessellation. First, each boundary curve is approximated. The algorithm uses oriented bounding box of each curve to determine its ϵ value [6]. This ensures that the same approximation of the curve is used for boundary strip on both its sides, thus preventing cracks. The interior tessellation uses the bounding box of the entire super-surface. Note that a boundary curve may be result in static LOD while the approximation of the interior of an adjacent patch may require dynamic tessellation, which is performed independently for each patch of the super-surface. In such cases, we still generate a common boundary strip for entire super-surface.

5 Multi-processor Pipeline

As we perform visibility computations (view-frustum and back-patch culling), bounds computation and dynamic tessellation on large spline models, triangle generation becomes a bottleneck. In this section, we present a parallel algorithm and a system pipeline for shared-memory multi-processor architectures. The two main goals for a walkthrough application are: smooth motion and low latency. We achieve these goals by decoupling triangle rendering with triangle generation and utilize frame-to-frame coherence. Figure 8 shows our pipeline (with three processors). If more processors are available, we allocate them for dynamic tessellation and visibility computations.

5.1 Processor Allocation

Our systems consists of three threads. Every super-surface is tessellated into triangles as a function of the viewpoint by thread T (corresponding to dynamic tessellator). The visibility computations are performed by thread V (on visibility processor). We use a greedy rendering strategy [18] and use thread P, the triangle pusher, to pass the current approximation of each super-surface down to the triangle rendering pipeline.

A thread may be allocated to more than one processors. If multiple processors are available for any thread, we use the lock-free dynamic load-balancing technique of Kumar

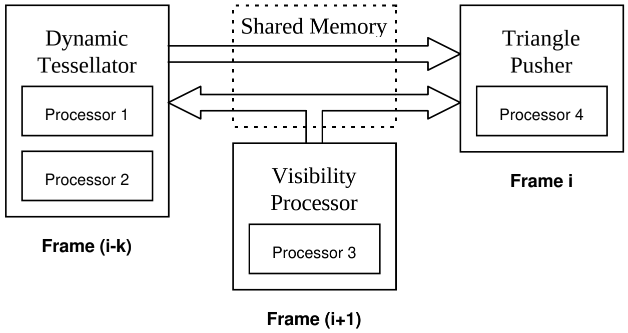

Figure 8: Multi-processor NURBS Pipeline

et al. [19] to distribute super-surfaces to those processors. The basic idea of the load-balancing[19] algorithm may be described as follows:

1. Each processor, p, maintains a local work queue $Q(p)$.
2. The basic loop of processor p consists of deleting the next element from $Q(p)$ and performing the corresponding work.
3. An *idle* processor, with no elements left in its queue, finds a *busy* processor, that has a non-empty queue.
4. The idle processor asynchronously partitions the $Q(busy)$ into two, adds one of the subqueues to its $Q(idle)$, and changes its status to busy.

The important properties of this algorithm are:

- The load-balancing algorithm itself has a low overhead. Any processor with a non empty work-queue does not explicitly perform any steps of the load-balancing algorithm. Only the processors with empty queues execute the load-balancing algorithm.
- The algorithm does not require locks for synchronization and thus further reduces the overhead.

5.2 Threads

For our pipeline description, we abstract away individual processors of a thread and consider three threads with three "processor groups". Ideally, at least one dedicated processor should be allocated to each thread in order to avoid process-context switch overheads. Tessellator thread T, which is the most compute intensive part of the NURBS pipeline, proceeds asynchronously with P. Thus, while P may display an approximation that was generated, say, k frames earlier, it never stops for T to complete. Due to frame-to-frame coherence, an update of surface tessellation once every $2-3$ frames works well in most applications. The algorithm ensures that a consistent tessellations is displayed corresponding to all the patches (to prevent cracks). Thus a new approximation is always generated in a new memory location and pointers to the tessellation of a patch are updated after completion. Using this technique we are able to avoid all locks for synchronization between T and P.

The visibility thread V executes synchronously with P. It performs visibility computations on frame i or $i+1$, while P pushes triangles corresponding to frame i. Synchronization is performed using shared variables. Our model is organized hierarchically for visibility computation. The visibility thread determines the visibility of nodes of the hierarchy; It classifies then into totally visible, partially visible or not visible at all. V adds a pointer to each completely visible node into a queue, *activity list*, and recursively traverses the tree for partially visible nodes. The triangle pushing thread P consumes the elements of the activity list, traversing the subtree corresponding to each element. The activity list has two types of End-of-Queue markers. A marker NULL implies all nodes produced by V have been consumed but V has not finished traversing the entire hierarchy. Once V completes the frame, it sets the end marker to NIL. We do not require any locks for mutual exclusion.

P uses a busy wait loop for synchronization when it encounters a NULL marker. In practice, this rarely occurs as triangle rendering is the bottleneck most of the time. After P ascertains that a super-surface is visible, it determines its required level of detail and uses a tessellation based on static LOD's or dynamically computed using incremental triangulation algorithms. It pushes the triangles corresponding to that tessellation down the graphics pipeline. The basic computation loops of the three threads are as follows:

Tessellator thread T:

1. Compute Surface bounds
2. Allocate memory M
3. Generate new triangles; save in M
4. After all triangles are generated, update P's address.

Visibility thread V — Traverse($Tree$):

1. If tree is invisible (i.e. none of the leaves are visible), continue.
2. If tree is partially visible, Traverse all children
3. If is visible (i.e. all leaves are potentially visible)
    - Append $Tree$ to the End of $ActivityList$

Triangle Pusher P:

1. Wait while Next of $ActivityList$ equals NULL
2. Delete Next element ($Tree$) of $ActivityList$
3. Traverse $Tree$, pushing each triangular approximation

The advantage of frame-overlapping V and P threads is that we are still limited in throughput by the slower of the two stages, but the latency reduces from two frames to one frame. The new user position is used by P at every frame. The advantage of allowing T to proceed asynchronously, apart from reducing latency by a frame, is that it is no longer a bottleneck. However, it is possible for different approximations of boundary curves to be used for adjacent super-surfaces. This occurs when multiple processors are allocated to thread T and only one of the super-surfaces gets updated before P reads the triangles. One possible solution is to let P choose the boundary approximation each frame. This unnecessarily complicates the logic of thread P. Unfortunately due to restrictions on concurrent access to the graphics hardware, we were limited in our implementation to allocating a single processor to thread P. Hence we decided to keep thread P simple, and chose the solution of Kumar et al. [19]:

Let T generate two boundary strips B_1 and B_2. B_1 is a triangulation that uses the new approximation of interior and the old approximation of the boundary curve. B_2 uses the new boundary and the new interior. B_2 is kept in a temporary location and P continues to use B_1. Similarly, B_1' and B_2' are asynchronously generated for the adjacent patch. The processor to perform the update of the boundary second, replaces B_1 and B_1' with B_2 and B_2', respectively, at the same time.

5.3 Multi-threads

In our current implementation we use only one processor per thread. The thread synchronization, as described above, works only if there exists only one processor per thread. In reality, a thread may execute concurrently on multiple processors. Thus there may be multiple producers and consumers executing simultaneously. The following generalization of the thread algorithm is needed for multi-threads:

Instead of one $ActivityList$, we maintain $\max N_p, N_c$ $ActivityLists$, if N_p processors are allocated to the producing thread and N_c processors are allocated to the consuming thread.

If $N_c \geq N_p$,

- Allocate N_p $ActivityLists$ to the producers.
- Statically allocate the rest $N_c - N_p$ lists to the producers in a round-robin manner.
- Each producer adds elements to each of its associated lists in a round-robin fashion.
- The $ActivityList$ associated with a given consumer is its work-queue
- The load-balancing algorithm mentioned above [19] ensures, equitable re-distribution of work among consumers.

If $N_p > N_c$,

- Associate one $ActivityList$ per consumer.
- Add a dummy consumers for each of the $N_p - N_c$ unallocated lists.
- The load-balancing algorithm eventually re-distributes the dummy queues to real processors

6 Implementation and Performance

We have completed a prototype implementation of our algorithms and report its performance on an SGI-Onyx with three 200MHz R4400 CPUs and a RealityEngine^2 graphics accelerator. We tested our system on Door, an architectural model (the entrance of a courtyard in the Yuan Ming garden, shown in Color Plate 1). This model has more than 9,900 B-spline surfaces. After applying knot-insertion algorithms, the model is composed of 38,750 Bèzier patches. Approximately 7000 of these are bi-linear. The rest are biquadric or bicubic tensor-product patches. Less than 5% of the model is composed of trimmed patches. In our current implementation, each super-surface corresponds to a B-spline patch. Color plate 2 shows distribution of the super-surfaces for the lion, one of the animals on the roof of the Door.

The model supplied to us was not clean. It had a few problems. The B-spline surfaces in the model do not have consistent orientation of the normals. That prevents us from

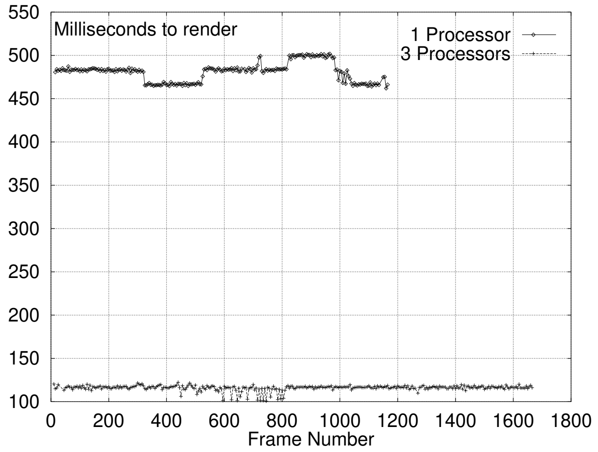

Figure 9: Performance Graph

performing back-face culling or back-patch culling. Furthermore, the model had a few cracks between the surfaces. As a result, we do not have complete adjacency information and our simplification algorithm can enlarge some of these cracks.

The simplification algorithm works well on these model and is able to achieve drastic polygon reduction. Different simplifications of the *lion* model are shown in Color Plate 3. For example when looked at from a far distance, the lowest detail (left most image) corresponds to about 80 triangles while still maintaining the general characteristics of the *lion*. At the same time, whenever the user zooms towards the *lion* (as shown in Color Plate 4), the dynamic tessellation algorithm incrementally computes a denser triangulation (as shown in Color Plate 5). Since the tessellation is updated asynchronously, we do not suffer from any slow-down due to dynamic tessellation.

The algorithm achieves considerable speed-up over earlier methods due to parallelization and simplification of super-surfaces. As for the Door model we obtain following speed-ups:

- **Parallelization:** The parallel implementation on a multi-processor SGI results in significant speed-up. The relative speed has been shown in Figure 9. For this comparison we used three CPUs, one for each thread: tessellation, visibility and triangle pushing. The sequential implementation achieves an average rate of approximately two frames a second. On the other hand, the parallel version is able to display the model at $8-10$ frames a second. The decoupling of dynamic tessellation with visibility and rendering threads accounts for most of this speed-up and helps *reduce the variation* in the frame rate. Furthermore, the tessellation thread does not lag behind the rendering by more than $2-3$ frames at most.

- **Model Simplification:** The implementation of [21] based on dynamic tessellation (with no static levels of detail) renders this model at 1.4 frames a second (using only processor). As a result, our simplification algorithms accounts for $40-50\%$ improvement in the overall frame rate.

The combination of parallel implementation and simplification improves the overall frame rate by almost *one order of magnitude*.

The static levels of detail add a little overhead to the *memory requirements* of this algorithm. Typically, we only computing four or five discrete levels for each super-surface. The additional memory needed is a linear function (with a very small constant) of the number of patches.

7 Conclusion

By effectively combining two successful techniques of simplification envelopes [6] and incremental dynamic tessellation [18], we have been able to obtain significant simplifications of large spline models while maintaining high detail where necessary. In addition, by re-arranging our rendering pipeline, we have been able to greatly speed-up the rendering of large spline models for walkthrough for a small cost of lag in quality of tessellation. Due to coherence, this lag is hardly noticeable in practice.

7.1 Future Work

Our work represents only a first step in research in methods for combining different techniques and employing the appropriate technique at any given time. Apart from better visibility and value determination algorithms, we need further research in super-surface clustering. Our algorithm does not always result in uniform sized clusters. As more super-surfaces are formed, the size of each super-surface goes down. While this results in better simplification, the number of super-surfaces can grow too big. Perhaps, a second pass, combining small super-surfaces could be performed. Another problem with our method manifests itself in the case of large number of relatively flat Bèzier patches. This results in super-surfaces that are too large and the fine control in displaying each region of a model at the appropriate detail is lost. Further investigation into a hierarchical super-surface construction is needed.

Both the super-face construction algorithm and the crack-prevention algorithm require adjacency data. We are working on robustly generating such data from an unordered collection of Bèzier patches. Another major issue is switching between discrete levels of detail. In practice, switching artifacts are not noticeable when performing dynamic tessellation incrementally. However, they become significant, if we use statically generated discrete levels. While gradually morphing from one level to another may reduce these artifacts, we believe an algorithm similar to dynamic tessellation of Bèzier patches would be more efficient. By appropriately controlling the sample density in the domain, such an algorithm would be able to incrementally update detail in an efficient fashion.

Better exploitation of available parallelism is another important goal. We need better algorithms to allocate processors to threads. A sophisticated algorithm might dynamically change processor allocation to different threads in order to achieve maximum speed-up.

7.2 Acknowledgements

We thank Lifeng Wang and the modeling group at University of British Columbia and XingXing Graphics Co. for providing the NURBS model of the Yuan Ming garden.

This work is supported in part by a Sloan fellowship, ARO Contract P-34982-MA, ARO contract DAAH04-96-1-0013, NSF grant CCR-9319957, NSF grant CCR-9625217, ONR contract N00014-94-1-0738, DARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for

Computer Graphics & Scientific Visualization NSF Prime contract No. 8920219.

References

[1] S.S. Abi-Ezzi and L.A. Shirman. Tessellation of curved surfaces under highly varying transformations. *Proceedings of Eurographics*, pages 385–397, 1991.

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

[3] J. F. Blinn. *Computer Display of Curved Surfaces.* Ph.d. thesis, University of Utah, 1978.

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

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

[6] J. Cohen, A. Varshney, D. Manocha, and G. Turk et al. Simplification envelopes. In *Proceedings of ACM SIGGRAPH*, pages 119–128, 1996.

[7] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of ACM SIGGRAPH*, pages 173–182, 1995.

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

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

[10] 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.

[11] Y. Hazony. Algorithms for parallel processing: Curve and surface definition with Q-splines. *Computers & Graphics*, 4(3-4):165–176, 1979.

[12] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Proc. Supercomputing '95*, 1995.

[13] H. Hoppe. Progressive meshes. In *Proceedings of ACM SIGGRAPH*, pages 99–108, 1996.

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

[15] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Technical Report TR95-064, Department of Computer Science, University of Minnesota*, 1995.

[16] R. Klein and W. Straber. Large mesh generation from boundary models with parametric face representation. In *Proc. of ACM SIGGRAPH Symposium on Solid Modeling*, pages 431–440, 1995.

[17] P.A. Koparkar and S. P. Mudur. A new class of algorithms for the processing of parametric curves. *Computer-Aided Design*, 15(1):41–45, 1983.

[18] S. Kumar. *Interactive Display of Parametric Spline Surfaces.* PhD thesis, University of North Carolina, 1996.

[19] S. Kumar, C. Chang, and D. Manocha. Scalable algorithms for interactive visualization of curved surfaces. In *Supercomputing*, Pittsburgh, PA, 1996.

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

[21] 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.

[22] 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.

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

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

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

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

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

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

[29] A. Varshney. *Hierarchical Geometric Approximations.* PhD thesis, University of North Carolina, 1994.

[30] J.T. Whitted. A scan line algorithm for computer display of curved surfaces. *ACM Computer Graphics*, 12(3):8–13, 1978. (SIGGRAPH Proceedings).

[31] J.T. Whitted. An improved illumination model for shaded display. *ACM Computer Graphics*, 13(3):1–14, 1979. (SIGGRAPH Proceedings).

Appendix

We use the following theorem in Section 4.2 to show the existence of valid triangulations for our offset curves.

Theorem 1 $\mathbf{o}_{\epsilon_2}$ and $\mathbf{b}_{\epsilon_1}$ do not intersect, if $\epsilon_1 \leq \epsilon_2$.

Proof: Since for each boundary curve we choose the minimum error, we ensure that $\epsilon_1 \leq \epsilon_2$. To prove Theorem 1, consider Figure 10. $\mathbf{o}_{\epsilon_2}$ is the offset curve for $b\epsilon_2$. Since $\mathbf{b}_{\epsilon_2}$ is at most ϵ_2 distant from the Bèzier trimming curve, the curve itself may not intersect $\mathbf{o}_{\epsilon_2}$. Since all points on $\mathbf{b}_{\epsilon_1}$ lie on the curve, any intersection of $\mathbf{b}_{\epsilon_1}$ with $\mathbf{o}_{\epsilon_2}$ must imply that either the curve order is $p_1p_2p_3$ or $p_1p_3p_2$. In the first case $\mathbf{b}_{\epsilon_2}$ is more than ϵ_2 far from the curve and in the second case $\mathbf{b}_{\epsilon_1}$ is more than ϵ_2 away from the curve – both contradictions.

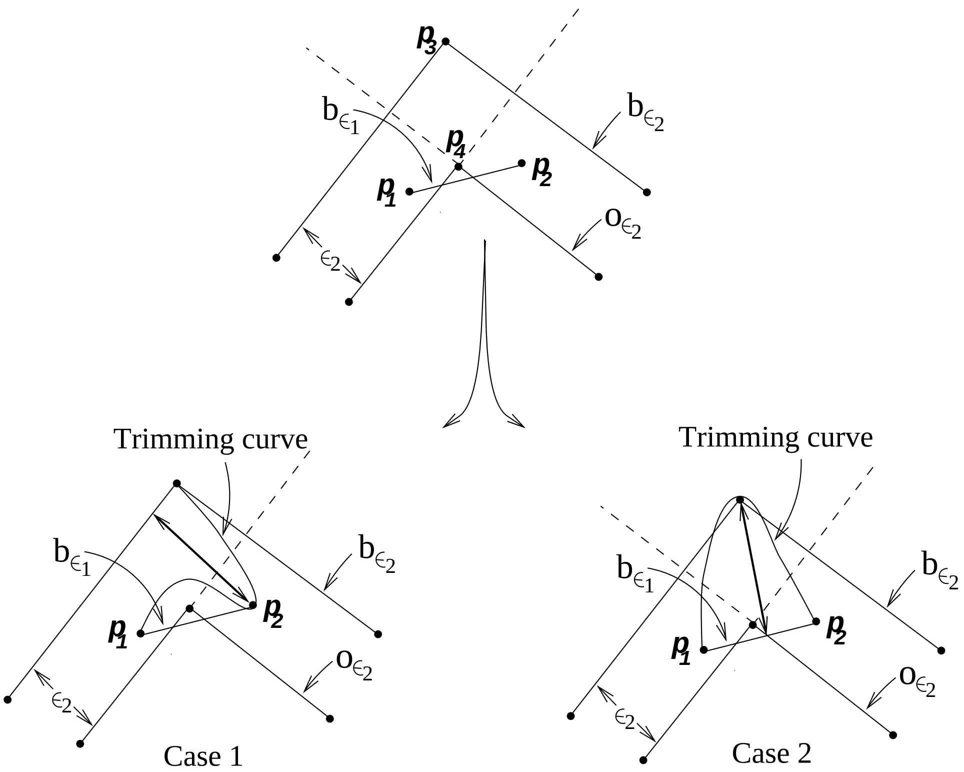

Figure 10: Offset Curve Intersection